

Dictionary of Debugging



Concept by
Dmitry Vostokov

Dictionary of Debugging

Software Diagnostics, Observability, and Debugging

1,385 entries • 23 domains • 884 primary references

Version 8
23 October 2026

This dictionary connects the language of software diagnostics, observability, and debugging with the artifacts, structures, behaviors, mechanisms, and analysis activities it describes. It spans native and managed execution, distributed systems, telemetry, testing, and AI diagnostics, with explicit coverage of pattern-oriented methods and software narratology.

Each entry gives a headword, an expansion where appropriate, a domain tag, a concise definition, direct references, and related terms. We distinguish what an artifact shows, which pattern we recognize, and which mechanism still requires investigation.

Find a term

7 8 A B C D E F G H I J K L M N O P Q R S T U V W Y Z

Coverage, 4 Version 2 expansion, 5 Ambiguity guide, 266 Alias index, 269 Domain index, 273 Standards and formats, 282 Primary references, 283

Reading conventions

Headwords use letter-by-letter alphabetical order, ignoring spaces and punctuation for sorting. Capitalization is retained for named patterns and tools. A shared spelling can have several meanings; the domain tag resolves the intended sense. Each entry has one primary tag even when its use crosses domains.

References identify the basis for each original summary. Book locators use printed page numbers. Related terms link to other entries; reference codes link to the bibliography. Index page locators support print use, and bibliography URLs remain clickable in digital copies. Tool names have expansions only where documented. Version and platform qualifications remain part of the definition.

Mathematical, physical, and narrative analogies are identified as such. Named proposals are distinguished from established formats and supported tools. Recognizing a pattern organizes the evidence; it does not, by itself, establish a unique cause.

Dictionary of Debugging

Software Diagnostics, Observability, and Debugging

Concept by Dmitry Vostokov

Developed with AI assistance (GPT-6 Astra and Fable 5.1)

Published by OpenTask, Republic of Ireland

Copyright © 2026 by Dmitry Vostokov

Copyright © 2026 by OpenTask

Copyright © 2026 by Software Diagnostics Institute

ISBN-13: 978-1-906717-35-3

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the publisher's prior written permission.

Product and company names mentioned in this book may be trademarks of their owners.

OpenTask books are available through booksellers and distributors worldwide. For further information or comments, send requests to press@opentask.com.

Version 8
October 2026

Origins and editorial basis

The early Dictionary of Debugging blog entries were initial attempts to develop this dictionary. The first draft entry, “Hang,” appeared on 23 June 2009 and announced the forthcoming book under ISBN 978-1-906717-35-3. The present volume expands these beginnings across software diagnostics, observability, and debugging (R424, opening announcement).

The diagnostic vocabulary and narrative approach are grounded in Theoretical Software Diagnostics, Fourth Edition, and Trace, Log, Text, Narrative, Data, Fifth Edition, with original summaries and page-level references. Version 2 expands the early blog entries and indexed memory and trace catalogs. Version 3 refines definitions, citation locators, source records, and cross-references. Version 4 clarifies the R171 preprint attribution. Version 5 adopts the present title and subtitle. Version 6 clarifies unit testing and specialist abbreviations, adds print page locators and visible source URLs, and pins selected documentation versions. Version 7 corrects the DWARF reference title and updates the document properties. Version 8 places the running headers on a single line.

The document follows the compact reference structure of Electronic Design Automation Acronym Dictionary, Version 7: a 7 × 10-inch page, two-column alphabetical entries, domain labels, running headwords, an ambiguity guide, domain navigation, standards and formats, and primary references.

Platform and technology entries draw on official specifications, tool documentation, author publications, and primary research. Source links and bibliographic records were checked on 23 October 2026. Some publishers restrict direct access to full articles; the cited record identifies the work independently of that access condition.

Product and company names may be trademarks of their respective owners. Source publications retain their own rights. The references attribute the concepts and technical material on which the summaries are based.

Coverage and domain tags

Tag	Coverage	Entries
Foundations	Diagnostic reasoning, pattern languages, mechanisms, and problem solving	92
Narratology	Software narratives, interpretation, models, and visual representation	21
Trace patterns	Named patterns for trace, log, text, and narrative analysis	268
Memory patterns	Behavioral, structural, and malware-related memory analysis patterns	433
Telemetry	Instrumentation, collection, transport, and evidence quality	23
Logs	Records, timestamps, severity, structured logging, and retention	12
Metrics	Measurements, aggregation, histograms, rates, and queries	24
Tracing	Spans, context, correlation, sampling, and distributed traces	16
Performance	Latency, throughput, utilization, queues, and resource analysis	32
Reliability	Objectives, alerts, incidents, budgets, and recovery measurements	26
Distributed systems	Causality, clocks, retries, deadlines, and partial failure	17
Networks	Transport behavior, packet evidence, DNS, HTTP, and TLS	9
Data systems	Queries, database activity, locks, storage, and consumer lag	10
Cloud	Containers, orchestration, resource limits, and health probes	13
Memory	Address spaces, allocation, lifetime, corruption, and pressure	52
Execution	Threads, synchronization, scheduling, exceptions, and interrupts	55
Artifacts	Dumps, symbols, object formats, and captured execution state	32
Debuggers	Execution control, inspection, replay, elementary actions, and debugware	68
Native tools	Windows, Linux, and macOS diagnostic tools and facilities	23
Program analysis	Static and dynamic reasoning, slicing, models, and profiling	44
Testing	Reproduction, oracles, coverage, reduction, fuzzing, and injection	37
Runtimes	JVM, .NET, Python, JavaScript, Go, and managed artifacts	50
AI and ML	Evaluation, drift, attribution, model behavior, and agent diagnostics	28

Counts distinguish meanings: each separately defined sense contributes one entry. The dictionary covers all 412 named senses in the indexed memory catalog and all 266 named patterns in the trace catalog. Additional structural and malware patterns share the Memory patterns tag. Aliases preserve catalog qualifications where the existing entry already describes that sense. Platform coverage remains selective for commands, event identifiers, and product-specific codes.

Version 2 expansion

The DumpAnalysis.org audit compared named catalog entries and their primary articles with Version 1. Counts below refer to catalog coverage, so shared entries across families are not additive. The audit date is 23 October 2026.

Catalog or family	Coverage	Version 2 treatment
Dictionary of Debugging blog entries	14 headword articles	9 new entries; 5 existing entries enriched
Memory analysis	412 catalog senses	370 new entries; 42 existing senses retained
Trace and log analysis	266 catalog patterns	30 new entries; 236 catalog entries retained
Diagnostic antipatterns	14 antipatterns	All 14 added
User-interface problems	2 patterns	Both added
Structural memory	8 patterns	All 8 added
Malware analysis	24 catalog names	11 added; 13 shared with memory analysis
Workarounds	5 patterns	All 5 added
DebugWare	13 patterns	All 13 added
Elementary analysis	18 named actions	All 18 given individual entries

Further additions include Spiking Interrupts (Linux), Code Path Locality, and Dereference Fixpoint, found beyond the main catalog links. The trace domain retains the separate Trace Inductance and Trace Capacitance entries alongside the catalog's Trace Reactance entry. Unified debugging patterns, Loop Construct, acquisition-pattern overviews, ADDR, and the proposed MCP analysis-pattern tools extend the methodological coverage.

A catalog name may point to an existing headword or a qualified alias. Distinct meanings remain separate: examples include Frozen Process as an observed state and as an intentional workaround, and System Call as an execution concept and as an analysis pattern. Announcements and promises of future patterns are not treated as defined entries.

Catalog sources: Early Dictionary of Debugging entries; Memory analysis; Trace and log analysis; Related catalogs. Individual entries carry their own primary references.

Alphabetical dictionary

7

7

Memory. A hexadecimal boundary cue in the equally divided 32-bit virtual-memory model: the user-space interval ends at 0x7FFFFFFF before kernel space starts at 0x80000000. The earlier Dictionary of Debugging blog entry illustrates this through Windows stack and module addresses. We treat the digit as a mnemonic for that layout; address interpretation still requires the actual architecture, configuration, and mapping.

Ref. R414, Definition and examples. See also: 8; User space; Kernel space; Virtual memory.

8

8

Memory. The leading hexadecimal digit at 0x80000000, the beginning of kernel space in the equally divided 32-bit virtual-memory model illustrated in the earlier Dictionary of Debugging blog entry. It complements the user-space boundary cue 7. We retain the layout qualification when reading addresses; the article explicitly shows that the same digit convention does not carry over to its x64 example.

Ref. R412, Definition and examples. See also: 7; Kernel space; User space; Virtual memory.

A

Abduction

Foundations. Reasoning from observed diagnostic indicators to a plausible explanation of how they arose. We use it when the available artifacts are incomplete and several mechanisms remain possible. An abductive result is a hypothesis to investigate: subsequent observations, reproductions, or controlled changes should discriminate it from competing explanations.

Ref. R001, p. 136. See also: Diagnostic hypothesis; Diagnostic evidence; Causal model.

ABI Application binary interface

Execution. The binary-level agreement that allows independently compiled components to interact. It includes matters such as data layout, calling conventions, and relevant platform rules. Diagnostic reconstruction depends on using the ABI of the captured component, including mixed-architecture cases. A correct symbol name with an incorrect structure layout can still lead us to read the wrong fields and infer nonexistent corruption.

Ref. R079, Definitions; Base requirements; Integer registers. See also: Calling convention; PDB [Artifacts]; DWARF.

Abnormal Value

Trace patterns. An unexpected value inside an otherwise familiar message structure. We can recognize the Message Invariant and still find a network address, size, status, or identifier outside its expected range. The value directs further investigation; its abnormality depends on the relevant execution and configuration context.

Ref. R002, p. 33. See also: Message Invariant; Significant Event.

Abridged Dump

Memory patterns. A dump deliberately contains only part of the potentially capturable state, such as selected registers, stacks, memory, or timing information. We identify the acquisition scope before choosing an analysis method and use the surviving evidence where possible. Omission limits the questions the artifact can answer; it does not make the dump useless or guarantee that sensitive contents are absent.

Ref. R632, Abridged Dump pattern and diagnostic examples. See also: Minidump; Small memory dump; Truncated Dump.

Abstract interpretation

Program analysis. A framework for analyzing program behavior through a simpler mathematical domain, such as intervals instead of individual integer values. Transfer operations approximate the effect of statements, and iteration finds stable facts. A sound overapproximation includes every concrete behavior covered by its assumptions; consequently, a reported possibility may be infeasible. Diagnostic interpretation requires knowing the abstraction and the information it discards.

Ref. R136, Abstract; abstract semantics and fixpoints. See also: Static analysis; Data-flow analysis; Alias analysis.

Access violation

Execution. A Windows exception caused by an access that is not permitted under the current memory mapping or protection, such as an invalid read, write, or instruction fetch. We examine the operation, target address, and exception context together. The faulting module identifies where the processor detected the access; it does not alone identify who supplied or corrupted the address.

Ref. R116, Members: ExceptionCode (EXCEPTION_ACCESS_VIOLATION); ExceptionInformation; R012, Introduction: virtual address spaces and hardware page protection. See also: Null pointer dereference; Page fault; Exception record.

Accidental Lock

Memory patterns. A lock observed in a dump that is incidental to the reported delay rather than its sustained cause. Automated analysis may emphasize the lock simply because a thread happened to encounter it during collection. We inspect ownership, waiting duration, contention, and the owner's work before extending the observation into an explanation. Repeated captures can help distinguish transient synchronization from a persistent obstruction.

Ref. R364, Opening incidental-lock discussion and automated hang-analysis examples. See also: High Contention; Waiting Thread Time; Wait Chain.

ACP Artifacts, Checklists, Patterns
Foundations. An iterative root cause analysis framework that connects available artifacts, analysis checklists, and diagnostic patterns. Checklists help us choose what to collect and inspect; patterns give observations a reusable structure. Repeated passes can expose missing artifacts or overlooked indicators. The framework was later extended by making defect mechanisms explicit.

Ref. R001, p. 126; pp. 239–241. See also: APM; Pattern-oriented diagnostics; Diagnostic artifact.

Activation Context
Memory patterns. An inconsistency in Windows activation-context handling, such as deactivating a context with a cookie that does not match the current activation stack. We correlate the side-by-side exception code, the deactivation call, and the thread's context records. The reported runtime failure directs attention to activation order and context lifetime rather than to an arbitrary faulting instruction.

Ref. R524, Pattern description: Activation Context. See also: Invalid Parameter (runtime function); Nested Exceptions (unmanaged code); Thread Age.

Active memory dump
Artifacts. A Windows system-dump option that filters a complete-memory-style capture to retain memory considered useful for host diagnosis. It includes selected user and kernel state while excluding categories such as free pages and guest virtual-machine memory. We account for that filtering when a desired object is missing. The artifact’s usefulness depends on whether the retained scope matches the investigation.

Ref. R067, Included and excluded pages. See also: Complete memory dump; Kernel memory dump; Memory map.

Active Space
Memory patterns. The execution domain in which a thread's observed CPU activity is concentrated, inferred from user-mode and kernel-mode time together with its stacks. We compare time changes across captures where available and obtain complementary artifacts when the active domain is missing. Lifetime totals and a single current wait must be interpreted against the incident's actual interval.

Ref. R562, Pattern description: Active Space. See also: Spiking Thread; Dual Stack Trace; Data Correlation (CPU times).

Active Thread (Mac OS X)
Memory patterns. A thread in a Mac OS X core appears to be executing rather than waiting or suspended. This identifies activity at capture time and helps select stacks for investigation. It does not measure sustained processor consumption. We seek CPU accounting or comparable observations from additional captures before describing the thread as a spike or a persistent performance offender.

Ref. R662, Active Thread (Mac OS X) pattern and diagnostic examples. See also: Active Thread (Windows); CPU time; Spiking Thread.

Active Thread (Windows)
Memory patterns. A Windows thread is captured executing within a function instead of sitting in an obvious wait or API boundary. We examine its CPU history and the event that caused collection to decide whether that activity explains the incident. An active instruction in one snapshot does not establish a CPU spike, especially when a debugger event determined the capture instant.

Ref. R800, Crash Dump Analysis Patterns (Part 232). See also: Spiking Thread; Busy System; Evental Dumps.

Activity Disruption

Trace patterns. Interruption or degradation of a recognizable activity, indicated by errors, gaps, or the disappearance of its messages. A few isolated errors do not necessarily explain complete loss of functionality. We compare their distribution with the activity that should have continued and with the reported time of the problem.

Ref. R002, pp. 34–35. See also: Activity Region; Discontinuity; No Activity.

Activity Divergence

Trace patterns. A change from several visible threads or adjoint threads of activity to fewer continuing activities. The pattern may suggest a hang or other change of execution when the missing activities were expected to continue. A change in tracing scope can produce a similar appearance and must be considered.

Ref. R002, p. 36. See also: Thread of Activity; Adjoint Thread of Activity; Focus of Tracing.

Activity Overlap

Trace patterns. Overlap between parts of activities that were expected to occur separately or in a particular sequence. Concurrent session initialization is one example. The overlap supplies a place to investigate synchronization and state transitions; it becomes evidence of a defect only when the required ordering is established.

Ref. R002, pp. 37–38. See also: Use Case Trail; Message Set; Event Sequence Order.

Activity Packet

Trace patterns. A set of messages sharing a time value, possibly after rounding to a selected resolution. The packet groups contemporaneous observations for comparison across activities. Equal displayed timestamps do not establish simultaneous execution: the grouping also depends on clock resolution and the chosen rounding rule.

Ref. R002, p. 39. See also: Activity Quantum; Message Set; Time Scale.

Activity Quantum

Trace patterns. A continuous local run of messages from the same thread of activity around a selected message. Its extent is determined by the observed trace and the analysis context. It is independent of the operating system's scheduling quantum and can include execution on different processors.

Ref. R002, p. 40. See also: Message Context; Thread of Activity; Activity Packet.

Activity Region

Trace patterns. A region of a trace with a higher message rate than its temporal surroundings. Such a region can identify session initialization, application launch, or another concentrated activity. We inspect its contents and boundaries before interpreting the density as useful work, repeated failure, or excessive tracing.

Ref. R002, p. 41. See also: Statement Density and Current; Characteristic Message Block.

Activity Resonance

Memory patterns. Interacting activity from products with overlapping functions produces a system-wide performance problem. The original example involves competing security products whose active threads occupy the available processors while other work stalls. We correlate the products, their call paths, and processor activity to investigate their interaction; the presence of several security modules by itself does not establish such interference.

Ref. R534, Pattern description: Activity Resonance. See also: Spiking Thread; Busy System; Problem Module.

Activity Theatre

Trace patterns. A higher-level grouping of messages according to the activity roles they express. The grouping need not follow component boundaries: one component can participate in several roles. Once a recognizable activity script is identified, we can compare its realization in different regions or different traces.

Ref. R002, p. 42. See also: Motif; Use Case Trail; Inter-Correlation.

ADC Abstract Debugging Commands

Foundations. A vocabulary of debugging operations whose purpose remains recognizable across different debugger command languages. Listing threads, examining a stack, or inspecting memory becomes an abstract operation with a platform-specific realization. The abstraction helps transfer analysis knowledge while preserving the concrete command semantics and prerequisites needed for a particular target.

Ref. R001, pp. 267–270. See also: Elementary analysis pattern; Pattern repertoire; Debugging.

ADDR Accelerated Disassembly, Reconstruction and Reversing

Program analysis. A training and pattern catalog for recovering program structure and behavior from machine-level representations. Its patterns range from pointer interpretation and execution context to function structure, calling relationships, data access, and control paths. We use the catalog to organize reconstruction steps while distinguishing recovered evidence from assumptions about the unavailable source program.

Ref. R001, p. 174. See also: Disassembly; Stack Trace [Memory patterns]; Analysis pattern.

Address Representations

Memory patterns. Different descriptions of the same storage location connect symbolic variables, register-relative operands, and raw memory. We compare debugger variable-location information with disassembly to identify which local object an instruction accesses, or which stack value corresponds to it. The mapping depends on valid symbols and the current execution context, especially when optimization changes where variables reside.

Ref. R571, Pattern description: Address Representations. See also: False Local Address; False Function Parameters; Pointer Cone.

AddressSanitizer

Program analysis. A compiler-and-runtime instrument that detects selected memory-access and lifetime errors, including out-of-bounds access and use after free. Its report relates a failing access to available allocation, deallocation, and execution context. The report is evidence from an instrumented execution. Unsupported code, unexercised paths, and errors outside the enabled checks remain beyond that result, while instrumentation changes memory layout and execution cost.

Ref. R176, Introduction; usage and limitations. See also: MemorySanitizer; LeakSanitizer; Hardware-assisted AddressSanitizer.

Address space

Memory. A set of addresses interpreted within a particular execution context. A process address space organizes code, stacks, heaps, mapped files, and other regions. An address printed in a log is therefore incomplete evidence without its process, architecture, and relevant time. We reconstruct the surrounding mapping before deciding which object occupied that address or whether the address was valid.

Ref. R011, Virtual address space and address translation. See also: Virtual address; Memory map; Debugger context.

Adjoint Message

Trace patterns. A change of trace perspective in which an attribute, such as a module name, becomes the message being analyzed and the original text becomes an attribute. This projection exposes sequences and repetitions that are difficult to see in full message text. The original correspondence should remain available.

Ref. R002, pp. 43–44. See also: Adjoint Thread of Activity; Message Metadata; Quotient Trace.

Adjoint Space

Trace patterns. Memory evidence collected alongside a trace and used to follow references or inspect state that the trace does not contain. A dump may extend a logged address into an object or linked structure. Conversely, a trace may supply earlier changes that cannot be recovered from the memory snapshot alone.

Ref. R002, pp. 45–47. See also: Memory dump analysis; Data Flow; State Dump.

Adjoint Thread of Activity

Trace patterns. An ordered activity obtained by holding an attribute other than the thread identifier constant, such as a process, module, function, or data value. Messages can therefore come from several execution threads. An adjoint thread ID (ATID) identifies the grouping used for this analytical activity. Filtering constructs the analytical thread; it does not create an additional operating system thread.

Ref. R002, pp. 48–52; R884, Glued Activity, PDF p. 63. See also: Thread of Activity; Adjoint Message; Braid of Activity.

Affine Thread

Memory patterns. A thread's processor affinity restricts where it can run. If those processors are occupied by higher-priority execution or a prolonged dispatch-level activity, the thread may make little progress while other processors remain available. We inspect the affinity mask together with processor state and dependent waits; system-wide idle capacity does not guarantee service to this particular thread.

Ref. R740, Crash Dump Analysis Patterns (Part 68). See also: Thread Starvation (normal priority); Dispatch Level Spin; High Contention (processors).

AFL++

Testing. A fuzzing framework that uses instrumentation and other execution modes to guide mutations toward interesting program behavior. A campaign combines target configuration, an initial corpus, execution feedback, and failure detection. Its outputs include inputs requiring separate reproduction and interpretation. Reported crashes can share one mechanism or arise from harness problems, so raw crash counts do not directly measure distinct defects.

Ref. R161, Official documentation; instrumenting and fuzzing targets. See also: Fuzzing; Coverage-guided fuzzing; Test-case reduction.

Agentic debugging

AI and ML. A descriptive term for a debugging workflow in which a model repeatedly selects actions, receives tool observations, and chooses subsequent actions toward an objective. This extends a single generated suggestion into an experimental sequence. Its conclusions depend on the observations actually obtained, while permissions and stopping rules constrain execution. A coherent action narrative can still follow an incorrect assumption unless results are checked.

Ref. R245, Interleaved reasoning, actions, and observations; application here to debugging. See also: Tool-augmented language model; LLM-assisted debugging; Indirect prompt injection.

Aggregated Frames

Memory patterns. Occurrences of functions or modules are counted across a collection of stack traces. The resulting table exposes frequently represented frames in samples, allocation records, or a dump's thread population. We state what was collected and how it was counted. Aggregation removes execution ordering; frequency in the collection does not automatically measure elapsed time or identify a causal offender.

Ref. R683, Aggregated Frames pattern and diagnostic examples. See also: Stack Trace Collection; Allocation profiling; Unified Stack Trace.

Aggregate Snapshot

Memory patterns. A single artifact assembled from several memory snapshots or captured regions. A minidump containing selected ranges and a trace file collecting structured records illustrate the composition. We identify the constituent snapshots and their acquisition context before interpreting the aggregate. Sharing one container does not establish that every included observation describes the same instant or complete system state.

Ref. R827, Pattern discussion. See also: Memory Snapshot; Snapshot Collection; Minidump.

Aggregation
Metrics. Combining observations across time, attributes, or instruments into a reduced representation, such as a sum, count, or histogram. Aggregation makes larger behavior visible by removing detail. Its operation must match the metric semantics and preserve the distinctions needed for the question. After dimensions or individual observations have been discarded, a later query cannot reliably reconstruct them from the aggregate alone.

Ref. R268, Aggregation; Views. See also: Histogram; Label; Recording rule.

Agile diagnostics
Foundations. An iterative approach to diagnostic work in which useful findings, artifact requests, and revised hypotheses are delivered as the investigation progresses. Pattern catalogs and checklists support consistency, while expert judgment, review, and collaboration improve the analysis. The emphasis is on learning from each iteration and making the next diagnostic action explicit.

Ref. R001, p. 173. See also: Diagnostic report; APM; Analysis narrative.

Alert
Reliability. A signal that a monitored condition requires attention or action. An alert connects an observation to an operational decision; it is not a diagnosis of the underlying mechanism. Useful alerts identify impact or a concrete risk and give the responder enough context to investigate. Detection, firing, notification delivery, acknowledgement, and response are separate stages that can fail or be delayed independently.

Ref. R286, Alerting; R287, Alerting (pending and firing states; context and notification routing). See also: Alerting rule; Incident; Runbook.

Alert fatigue
Reliability. Reduced ability or willingness to respond effectively when alerts are frequent, repetitive, or insufficiently actionable. We examine which notifications lead to useful action and which duplicate known conditions. Reducing noise should preserve evidence of new or materially changed incidents and help responders maintain an appropriate sense of urgency.

Ref. R285, Tying These Principles Together. See also: Alert grouping; Alert inhibition; Runbook.

Alert grouping
Reliability. Combining related alerts into a shared notification according to configured attributes and timing. Grouping can make a widespread incident easier to recognize and reduce repeated interruptions. Its chosen dimensions also shape what responders see together. A group is an operational presentation of matching alerts, not proof that all of its members share one mechanism or should receive the same corrective action.

Ref. R276, Grouping. See also: Alert inhibition; Alert fatigue; Incident.

Alerting rule
Reliability. A configured condition evaluated against monitoring data to determine whether an alert is active. In Prometheus, an optional for duration requires the expression to remain active before it becomes firing. Evaluation does not itself guarantee notification delivery. The rule's expression, labels, time behavior, and handling of absent data determine which events are detected and which apparent recoveries end the alert.

Ref. R275, Defining alerting rules. See also: Alert; PromQL; Alert inhibition.

Alert inhibition

Reliability. Suppressing selected alert notifications because another active alert indicates a related condition that makes those notifications redundant. In Alertmanager, inhibition is defined through matching source and target labels. It can reduce cascades of notifications, but its dependency assumptions must be correct. Inhibition changes notification behavior; it does not repair the system or establish that every suppressed symptom has the same cause.

Ref. R276, Inhibition. See also: Alert; Alert grouping; Silence.

Alias analysis

Program analysis. Analysis of whether different expressions may refer to overlapping memory. Its results constrain optimizations and analyses that follow reads and writes. A may-alias result leaves an overlap possible; it does not demonstrate that the observed execution used the same storage. In debugging, alias information helps explain how a write through one reference can affect a value inspected through another.

Ref. R137, The AliasAnalysis interface; alias results. See also: Data-flow analysis; Program slicing; Static analysis.

Alien Component

Foundations. Attributing a problem to an unfamiliar component after familiar analysis techniques have failed to explain it. The component's organizational origin or lack of recognition becomes a substitute for evidence. We instead establish its participation in the observed path and test a specific mechanism. Exhausting a personal command repertoire does not establish that another component is responsible.

Ref. R863, Antipattern discussion. See also: Diagnostic hypothesis; Unknown Component; Root cause analysis.

Allocation profiling

Runtimes. Collection of information about where, how much, or how often memory is allocated during a workload. It can reveal allocation-heavy paths that a single heap snapshot misses after objects have been reclaimed. A profile's meaning depends on whether it records allocated bytes, surviving bytes, counts, or samples. Allocation origin and the later reason for retaining an object are separate questions.

Ref. R230, Heap and allocs profiles. See also: Retained size; tracemalloc; Massif.

Analysis narrative

Narratology. A record of how an investigation developed: which artifacts were examined, which patterns were recognized, which hypotheses were rejected, and why subsequent actions were taken. It is a narrative about diagnostic analysis rather than the original execution alone. Retaining revisions and uncertainties makes the reasoning available for review and further investigation.

Ref. R001, pp. 202–205. See also: Pattern narrative; Diagnostic report; CoTrace.

Analysis pattern

Foundations. A recurrent method of extracting or interpreting diagnostic indicators in a particular context. The pattern describes how to investigate; its application can reveal a problem pattern or normal structure and behavior. A catalog entry may present a general technique, a concrete tool-based realization, or both, so we identify which level is intended.

Ref. R001, pp. 216–219; p. 352. See also: Analysis Pattern Square; Diagnostic indicator; Problem pattern.

Analysis Pattern Agent

AI and ML. An agentic implementation behind a Diagnostic Analysis Pattern Tool. Externally it presents one callable method; internally it may plan operations, invoke other tools, generate helpers, and assess evidence. The proposal makes that implementation subordinate to a defined analysis pattern. We evaluate the returned evidence and method rather than treating the agent's conclusion as self-validating.

Ref. R880, Vocabulary; agents and helpers. See also: Diagnostic Analysis Pattern Tool; Agentic debugging; Diagnostic evidence.

Analysis pattern duality

Foundations. A relationship obtained when a problem pattern and one of its contextual parameters exchange diagnostic roles. Insufficient memory affecting stack growth and stack exhaustion reported as insufficient memory illustrate different causal readings. Exploring the dual helps generate alternative mechanisms; the reversal is a method of inquiry and requires independent evidence in each case.

Ref. R001, p. 313. See also: Defect mechanism; Analysis Pattern Square; APM.

Analysis Pattern Square

Foundations. A classification that separates general problem patterns, concrete problem patterns, general analysis patterns, and concrete analysis patterns. One axis distinguishes what is observed from how it is investigated; the other distinguishes abstraction from a particular realization. The square helps us transfer knowledge across products without confusing a tool command with the phenomenon it reveals.

Ref. R001, pp. 216–218. See also: Analysis pattern; Problem pattern; Concrete analysis pattern.

Analysis Summary

Memory patterns. A compact account of diagnostic findings, connecting the observed symptoms, supporting patterns, and remaining uncertainties. A language model can help turn debugger output into such a narrative, but we check its claims against the source evidence. The summary should preserve qualifications and distinguish a reported status or module association from a verified mechanism, defect, or security conclusion.

Ref. R820, Crash Dump Analysis Patterns (Part 294). See also: Annotated Stack Trace; Diagnostic evidence; Dump Context.

Analytic memory dump

Artifacts. An interpreted representation of memory containing values paired with symbolic descriptions, potentially acquired at different times. It makes selected structure and meaning explicit instead of retaining only raw bytes. We still need acquisition context and provenance: an analytic representation may omit information required for another interpretation or combine observations that were never simultaneous.

Ref. R001, p. 188. See also: Memory dump analysis; Diagnostic artifact; General trace.

!analyze

Debuggers. A Windows debugger extension command that analyzes an exception or bug check and presents a structured diagnostic summary. Verbose output can include interpreted parameters, stack information, and suggested fault associations. We use this summary as an entry point to the underlying evidence. A named module or failure bucket is a classification result; further reconstruction is needed before assigning the mechanism or responsibility.

Ref. R107, General parameters; -hang. See also: WinDbg; Bug check; Exception record.

Anchor Messages

Trace patterns. Selected messages used to divide a long trace into regions. Process attachment, session boundaries, or other recognizable events can provide anchors. Different anchor sets produce different partitions, so the chosen partition should follow the diagnostic question while retaining enough surrounding context to interpret each region.

Ref. R002, pp. 53–55. See also: Trace Partition; Vocabulary Index; Message Context.

Anchor Region

Memory patterns. A region containing the metadata or starting structure needed to interpret a larger memory artifact. Module information and debugger data blocks are examples. We locate and validate this interpretive anchor before relying on the structures reached through it. Damage or omission can make otherwise present bytes difficult to interpret without establishing that those bytes are themselves corrupt.

Ref. R832, Pattern discussion. See also: Memory Snapshot; Corrupt Dump; Truncated Dump.

Annotated Disassembly (JIT .NET code)

Memory patterns. Native disassembly of JIT-compiled .NET code is enriched with runtime information about methods, types, or call targets. The annotations connect machine instructions to managed execution. We recover missing method identities through runtime metadata and check the debugger's display settings when the view is incomplete. An unannotated instruction listing does not by itself mean that the metadata is unavailable.

Ref. R649, Annotated Disassembly (JIT .NET code) pattern and diagnostic examples. See also: JIT Code (.NET); Annotated Disassembly (unmanaged code); SOS.

Annotated Disassembly (unmanaged code)

Memory patterns. Disassembly includes symbolic annotations that associate instruction operands with native variables or fields. The published WinDbg example shows a disassembly window exposing names unavailable in the corresponding plain command output. We use these annotations to connect machine operations to source-level data, while checking that symbols, code version, and the current frame support the interpretation being displayed.

Ref. R772, Crash Dump Analysis Patterns (Part 151b). See also: Reduced Symbolic Information; Frame Trace; Frame Pointer Omission.

Annotated Stack Trace

Memory patterns. A stack trace is accompanied by explanations of its frames and the operation they collectively suggest. We use annotations to communicate unfamiliar paths and relate component roles, including when a language model supplies an initial interpretation. The explanation must remain tied to verified symbols and context, because plausible descriptions can misidentify a subsystem or imply behavior the stack does not establish.

Ref. R819, Crash Dump Analysis Patterns (Part 291). See also: Analysis Summary; Stack Trace [Memory patterns]; Message Annotations.

Anomaly detection

AI and ML. Identification of observations that depart from a learned or specified pattern of ordinary behavior. In diagnostic use, the result prioritizes cases for interpretation; unusual behavior may be legitimate, and a familiar failure may not be unusual. The training population, feature representation, threshold, and distinction between outlier and novelty detection determine what the detector can meaningfully call anomalous.

Ref. R232, Novelty versus outlier detection. See also: Clustering; Data drift; Precision.

Anonymous memory

Memory. Virtual memory whose contents are not supplied by an ordinary file mapping, such as many heap and stack allocations. Its backing and residency are managed by the operating system and may change during execution. A rise in anonymous resident memory narrows an investigation toward process data, but we still need allocation and ownership evidence to distinguish useful retained state from a leak.

Ref. R058, DESCRIPTION; MAP_SHARED; MAP_PRIVATE; MAP_ANONYMOUS. See also: Heap; Stack; Memory leak.

API Query

Debuggers. A diagnostic utility that queries an API used by the investigated product and records the resulting information independently. It can expose environmental state or API behavior absent from the product's own logs. We correlate the query's inputs, time, and execution context with the product activity; a separate query need not observe exactly the same conditions.

Ref. R850, Pattern discussion. See also: Debugware; Telemetry; Observer effect.

APM Artifacts, Patterns, Mechanisms

Foundations. An iterative root cause analysis framework connecting artifacts, diagnostic patterns, and defect mechanisms. A recognized pattern provides indicators; a proposed mechanism explains how those indicators could have been produced. The explanation guides further collection, experiments, and debugging. Several mechanisms may fit one pattern, and one mechanism may generate several patterns.

Ref. R001, pp. 239–241; pp. 253–254. See also: ACP; Defect mechanism; Root cause analysis.

APN Analysis Pattern Network

Foundations. An architecture in which analysis-pattern components repeatedly process artifacts and enrich the available evidence with transformations, indicators, and derived artifacts. Later iterations operate on this expanded material. Components may use rules, machine learning, transformations, or human analysis. The neural-network comparison is architectural inspiration; an APN does not require conventional trained neurons or numerical backpropagation.

Ref. R001, pp. 343–347. See also: Analysis pattern; Software codiagnostics; Analysis narrative.

ArtDbg Artificial Debugger

Foundations. A proposed artificial-debugger approach that packages recorded debugger commands and their outputs into a replayable diagnostic representation. It can make selected analysis interactions reproducible for teaching, review, or tooling. Its fidelity is bounded by the recorded material and supported commands; replaying a command transcript does not reconstruct every state of the original process.

Ref. R001, pp. 242–243. See also: Analysis narrative; Analytic memory dump; Projective debugging.

ASLR Address space layout randomization

Memory. A family of mechanisms that varies memory placement to make addresses less predictable. Windows image rebasing and allocation randomization are related settings with distinct effects. We interpret recorded addresses using the actual module bases and memory map for that run. Comparing raw addresses across runs can be misleading, while a stable module-relative offset may identify the same instruction in matching binaries.

Ref. R129, Force randomization for images; Randomize memory allocations. See also: Module list; Build ID; Symbolication.

Assertion

Program analysis. An executable statement of a condition expected at a particular program point. A failed assertion preserves a useful boundary: the observed state does not satisfy the expressed expectation. It does not by itself identify the earlier operation that made the state invalid. Assertion availability depends on language and build settings; for example, optimized Python execution can omit assert statements.

Ref. R138, The assert statement. See also: Invariant; Contract; Test oracle.

asyncio debug mode

Runtimes. A Python execution mode that enables additional checks and observations for asynchronous code, including selected thread-affinity violations, slow callbacks, and coroutine-related problems. It helps connect scheduling symptoms with concrete operations. Thresholds and enabled diagnostic channels determine what becomes visible. A callback classified as slow needs interpretation against its work and workload; the diagnostic mode can also change timing.

Ref. R223, Debug mode; slow callbacks and coroutine diagnostics. See also: Event-loop blocking; faulthandler; cProfile.

Atomic operation

Execution. An operation with the indivisibility guarantees defined by its language, platform, or API. Its memory-ordering guarantees are a separate part of that contract. We inspect both the operation and the surrounding protocol: an atomic update to one field does not make a multi-field invariant atomic. Correct-looking individual values can still describe a state that no participant was intended to observe.

Ref. R082, API; SEMANTICS; ordering. See also: Memory barrier; Race condition; CPU register.

Attribute

Telemetry. A named value attached to telemetry to describe an operation, event, resource, or measurement. Attributes supply the context used for filtering, grouping, and interpretation. Their presence does not establish their correctness: values may be copied, truncated, redacted, or supplied by an untrusted caller. When attributes become metric dimensions, their possible combinations also determine the number of time series.

Ref. R335, Attribute. See also: Resource attributes; Cardinality; Semantic conventions.

Audit log

Logs. A record maintained to support accountability and reconstruction of relevant actions, typically including the actor, action, target, time, and outcome. It answers a different set of questions from a purely diagnostic message stream. A recorded identity may represent a service account or delegated action rather than a particular human. Interpretation also depends on coverage, access controls, and protection against alteration.

Ref. R331, Section 2.1.2 (Operating Systems), p. 2-4; section 2.3.2 (Log Protection), p. 2-9. See also: Log record; Structured logging; Log retention.

Automatic instrumentation

Telemetry. Instrumentation applied with limited or no changes to application source, commonly through an agent, library integration, or runtime-specific mechanism. It can reveal common requests and dependency calls quickly, but its coverage follows supported frameworks and execution boundaries. Application-specific decisions may remain invisible. The absence of a span therefore requires checking instrumentation support and configuration before interpreting application behavior.

Ref. R252, Zero-code instrumentation. See also: Instrumentation; Manual instrumentation; Span.

Availability

Reliability. The extent to which a service is usable under a defined measurement convention. A time-based measure divides usable time by the total measured interval, requiring clear criteria for when service is available, degraded, or unavailable. Partial failures complicate a single value because different users may experience different outcomes. The result should retain its observation boundary, eligibility rules, and treatment of planned interruptions.

Ref. R344, Understanding availability. See also: SLI; SLO; Partial failure.

Axed Code

Debuggers. A workaround that disables a problematic execution path by changing source or executable code. The intended effect is to prevent the selected behavior from occurring. We identify the removed behavior and the assumptions required by the surrounding code; a small change can still affect calling conventions, state transitions, or dependencies beyond the original symptom.

Ref. R847, Pattern discussion. See also: Workaround; Patched Code; Fake API.

Axed Data
Debuggers. A workaround that reduces or alters the extent of data exposed to a problematic execution path, for example through a size or length field. The changed metadata can prevent the offending operation from reaching certain contents. We preserve the relationship between the declared extent and the actual data, and identify which useful behavior the intervention also removes.

Ref. R849, Pattern discussion. See also: Workaround; Axed Code; Memory corruption.

B

Background and Foreground Components
Trace patterns. A distinction between components currently relevant to the investigated problem and the surrounding component activity. Foregrounding is an analytical selection that may change during investigation. Messages placed in the background should remain recoverable because a dependency or unexpected interaction can make them relevant again.

Ref. R002, pp. 57–59. See also: Focus of Tracing; Guest Component; Message Set.

Backoff
Distributed systems. A policy that delays subsequent attempts after failure, often increasing the delay with repeated attempts. It reduces immediate pressure on a dependency but does not guarantee recovery or make a repeated side effect safe. Deterministic delays can synchronize clients, while unbounded delays can exceed the useful lifetime of the operation. Interpret observed gaps in a trace against the configured retry schedule.

Ref. R321, Exponential backoff. See also: Retry; Jitter; Deadline.

Backpressure
Telemetry. Feedback by which a receiver indicates that it cannot accept data at the offered rate, causing an upstream component to slow, buffer, reject, or retry work. In telemetry pipelines, backpressure can propagate a backend problem toward the application. Buffer growth is evidence of an imbalance between arrival and service rates; it does not by itself identify which downstream resource or operation caused the imbalance.

Ref. R262, Throttling; backpressure. See also: Queue length; Exporter; Telemetry loss.

Back Trace
Trace patterns. A trace constructed by working backward from a significant or error message through relevant data and message relationships. Its purpose is to identify preceding observations that may explain the selected event. It may cross execution threads and differs from the call stack recorded at one instant.

Ref. R002, p. 56. See also: Error Thread; Causal History; Data Flow.

Baggage
Tracing. Application-defined context propagated as named properties across execution or service boundaries. Baggage can preserve a useful classification for later observations, but it is separate from the trace identifier and is not necessarily recorded as span attributes automatically. Propagated values may cross trust boundaries or reveal sensitive information. Their origin, permitted destinations, and use in grouping should remain explicit.

Ref. R265, Baggage; privacy considerations. See also: Context propagation; Attribute; Span context.

Basic Facts

Trace patterns. Facts from the problem description that can be checked against the trace, including identity, reported functionality, and observed resource behavior. This correspondence establishes whether the artifact plausibly covers the reported occurrence. A missing expected fact prompts investigation of capture timing, scope, and completeness before detailed interpretation.

Ref. R002, p. 60. See also: Identification Messages; Hidden Facts; Calibrating Trace.

Behavioral pattern

Foundations. A recurrent organization of software activity, including normal operation and recognizable deviations. It may be identified through a trace, several snapshots, or structures that retain evidence of earlier execution. We describe the observed behavior before assigning a mechanism, because similar behavior can arise from different code paths and environmental conditions.

Ref. R001, pp. 103–105; p. 152. See also: Structural pattern; Problem pattern; Defect mechanism.

Be Language

Foundations. Presenting a provisional interpretation as an established fact through categorical wording. The antipattern concerns the evidential status of the claim, rather than a prohibition on a particular verb. We distinguish observation, inferred relationship, and candidate cause in the report, and identify the evidence that would strengthen or reject the explanation.

Ref. R869, Antipattern discussion. See also: Diagnostic report; Diagnostic hypothesis; Diagnostic evidence.

Bethe Ansatz

Trace patterns. A trace-analysis pattern that models stable activities and their pairwise encounters to test whether those local interactions explain the observed global behavior. Delays, state changes, and reordered work can identify the interaction effects. Residual behavior may require collective mechanisms. The physical analogy supplies a decomposition strategy rather than a claim that the software system is mathematically integrable.

Ref. R010. See also: Adjoint Thread of Activity; Motivic Trace; Trace Multiphysics.

Bifurcation Point

Trace patterns. A point at which comparable working and nonworking traces begin to differ in a diagnostically relevant way. Subsequent changes in message content, ordering, or density can focus the investigation. The first visible difference is an analysis starting point; earlier unrecorded differences may still explain it.

Ref. R002, pp. 61–62. See also: Inter-Correlation; Critical Point; Master Trace.

Black Box

Memory patterns. Additional system or acquisition information is embedded inside the dump file and can supplement its ordinary memory contents. Examples include subsystem history exposed by suitable debugger commands and metadata written by collection tools. We inspect this evidence alongside current state and execution residue, distinguishing it from paratext supplied in separate files and checking what period it actually describes.

Ref. R569, Pattern description: Black Box. See also: Embedded Comment; Historical Information; Paratext.

Black-box monitoring

Reliability. Observing a service through externally visible behavior, such as a request and the response a user would receive. A probe covers only its route, input, identity, and time. Success can miss failures affecting other users or operations; failure can originate in the probe's own network path.

Ref. R285, Definitions. See also: White-box monitoring; Health check; SLI.

Blackout

Trace patterns. A temporary gap in the trace as a whole, caused, for example, by paused collection, missing files, or a paused system. It differs from a gap in one selected activity and from a permanent limit of instrumentation. We investigate the collection history before treating the interval as execution silence.

Ref. R002, pp. 63–64. See also: Discontinuity; Visibility Limit; Split Trace.

Blocked DPC

Memory patterns. Ordinary queued deferred procedure calls are delayed when preceding activity keeps their processor at DISPATCH_LEVEL or a higher IRQL. The pending DPC software interrupt cannot preempt that activity. We examine the queue together with processor context, IRQL, and the active stack. Here the blocked work is waiting for execution opportunity; the evidence does not imply that a DPC routine itself entered an ordinary blocking wait.

Ref. R656, Blocked DPC pattern and diagnostic examples; R094, DISPATCH_LEVEL; interrupts masked off. See also: DPC; IRQL; Dispatch Level Spin.

Blocked Queue (LPC/ALPC)

Memory patterns. Requests accumulate in an LPC or ALPC port queue while the expected processing does not advance sufficiently. We inspect queue populations at a wait-chain endpoint and relate the retained messages to server-thread activity. A large pending population identifies pressure or blockage to investigate; it must be interpreted with workload and timing evidence before assigning the responsible component.

Ref. R512, Pattern description: Blocked Queue (LPC/ALPC). See also: Wait Chain (LPC/ALPC); Network Packet Buildup; Blocked Thread.

Blocked Thread

Memory patterns. A thread whose observed inability to progress conflicts with its expected role in the problem context. Recognizing the pattern requires more than finding a waiting function: we compare the stack, dependency, and waiting duration with normal behavior. Following the responsible resource or participant can extend the investigation. The same waiting stack may describe a healthy passive thread in another application state.

Ref. R388, Opening context-dependent definition and comparison of expected thread states. See also: Passive Thread; Waiting Thread Time; Wait Chain.

Blocked Thread (hardware)
Memory patterns. A thread's progress depends on a response from hardware, as in the illustrated wait for an ACPI register-port operation during initialization. We identify the operation and its expected response before classifying the wait as abnormal. The captured wait establishes the hardware-facing dependency, while determining whether software, configuration, or the device caused the delay requires additional evidence.

Ref. R487, Pattern description: Blocked Thread (hardware). See also: Blocked Thread; Hardware Activity; Young System.

Blocked Thread (timeout)
Memory patterns. A blocked thread whose wait has a timeout, so the observed absence of progress may be temporary. We inspect the timeout value and its units together with the wait operation and elapsed observation period. A finite timeout does not prove that the application will recover promptly, but it changes the interpretation of a stack that otherwise resembles an indefinite wait.

Ref. R611, Blocked Thread (timeout) pattern and diagnostic examples. See also: Blocked Thread; Timeout [Distributed systems]; Wait Chain.

Blocking File
Memory patterns. A file operation forms a blocking point in a thread or wait chain. The stack identifies the operation, while a file object, handle, or I/O request can identify the affected file and outstanding work. We follow that dependency to its owner or completion path. A file name alone does not explain why the operation failed to progress.

Ref. R647, Blocking File pattern and diagnostic examples. See also: Wait Chain; IRP; Blocking Module.

Blocking Module
Memory patterns. A module occupies the part of a stack that initiates a wait or delay, helping localize where the thread stops making progress. We identify the blocking call and then follow the resource, owner, or external operation on which it depends. Naming the blocking module describes its role in the observed stack; responsibility may lie elsewhere in the dependency chain.

Ref. R760, Crash Dump Analysis Patterns (Part 122). See also: Wait Chain; Blocked Thread; Top Module.

Bohrbug
Foundations. An informal name for a defect whose failure can be reproduced reliably under the relevant conditions. Predictability supports controlled comparison and reduction of the reproducing case. It does not make the underlying mechanism obvious or imply that every input fails. Preserve the triggering conditions and distinguish repeated manifestation of the same defect from superficially similar failures.

Ref. R404, Software faults are soft—the Bohrbug/Heisenbug hypothesis, printed pp. 11–12. See also: Heisenbug; Reproducer; Determinism.

Bounded model checking
Program analysis. Verification that encodes executions up to a specified bound as constraints and asks whether a property can be violated. A satisfying solution supplies a counterexample within the encoding. A negative result establishes only what the chosen bounds and model justify unless additional completeness arguments apply. Loop unwinding, environment assumptions, and arithmetic semantics therefore belong with the diagnostic result.

Ref. R139, Abstract; bounded verification and assertions. See also: CBMC; Model checking; Symbolic execution.

Braid Group

Trace patterns. A braid-inspired view that combines execution threads with the processors on which the emitting code executes. Changes of processor and thread association expose scheduling structure. The pattern is an analytical analogy; a trace does not automatically supply the mathematical relations required for a formal braid group.

Ref. R002, p. 65. See also: Braid of Activity; Adjoint Thread of Activity; Trace Braidoids.

Braid of Activity

Trace patterns. A message set formed by finding an attribute value both in its usual field and in message text that refers to it. This joins direct activity with references made by other activities. The value's identity and scope require checking, especially when identifiers can be reused.

Ref. R002, p. 66. See also: Adjoint Thread of Activity; Linked Messages; Intrinsic ID.

Branch coverage

Testing. A coverage measure recording which alternatives of program decisions were exercised. It exposes untested outcomes that a line count may hide, including multiple decisions on one source line. Full branch coverage still permits missing combinations, values, and sequences. For a suspected defect, the useful question is whether the tests exercised the branch under the conditions that make its behavior significant.

Ref. R140, Branch Coverage. See also: Code coverage; MC/DC; Test oracle.

Break-in Activity

Trace patterns. A message or message group appearing just before a previously interrupted activity resumes. It identifies a possible trigger for the resumption. Temporal proximity guides a search through dependencies and data flow; the preceding message is not automatically the event that released the activity.

Ref. R002, p. 67. See also: Discontinuity; Resume Activity; Coupled Activities.

Breakpoint

Debuggers. An execution or processor-state condition arranged to divert control to a debugger. It can be implemented by inserting a trap instruction or configuring hardware support. We use the resulting stop to inspect selected state. Reaching a breakpoint establishes that its condition was encountered, while the interruption can alter timing and does not itself identify a failure mechanism.

Ref. R019, Section 5.1; R114, Section 5.1.1; hardware-assisted breakpoints; R416, Definition. See also: Conditional breakpoint; Hardware breakpoint; Single stepping.

Broken Link

Memory patterns. A damaged or unavailable link interrupts traversal of a linked structure, causing an enumeration to stop or reject an object. We validate forward and backward relationships, inspect accessible neighbors, and consider alternative discovery methods such as allocation searches. The missing connection may reflect corruption or incomplete capture, so a broken enumeration should not be interpreted as a complete population.

Ref. R787, Crash Dump Analysis Patterns (Part 193). See also: Truncated Dump; Lateral Damage; Corrupt Structure.

Buffer overflow

Memory. An operation that writes beyond the intended boundary of a buffer. The destination may remain mapped and writable, so corruption can continue without an immediate exception. We reconstruct the intended buffer extent and the actual write extent, then follow affected data or control information. The eventual failure location may belong to a consumer of damaged state rather than the overflowing operation.

Ref. R045, Description. See also: Out-of-bounds read; Memory corruption; PageHeap.

Bug check

Artifacts. A Windows kernel stop operation that records a code and parameters when the system cannot safely continue under the detected condition. Those parameters have meanings specific to the code. We use them to select an investigation path, then examine the corresponding state. A stop code classifies the detected condition; it does not automatically name the original defective component.

Ref. R089, Use WinDbg to display stop code information. See also: Kernel memory dump; Driver Verifier; !analyze.

Build ID

Artifacts. An identifier used to associate a binary with corresponding build artifacts, particularly separate debug information. GNU build IDs and Apple build UUIDs provide platform-specific examples. We use the identifier to select the correct evidence interpretation rather than rely on filenames alone. A match supports artifact association; it is not a general proof that all deployed configuration or runtime state was identical.

Ref. R025, Section 18.3; R100, Overview; Use unique build UUIDs. See also: Symbol file; PDB [Artifacts]; dSYM.

Burn rate

Reliability. The rate of error-budget consumption relative to the rate that would exhaust the budget over the objective's full period. For a ratio-based objective, it compares the observed bad-event fraction with the allowed fraction. A burn rate above one indicates faster consumption if sustained. The observation window and traffic volume matter; short windows and few requests can produce large, unstable values.

Ref. R288, Alert on Burn Rate; Low-Traffic Services. See also: Error budget; SLO; Alerting rule.

Busy System

Memory patterns. A system snapshot showing substantial processor activity, sometimes accompanied by runnable threads waiting for execution. Processor contexts, current threads, and ready queues provide a system-wide view that individual process stacks cannot supply. We relate this view to workload and the reported interval. Occupied processors can indicate useful throughput, excessive work, or contention; one snapshot does not determine which interpretation explains the symptom.

Ref. R367, Definition preceding processor-state examples and ready-thread discussion. See also: Spiking Thread; Context switch; Starvation.

C

Cachegrind

Program analysis. A Valgrind tracing profiler that produces detailed execution counts and can simulate cache and branch-prediction behavior. It supports comparisons of where instruction work changes between controlled runs. Simulated misses depend on the configured model, and instruction counts omit many causes of elapsed-time variation. The useful diagnostic claim should identify the reported event and how it relates to the performance question.

Ref. R190, Overview; instruction counts and simulation. See also: Valgrind; Callgrind; Dynamic analysis.

Cache miss

Performance. A processor-cache lookup that cannot supply the requested instruction or data at the specified cache level. A miss at one level can be satisfied elsewhere in the hierarchy, so it does not automatically imply a main-memory access. Hardware counters differ in demand, prefetch, and request coverage. Interpret miss counts with the event definition and measured stall or latency consequences.

Ref. R400, ICache Misses; L2 Miss Count; LLC Miss Count. See also: CPU utilization; False sharing; CPU profile.

Calibrating Trace

Trace patterns. A selected trace used as a reference when aligning a collection of traces with basic facts, capture intervals, and the reported problem time. It helps expose incomplete or differently timed captures. Calibration must account for clock differences and collection settings rather than simply sorting all displayed timestamps.

Ref. R002, p. 68. See also: Basic Facts; Inter-Correlation; Unsynchronized Traces.

Caller-n-Callee

Memory patterns. A caller-and-callee relationship reconstructed from call-like addresses found in raw stack memory. We inspect the instructions preceding candidate return addresses and identify their call targets, rather than accepting every symbolic address as an active frame. The resulting pairs can summarize execution residue, including on nonmanaged threads, while still requiring checks for coincidental symbolic information.

Ref. R525, Pattern description: Caller-n-Callee. See also: Execution Residue; Coincidental Symbolic Information; Stack Trace [Memory patterns].

Call graph

Performance. A graph of caller and callee relationships between functions. A graph built from runtime profiling describes observed relationships and may attach call counts or attributed costs. It supplies context that a flat function ranking omits, including recursive relationships. It is not an execution timeline: aggregated edges do not preserve every call's order, and missing coverage can remove relationships from the graph.

Ref. R395, The Call Graph. See also: Profiling; Inclusive time; Exclusive time.

Callgrind

Program analysis. A Valgrind profiler that records call relationships and execution costs, with optional cache and branch-prediction simulation. Inclusive costs include work attributed through callees, while exclusive costs describe the selected routine's own work. These views help locate where computation accumulates. Instruction counts and simulated events describe the profiling model; they are not interchangeable with elapsed time on a production processor.

Ref. R189, Call graphs; cost attribution; cache simulation. See also: Valgrind; Cachegrind; cProfile.

Calling convention

Execution. Rules describing how a function call passes arguments, returns values, preserves registers, and organizes its stack state. The convention depends on platform, architecture, and sometimes function declaration. We apply the actual convention when reconstructing a call from registers and memory. Copying an x64 Windows argument recipe to another ABI can create a plausible but incorrect interpretation.

Ref. R078, Calling convention defaults; Parameter passing; Unwindability; R079, Definitions; Base requirements; Integer registers. See also: ABI; CPU register; Stack unwinding.

Canonical Trace

Trace patterns. A trace transformed into a common representation with an agreed Trace Schema for comparison, storage, or further processing. Canonicalization makes corresponding structures easier to align while preserving the distinctions required by the analysis. It differs from a Master Trace, which supplies an expected archetype, and from a Meta Trace, which concerns trace evolution as the underlying code changes.

Ref. R428, Part 233. See also: Trace Schema; Master Trace; Meta Trace.

CAPS Cloud Analysis Patterns

Cloud. A proposed system of analysis patterns for cloud structures and behavior, extending pattern-oriented diagnostics beyond individual process and kernel spaces. Relationships among nodes, pods, containers, and control components supply additional contexts for analysis. These abstractions can guide collection, but investigating a cloud-level problem may still require ordinary process, runtime, network, and memory artifacts.

Ref. R001, pp. 333–341. See also: Systemic diagnostics; Trace Quilt; Software diagnostics engineering.

Cardinality

Metrics. The number of distinct identities represented by a metric’s dimensions, commonly the number of unique label combinations and resulting time series. Unbounded identifiers can make cardinality grow with requests or users rather than with the system’s structural components. This affects collection and query cost. Reducing cardinality also removes distinctions, so the discarded context may need to survive in logs, traces, or exemplars.

Ref. R277, Labels. See also: Label; Exemplar; Attribute.

Cartesian Trace

Trace patterns. A product-like association between a main trace and independent supplementary traces, such as configuration records. It lets us interpret activity against additional dimensions of context. The supplementary record's applicability must be established; an independently collected configuration file may describe a different interval or environment.

Ref. R002, p. 69. See also: Small DA+TA; Trace Extension; Fiber Bundle.

Cascading failure

Distributed systems. A failure that creates conditions causing other components to fail. Overload transfer, retries, and resource exhaustion can reinforce one another. The first observed failure need not be the initiating event. We reconstruct load movement, dependencies, and protective behavior to distinguish the initial disturbance from feedback sustaining the outage.

Ref. R337, Introduction; overload; retries. See also: Retry storm; Load shedding; Incident.

Case Messages

Trace patterns. Messages assigned a degree of relevance to the case being investigated, by analogy with fuzzy-set membership. This records uncertainty about which observations belong in the diagnostic view. A relevance score describes an analytical choice; it should not be confused with a calibrated probability that a message identifies the cause.

Ref. R002, p. 70. See also: Basic Facts; Message Set; Significant Event.

Catchpoint

Debuggers. A debugger stopping condition associated with an execution event, such as a thrown exception, a signal-related event, or a supported process transition. The available events vary by debugger and target. We use it to capture state near a transition that ordinary code breakpoints may miss. The captured event can be part of expected control flow, so its diagnostic meaning depends on subsequent handling.

Ref. R019, Section 5.1. See also: Breakpoint; Exception; Signal [Execution].

Causal Chains

Trace patterns. Relations between messages in a reconstructed causal history, together with relations between those relations at higher levels. Chains make the structure of an explanation explicit and can join where their endpoints agree. Each asserted link requires evidence or a stated hypothesis; visual continuity alone does not establish causation.

Ref. R002, pp. 71–72. See also: Causal History; Causal Messages; Trace Path.

Causal History

Trace patterns. A reconstruction that connects trace paths and backward investigations into a history of possible causation. The graph organizes relationships inside the software narrative. It need not use a single displayed time axis, but its links must preserve the distinction between observed ordering, established dependencies, and proposed explanations.

Ref. R002, p. 73. See also: Back Trace; Causal Chains; Trace Path.

Causal Messages

Trace patterns. Messages selected for their possible causal role within a reconstructed history. They need not be the first messages or the most conspicuous errors. We distinguish them from a group of messages merely associated with failing runs and examine the mechanism that connects them with the observed outcome.

Ref. R002, p. 74. See also: Causal History; Defect Group; Coupled Activities.

Causal model

Foundations. An explicit account of events, states, and dependencies that could explain an observed problem. Diagnostic artifacts constrain the model, while assumptions connect what was recorded to what may have happened. We distinguish temporal order and correlation from demonstrated causal dependence, and revise the model when new evidence fails to fit its predictions.

Ref. R001, p. 52; pp. 239–241. See also: Abduction; Diagnostic hypothesis; Defect mechanism.

Causal order

Distributed systems. The partial ordering of events established by local execution order, message transmission and receipt, and the transitive consequences of those relationships. Distinct events unordered by this relation are concurrent in Lamport's sense. An incomplete capture can still hide dependencies. A timestamp sort creates a sequence but does not establish all of these relationships. In distributed diagnostics, reconstruct the communication and execution relationships before interpreting apparent temporal proximity as influence.

Ref. R305, The Partial Ordering. See also: Logical clock; Vector clock; Clock skew.

CBMC C Bounded Model Checker

Program analysis. A verification tool that checks properties of C and C++ programs using bounded model checking. It can encode assertions and checks for operations such as invalid memory access, then report a violating execution or a successful bounded check. The diagnostic record should preserve unwinding bounds, assumptions, and enabled checks, because these determine the scope of the result.

Ref. R139, Tool overview; assertions and bounded verification. See also: Bounded model checking; Assertion; Model checking.

C++ Exception

Memory patterns. A C++ language exception recognized through runtime throwing and handling frames in a memory artifact. On the illustrated Microsoft implementation, a runtime throw can lead through the same operating-system exception machinery used by other software exceptions. We identify the runtime and throwing operation before interpreting the status. The top exception-raising frame alone does not explain the failed C++ operation.

Ref. R622, C++ Exception pattern and diagnostic examples. See also: Exception; Software Exception; Nested Offender.

Changed Environment
Memory patterns. A change in operating conditions that exposes a previously dormant defect or changes its manifestation. An operating-system upgrade or an additional module can alter addresses, timing, and interactions sufficiently to reveal an invalid pointer. We compare the actual environments and reconstruct the dependency on the change. The newly visible component is evidence of a difference, not automatic evidence of responsibility.

Ref. R352, Opening environmental-change discussion and relocated-module example. See also: Module Variety; Not My Version; Dangling pointer.

Chaos engineering
Testing. Experimentation that tests whether a system maintains specified behavior under realistic disturbances. The experiment states a steady-state expectation, introduces a controlled disruption, and compares observations with the hypothesis. Its diagnostic value comes from the explicit relationship between disturbance and measured effect. Scope, termination conditions, and the affected population matter because a disturbance can propagate beyond the initially selected component.

Ref. R170, Principles; hypothesis, disturbances, and blast radius. See also: Fault injection; Test oracle; Reproducer.

Characteristic Message Block
Trace patterns. A recognizable block of trace output, often visible through repetition, coloring, or a change in message structure. Blocks can contain finer or nested blocks. Their boundaries provide useful entry points for analysis, including transitions into repeated requests, retries, or error reporting.

Ref. R002, pp. 75–77. See also: Activity Region; Periodic Message Block; Message Context.

Checking Instrumentation
Debuggers. Determining which diagnostic facilities were active when evidence was produced and which information they could supply. This action can establish whether specialized checks, allocation records, or related instrumentation support the intended analysis. We retain the configuration and acquisition context, since an absent recorded observation may reflect disabled instrumentation rather than absent behavior.

Ref. R001, pp. 268–270. See also: Instrumentation; Telemetry; Negative evidence.

Checklist
Debuggers. A debugware design that coordinates reusable Troubleshooting Units of Work according to a stored checklist. The checklist describes which checks to perform, while the units implement them and contribute results. We retain that separation when adapting the workflow: changing the order or selection of checks need not require rewriting the diagnostic operations themselves.

Ref. R858, Pattern discussion. See also: Troubleshooting Unit of Work; Supporting Module; Tool Chain.

Circuit breaker

Distributed systems. A mechanism that temporarily rejects calls to a dependency after observed failures reach a policy threshold, then permits controlled attempts to determine whether service has recovered. Closed, open, and half-open states describe common implementations. The mechanism limits repeated pressure and can alter the observed error population. An open breaker is evidence of a policy decision based on preceding results, not a diagnosis of the dependency.

Ref. R320, Solution; closed, open, and half-open states. See also: Retry storm; Timeout [Distributed systems]; Partial failure.

Circular Trace

Trace patterns. A trace whose storage has wrapped or overwritten older records. Its displayed beginning may therefore differ from the earliest retained event. The pattern directs attention to timestamps, wrap metadata, and retained boundaries; file size alone cannot establish circularity or recover the overwritten history.

Ref. R002, pp. 78–79. See also: Incomplete History; Truncated Trace; Calibrating Trace.

Clang Static Analyzer

Program analysis. A source-analysis tool in the Clang project that uses path-sensitive, interprocedural reasoning to find defects in supported languages and checkers. Its reports describe a possible execution leading to a suspicious operation. The path is a diagnostic explanation generated from the analyzer's model; library models, unsupported constructs, and analysis limits must be considered before treating it as a reproduced failure.

Ref. R151, Clang Static Analyzer overview. See also: Static analysis; Symbolic execution; Taint analysis.

Clock skew

Distributed systems. A term often used operationally for disagreement between clocks on different systems. More precise clock models distinguish time offset from differences in clock rate and synchronization uncertainty. These differences can make logs appear reordered and spans appear to start before their parents. Before diagnosing impossible timing, identify the clocks, synchronization method, and timestamp correction applied by the analysis tool.

Ref. R306, NTP model; clock offset, delay, and synchronization. See also: Event timestamp; Observed timestamp; Logical clock.

Clone Dump

Memory patterns. A dump taken from a reflected or cloned process provides a snapshot of the original address space while introducing acquisition-specific thread context. We inspect the capture metadata and any companion thread information before interpreting its stacks. The original Windows example combines a clone's memory with a separate minidump to reconstruct the original thread's calls and execution residue.

Ref. R554, Pattern description: Clone Dump. See also: Manual Dump; Execution Residue; Embedded Comment.

Cloud Environment

Memory patterns. Infrastructure modules and platform metadata identify a dump as belonging to a cloud-hosted application or its development environment. We establish that context before interpreting service roles, dependencies, and expected stacks. The original examples concern early Windows Azure; their particular module names and operating-system versions are historical indicators, not a universal test for present cloud environments.

Ref. R521, Pattern description: Cloud Environment. See also: Changed Environment; Special Process; Module Variety.

ClrMD

Runtimes. The Microsoft.Diagnostics.Runtime library for programmatic inspection of .NET processes and dumps. It allows repeated diagnostic queries over runtime structures to be expressed as code, which is useful for large object populations or multiple artifacts. Such automation still depends on compatible runtime data and correctly interpreted APIs. A script's selection rules and missing-data handling determine what its result actually supports.

Ref. R203, Official repository overview. See also: SOS; dotnet-dump; Heap dump.

Clustering

AI and ML. Grouping observations according to a chosen representation and measure of similarity, usually without supplied class labels. It can organize reports, stack contexts, or incidents into recurring populations. The groups describe the algorithm's structure in the data rather than automatically identifying common causes. Feature choices, distance measures, scale, and cluster settings can split one mechanism or combine several distinct mechanisms.

Ref. R233, Clustering overview and algorithm assumptions. See also: Embedding; Feature vector; Anomaly detection.

CoActivity

Trace patterns. An activity inferred from data contained within another recorded activity. For example, values present in keyboard-related messages may reveal an additional operation. The pattern follows the activity signified by the data, whereas Data Flow follows the movement or transformation of values between messages.

Ref. R002, p. 80. See also: Data Flow; Hidden Facts; CoTrace.

C++ Object

Memory patterns. A C++ object with virtual functions can leave a recognizable virtual-table pointer in memory. We follow that evidence to inspect adjacent fields or, through an additional pointer level, a dynamically allocated instance. The technique helps recover objects from execution residue, but the candidate layout and lifetime require validation; it does not apply uniformly to every C++ object type.

Ref. R813, Crash Dump Analysis Patterns (Part 273). See also: Execution Residue; COM Object; Pointer Class.

Code coverage

Testing. A description of which program elements executed during selected tests or workloads. The element may be a function, source region, line, branch, or condition, so a percentage needs its criterion and denominator. Coverage identifies exercised structure and remaining gaps. It does not establish that the executed behavior was checked correctly or that the collected workload represents the reported problem.

Ref. R140, Interpreting reports; region and function coverage. See also: Branch coverage; MC/DC; Mutation testing.

Code Path Locality

Memory patterns. Different incidents show similar portions of their execution paths, visible in partial stacks or surviving code references. We compare those similarities with previous cases to focus a hypothesis or choose the next diagnostic step. Agreement with Data Contents Locality strengthens the comparison, while code-path resemblance alone does not establish that the same defect explains both incidents.

Ref. R739, opening discussion defining Code Path Locality. See also: Data Contents Locality; Execution Residue; Crash Signature.

Coincidental Error Code

Memory patterns. A memory value matches a familiar error code without representing an actual error occurrence. It may be part of an instruction, image data, or a table of possible statuses. We inspect the containing region and its use before treating a search hit as historical failure evidence; matching the numeric code alone does not establish diagnostic significance.

Ref. R779, Crash Dump Analysis Patterns (Part 170). See also: Coincidental Symbolic Information; Execution Residue; Regular Data.

Coincidental Frames

Memory patterns. Plausible-looking frames appear in a reconstructed stack without belonging to the actual call chain. Even symbols and nearby valid instructions can encourage a false interpretation. We compare stack links, saved context, and the faulting instruction, reconstructing the trace where necessary. A convincing routine name in a generated frame is evidence to validate before attributing corruption or failure to that routine.

Ref. R630, Coincidental Frames pattern and diagnostic examples. See also: Incorrect Stack Trace; Coincidental Symbolic Information; Stack unwinding.

Coincidental Symbolic Information

Memory patterns. A symbolic interpretation attached to a value that happens to resemble an address but has another meaning in its actual context. Text fragments, flags, or stale data may therefore appear to name a function. We inspect surrounding bytes, instruction boundaries, and the object's representation before using the name as execution evidence. Symbol resolution alone does not prove a call, reference, or ownership relationship.

Ref. R360, Opening symbolic-value discussion and raw-data examples. See also: Execution Residue; Memory examination; Incorrect Stack Trace.

Collapsed Message

Trace patterns. Several messages combined into one analytical message while retaining their content. A multiline exception report can thereby become searchable as one unit. The transformation should preserve ordering and original boundaries in a recoverable form, especially when messages from different activities are interleaved.

Ref. R002, p. 81. See also: Exception Stack Trace [Trace patterns]; Quotient Trace; Motivic Trace.

Collector

Telemetry. A service or agent that receives telemetry, applies configured processing, and exports the results. The OpenTelemetry Collector can host several pipelines, each with its own signal and component configuration. Its deployment location affects the failure domain and resource constraints of collection. A healthy application with missing telemetry may require investigation of the Collector's queues, resource use, configuration, and downstream responses.

Ref. R256, Pipelines. See also: Telemetry pipeline; Receiver; Processor; Exporter.

Combed Trace

Trace patterns. A presentation that separates interleaved trace messages into columns or lanes for individual threads, processes, or other adjoint activities. The rearrangement exposes continuity within each activity. Relationships across lanes still require the original ordering and relevant correlation information.

Ref. R002, p. 82. See also: Adjoint Thread of Activity; Message Interleave; Weave of Activity.

Combinatorial testing

Testing. Selection of tests to cover combinations of values across input factors. A covering array can exercise all interactions of a chosen size with fewer tests than the complete Cartesian product. The factor model and interaction strength define the claim. Defects depending on omitted values, stronger interactions, ordering, or evolving state may remain outside the generated suite.

Ref. R171, Interaction coverage and covering arrays. See also: Pairwise testing; Property-based testing; Code coverage.

COM Exception

Memory patterns. A COM-related failure is carried through a software exception, often with a C++ exception object that preserves an HRESULT and descriptive text. We recover the appropriate exception or object context, including nested or hidden instances, and interpret those fields together. The message and code provide application-level meaning that the generic exception-dispatch stack may not reveal.

Ref. R567, Pattern description: COM Exception. See also: Hidden Exception; Nested Exceptions (unmanaged code); Invalid Exception Information.

Committed memory

Memory. Memory for which the operating system has made a backing commitment under its allocation model. On Windows, committing a region is different from merely reserving addresses and from having every page resident in RAM. We examine commitment, access protection, and residency separately. A process can exhaust its available commitment while a simple view of resident memory still appears unremarkable.

Ref. R091, Free, Reserved, and Committed. See also: Reserved memory; Working set [Memory]; Memory pressure.

COM Object

Memory patterns. A candidate COM interface is recognized through its pointer to a virtual-function table, whose initial entries follow the IUnknown method order: QueryInterface, AddRef, and Release. We follow the pointer chain and inspect the associated memory and symbols to identify the implementation. The recognizable interface layout supports discovery but does not by itself establish object validity or lifetime.

Ref. R816, Crash Dump Analysis Patterns (Part 282). See also: C++ Object; Function Pointer; Execution Residue.

Complete memory dump

Artifacts. A Windows system dump containing the physical memory used by Windows, with platform and acquisition qualifications. Completeness here concerns a defined capture scope; it does not mean every page of every process's virtual address space or all firmware state. We still inspect unavailable mappings and capture conditions. The larger artifact expands reconstruction opportunities without supplying a history of earlier execution.

Ref. R065, Dump contents. See also: Kernel memory dump; Physical address; Memory dump.

Concept drift

AI and ML. A change over time in the relationship between observed inputs and the outcome a model is intended to predict. For diagnosis, an old symptom-to-category mapping may become unreliable after software or operating behavior changes. Input distributions can also change without this relationship changing, so the two questions should be separated. Detecting drift requires an appropriate stream, comparison, and outcome information.

Ref. R239, Concept drift and changing input-target relationships. See also: Data drift; Cross-validation; Anomaly detection.

Concolic execution

Program analysis. Execution that combines concrete values with symbolic expressions. A concrete run determines a path, while accumulated constraints describe inputs that could follow it; changing a constraint can generate a test for another path. This connects an observed execution with controlled alternatives. External calls, unsupported operations, solver limits, and execution bounds can restrict the behaviors explored.

Ref. R141, DART algorithm and directed test generation. See also: Symbolic execution; Path condition; KLEE.

Concrete analysis pattern

Foundations. A general diagnostic technique realized for a specific artifact format, platform, product, or tool. It identifies the commands, fields, conditions, and interpretation needed in that setting. The concrete form supports repeatable investigation, while its general counterpart explains which diagnostic question the procedure answers and what can transfer to another environment.

Ref. R001, pp. 216–218. See also: General analysis pattern; Analysis Pattern Square; ADC.

Concrete problem pattern

Foundations. A recurrent set of diagnostic indicators expressed in a particular platform or product context. Exact function names, exception codes, object types, or trace fields can distinguish this realization from related cases. We can generalize the pattern without discarding the concrete evidence required to recognize it correctly in an actual artifact.

Ref. R001, pp. 216–218. See also: General problem pattern; Problem pattern; Diagnostic indicator.

Conditional breakpoint

Debuggers. A breakpoint whose stopping decision depends on an expression evaluated when the breakpoint is encountered. It can select a particular argument, object state, or iteration from many executions of the same code. We check the expression's scope and evaluation context. Conditions may add overhead even when they do not produce a visible stop, and expressions with side effects can modify the evidence.

Ref. R113, Section 5.1.6. See also: Breakpoint; Expression evaluation; Watchpoint.

Condition variable

Execution. A synchronization mechanism that lets a thread release a lock and wait for notification that a predicate may have changed. After waking and reacquiring the lock, the thread must recheck the predicate. We reconstruct the predicate and notifying code rather than interpreting a wakeup as proof that useful work is available. Missed protocol transitions can leave apparently ordinary waiting stacks.

Ref. R090, Condition-variable behavior and wakeup semantics. See also: Mutex; Deadlock; Semaphore.

Configuration Wrapper

Debuggers. A unified interface for diagnostic settings that are otherwise scattered, internal, or difficult to configure. The wrapper can expose tracing switches, checking options, and related controls in one place. We record the effective configuration and any resulting behavioral change. Enabling a facility through a convenient interface still changes the conditions under which evidence is acquired.

Ref. R852, Pattern discussion. See also: Debugware; Instrumentation; Observer effect.

Confusion matrix

AI and ML. A table comparing predicted classes with reference classes. It reveals which categories are confused instead of hiding them within overall reported accuracy. Interpretation requires the table's orientation and trustworthy labels. Diagnostic disagreements can reflect a model error, an ambiguous category, or an incorrect reference judgment.

Ref. R234, Confusion matrix. See also: Precision; Recall; F1 score.

Constant Subtrace

Memory patterns. Several stack traces share a stable subtrace whose divergence point helps group related execution paths or suggest a wait relationship. We compare the common frames and the branches around them, then verify the proposed dependency using thread or synchronization evidence. The repeated fragment organizes the collection; it does not by itself prove that one thread waits for another.

Ref. R679, Constant Subtrace pattern and diagnostic examples. See also: Stack Trace Collection; Wait Chain; Variable Subtrace.

Consumer lag

Data systems. A measure of how far a message consumer is behind a defined producer or log position. The offset basis matters: fetched, processed, and committed progress are different states. Kafka 3.6's consumer records-lag-max metric uses the current offset, not the committed offset. A low value for one lag measure therefore does not by itself prove that application processing and durable completion have caught up.

Ref. R330, Consumer Fetch Metrics: records-lag-max. See also: Queue length; Queueing delay; Throughput.

Container log

Cloud. Output associated with a container's execution, commonly captured from standard output and standard error by the runtime and node logging system. Kubernetes exposes this output through its logging interfaces, but storage and retention depend on node and cluster configuration. A restarted container can have a different log instance. Investigation should retain container identity, restart history, and the distinction between current and previous execution.

Ref. R296, Pod and container logs; log rotation. See also: Log rotation; CrashLoopBackOff; Termination message.

Container Trace

Trace patterns. A shared output stream containing messages from components that may use different trace schemas, including multiline exception reports or unstructured text. The term also applies beyond software containers. Before applying field-based analysis, we identify which records obey which schema and how their boundaries are represented.

Ref. R002, p. 83. See also: Trace Schema; Text Trace; Split Message.

Contention

Execution. Competition among activities for a shared resource whose access or capacity cannot satisfy them simultaneously. Lock ownership and waiting threads make synchronization contention visible, while other resources require different measurements. We relate waiting, throughput, and ownership to the workload before assigning significance. Some contention follows from necessary coordination; a contended resource becomes a diagnostic lead when it explains the observed loss of progress.

Ref. R409, Synchronization: introductory discussion of thread contention. See also: High Contention; Lock convoy; Critical section; Starvation.

Context Pointer

Memory patterns. A register context is treated as a collection of pointers into memory, providing several starting points for examining related structures. We filter out values that do not meaningfully denote addresses and follow validated references into their surrounding regions. The view extends analysis beyond the instruction and stack pointers. A register's numerical resemblance to an address still requires contextual confirmation.

Ref. R689, Context Pointer pattern and diagnostic examples. See also: Register context; Pointer Cone; Memory examination.

Context propagation

Tracing. Transferring execution context across boundaries so that later work can be associated with earlier work. Propagation commonly injects context into a carrier, such as request headers, and extracts it at the receiving side. A missing or replaced context can fragment one workflow into unrelated traces. Propagation must also follow asynchronous execution boundaries; sharing a process does not automatically preserve the intended context.

Ref. R267, Context propagation. See also: Span context; traceparent; Baggage.

Context switch

Execution. The replacement of one executing thread context with another by the scheduler. It involves preserving and restoring execution state and may follow blocking, preemption, or completion of a time slice. We interpret a high switch rate in relation to useful work and scheduling events. It can indicate fine-grained coordination or contention, but the count alone does not identify wasted execution.

Ref. R016, Ready threads and context-switch causes. See also: Thread; CPU register; Priority inversion.

Contract

Program analysis. A specification of obligations between a caller and a called component, commonly expressed through preconditions, postconditions, and invariants. It makes an expected interaction explicit enough to check or reason about. A contract violation helps locate the boundary where an obligation was not met. The contract can itself be incomplete or incorrect, so its intended meaning remains part of the analysis.

Ref. R142, Design by Contract; client and supplier obligations. See also: Precondition; Postcondition; Invariant.

Control-flow graph

Program analysis. A graph of program locations or basic blocks connected by possible transfers of control. Branches, loops, and exceptional transfers determine its shape. It supplies the structural model for many coverage, slicing, and data-flow analyses. A path through the graph may be infeasible because its conditions conflict; graph connectivity alone does not establish that a concrete execution can take that path.

Ref. R144, Control-flow graph and data-flow examples. See also: Data-flow analysis; Path condition; Branch coverage.

Coordinated omission

Performance. A measurement bias in which the measurement process stops or reduces issuing observations while the system is slow, omitting delays that an independent arrival stream would have experienced. The resulting latency distribution can look better during a stall. Corrections based on an expected interval add modeled observations; they are not recovered measurements. The workload's arrival model must therefore accompany the reported result.

Ref. R304, copyCorrectedForCoordinatedOmission; recordValueWithExpectedInterval. See also: Latency; Histogram; Throughput.

Copy-on-write

Memory. A sharing mechanism in which participants initially reference common storage and obtain a private copy when a write requires separation. It can reduce copying and memory consumption before modification occurs. We use mapping and sharing information to explain why physical-memory use changes after writes. Two mappings that began with identical contents do not necessarily continue to expose the same bytes.

Ref. R012, Copy-on-Write Protection. See also: Memory-mapped file; Working set [Memory]; Memory protection.

Cord of Activity

Trace patterns. A combination of activity strands formed using different kinds of adjoint identifiers. The resulting view joins perspectives such as process, thread, module, and referenced data. It gives an organized way to examine relationships across these projections without assuming that one identifier captures the whole activity.

Ref. R002, pp. 84–85. See also: Strand of Activity; Adjoint Thread of Activity; Braid of Activity.

Core dump

Artifacts. A saved process-state artifact commonly used on Unix-like systems for postmortem debugging. On Linux, generation and contents depend on signal disposition, limits, permissions, configuration, and dump filters. We check these conditions before expecting a core to contain a particular mapping. A missing or truncated core can result from collection policy rather than from an absence of a process failure.

Ref. R054, DESCRIPTION; Naming of core dump files; coredump_filter. See also: Crash dump; ELF; coredumpctl.

coredumpctl

Native tools. A systemd tool for listing and retrieving recorded process core dumps and their associated information, and for invoking supported debugging workflows. It connects collection records with the saved artifact. We distinguish a journal entry describing a crash from the continued availability of its core file. Retention policy, permissions, and storage settings can leave metadata available after the memory evidence has been removed.

Ref. R122, Description; Commands; COREFILE status. See also: Core dump; GDB; Crash dump.

Correlated Discontinuity

Trace patterns. Gaps in different activities or traces that correspond in a diagnostically relevant way. The correspondence may suggest a shared dependency, communication problem, or common collection interruption. We compare the intervals and surrounding events, then seek evidence that distinguishes these possible mechanisms.

Ref. R002, p. 86. See also: Discontinuity; Inter-Correlation; Coupled Activities.

Correlation identifier

Logs. A value carried through related operations and recorded so that their observations can be grouped. It may identify a request, business interaction, or workflow and need not be a tracing-system trace identifier. Its scope and lifetime determine what a match means. Reuse across attempts can be useful, but attempts must remain distinguishable when investigating retries, duplicates, or partial completion.

Ref. R332, Event attributes: Interaction identifier. See also: Log correlation; Trace identifier; Retry.

Corrupt Dump

Memory patterns. A dump artifact whose own structure or contents are damaged, preventing reliable interpretation of some or all of the recorded state. Invalid headers, unusable metadata, and damaged ranges require validation before ordinary analysis proceeds. We preserve and examine any independently recoverable information. Corruption of the dump file is distinct from correctly captured corruption in the target, although both conditions can occur together.

Ref. R377, Invalid-header and partially usable dump examples. See also: Truncated Dump; Lateral Damage; Inconsistent Dump.

Corrupt Message

Trace patterns. A message whose recorded or displayed content has been damaged or incorrectly transformed. Causes include faulty formatting, buffer handling, truncation, and presentation-time type conversion. We compare the raw artifact with another representation or message family before relying on searches and extracted values.

Ref. R002, p. 87. See also: Truncated Data; Message Invariant; Data Flow.

Corrupt Structure

Memory patterns. A structure contains values or relationships inconsistent with its expected layout, such as broken list links, implausible pointers, or text bytes occupying pointer fields. We first validate the address, type, symbols, and memory availability used to interpret it. Once those checks hold, the malformed structure becomes evidence for corruption and a starting point for reconstructing its origin.

Ref. R550, Pattern description: Corrupt Structure. See also: Critical Section Corruption; Wild Pointer; Paged Out Data.

CoTrace

Trace patterns. The trace of an analysis itself: the messages, regions, and relationships visited while investigating another trace. Its order follows the investigation and may differ from execution order. Recording this path makes reasoning reviewable and allows analysis patterns to be applied to the analysis activity.

Ref. R002, pp. 88–90. See also: Meta Trace; Trace Path; Message Annotations.

Counter

Metrics. A metric that accumulates nonnegative increments, normally increasing until it resets or is replaced. It describes how much activity has occurred, such as completed requests or reported failures. Its absolute value depends on the counting lifetime; rate and increase calculations describe activity over an interval. A decrease usually requires considering reset or identity changes before interpreting it as a reduction in activity.

Ref. R270, Counter. See also: Rate; Gauge; Cumulative temporality.

Counter Value

Trace patterns. A recorded performance or state measurement treated as a trace message, so that activity and correlation patterns can be applied to it. This pattern uses counter in the broad performance-monitoring sense; the recorded quantity may be a level, rate, or derived value rather than a strictly increasing count.

Ref. R002, p. 91. See also: Measurement; Activity Region; Time Delta.

Counting Objects

Debuggers. Measuring the number of objects in a defined population as an elementary diagnostic action. Counts can support comparisons across types, regions, or acquisition times. We specify what qualifies as an object and which collection was counted, then relate any difference to workload and lifetime. A large count alone does not establish a leak.

Ref. R001, pp. 268–270. See also: Object Distribution Anomaly; Memory leak; Snapshot Collection.

Coupled Activities

Trace patterns. Activities related through a service, request, or other dependency, so that a problem in one can become visible in another. We follow the relationship from a client symptom toward the corresponding server or supporting activity. The coupling guides artifact selection and the search for propagation mechanisms.

Ref. R002, p. 92. See also: Correlated Discontinuity; Inter-Correlation; Causal History.

Coupled Machines

Memory patterns. Data or execution in one computer reveals a dependency on another computer, such as a path naming a remote share. We follow the operation and its participating processes to establish the boundary across which progress depends. The pattern extends investigation beyond visible local IPC relationships and identifies where complementary artifacts may be needed.

Ref. R508, Pattern description: Coupled Machines. See also: Coupled Processes; Distributed Wait Chain; Wait Chain (RPC).

Coupled Modules

Memory patterns. Modules depend on one another's services, so a resource or failure attributed to one module may involve requests from another. We investigate those functional relationships when a pool tag, stack, or other indicator singles out a component. The pattern extends attribution beyond the immediate owner while preserving the distinction between a dependency and evidence of a defect.

Ref. R768, Crash Dump Analysis Patterns (Part 140). See also: Directing Module; Coupled Processes; Insufficient Memory (kernel pool).

Coupled Processes

Memory patterns. Processes connected by dependencies that make the observed state of one process depend on activity in another. Requests, replies, shared resources, and lifecycle relationships can carry the problem beyond the original dump boundary. We identify the participating processes and collect corresponding evidence with suitable timing. A waiting client may expose the symptom while the obstructed operation resides in a different process.

Ref. R362, Opening process-dependency discussion and related-process examples. See also: Process; Wait Chain; Stack Trace Collection.

Coupled Processes (semantics)

Memory patterns. Processes cooperate to provide a function even when their relationship is not directly visible as an IPC chain. We extend the investigation to this semantic group using architecture and workload knowledge, then inspect each member's artifacts for relevant anomalies. A healthy-looking endpoint in the visible chain does not exclude a failure in another cooperating process.

Ref. R477, Pattern description: Coupled Processes (semantics). See also: Coupled Processes; Coupled Processes (weak); Systemic diagnostics.

Coupled Processes (weak)

Memory patterns. Processes interact through a comparatively loose relationship, such as subscription and notification, without necessarily waiting for each individual response. Activity in a notifier may explain correlated activity in consumers. We still examine the delivery mechanism: synchronous notification to an unresponsive recipient can introduce a blocking dependency and turn an apparently loose relationship into a stronger coupling.

Ref. R722, Crash Dump Analysis Patterns (Part 28b). See also: Coupled Processes; Wait Chain; Spiking Thread.

Coverage-guided fuzzing

Testing. Fuzzing that uses execution feedback to retain inputs reaching new or otherwise interesting program behavior. Mutations then explore around this evolving corpus. Coverage guides the search toward structural novelty, while a separate oracle determines whether behavior is faulty. A campaign can increase measured coverage without reaching the values, state sequences, or environmental conditions needed to expose a particular defect.

Ref. R162, Coverage feedback and corpus evolution. See also: Fuzzing; libFuzzer; Code coverage.

cProfile

Runtimes. Python's familiar interface for deterministic profiling of calls and associated execution-time statistics. It helps identify frequent calls and distinguish a function's own measured time from time accumulated through callees. Profiling changes execution cost, and measured time is not automatically an explanation of waiting or end-user latency. Module organization and implementation details are version-sensitive, so a report should retain its Python version.

Ref. R221, cProfile compatibility note; What is deterministic profiling?. See also: Callgrind; Allocation profiling; Event-loop blocking.

CPU profile

Performance. A distribution of recorded CPU activity attributed to instructions, functions, or call paths. A common form aggregates stack samples collected while threads execute on a processor. It helps identify where CPU time is spent, but does not explain time spent waiting off CPU. The profiled threads, interval, symbols, and sample population define the scope of the resulting evidence.

Ref. R392, Introduction; Description. See also: CPU time; Off-CPU profile; Flame graph.

CPU register

Execution. A processor storage location used for computation, addressing, control, or execution state. Its interpretation depends on architecture, instruction position, and calling convention. We examine registers with the corresponding code and saved context. A value that once held a function argument may already have been overwritten, so a register displayed at failure time is not automatically an entry-time argument.

Ref. R028, Section 10.14; R078, Calling convention defaults; Parameter passing; Unwindability. See also: Program counter; Calling convention; Register context.

CPU time

Performance. Processor time charged to a thread or process, commonly separated into user-mode and kernel-mode execution. It excludes intervals when the measured execution is not running on a processor. Process CPU time can sum activity across several threads and therefore exceed elapsed time when they run concurrently. Specify the accounting scope before comparing CPU time with a request's duration.

Ref. R397, Real time and process time; R398, CLOCK_PROCESS_CPUTIME_ID; CLOCK_THREAD_CPUTIME_ID. See also: Elapsed time; CPU utilization; CPU profile.

CPU utilization

Performance. A measure of processor use over an interval, commonly derived from time accounted as non-idle. The denominator may represent one logical CPU or a larger set, so percentage conventions must be stated. High utilization does not establish efficient progress: memory stalls and spinning can consume accounted CPU time. Low averages can also conceal short bursts or uneven activity across processors.

Ref. R399, What really is CPU Utilization?; Other reasons CPU utilization is misleading. See also: CPU time; Saturation; Cache miss.

Crash

Execution. A functional failure in which a component or system ceases to be observed in its previous running form, such as a disappearing process, bug check, or reboot. We distinguish that outcome from the exception or corruption that may precede it. A handled fault need not cause a crash, and a crash report alone does not identify the initiating mechanism.

Ref. R001, p. 38, Crashes and Hangs Differentiated; R423, Definition. See also: Hang; Crash dump; Exception.

Crash dump

Artifacts. A diagnostic artifact collected in connection with a failure that stops normal execution. It may preserve memory, registers, thread state, modules, and exception or stop information, depending on the format. We begin by establishing what was captured and when. The presence of a crash dump provides an observation of failure state; reconstructing the preceding mechanism requires further interpretation.

Ref. R054, DESCRIPTION; Naming of core dump files; coredump_filter; R065, Dump contents. See also: Memory dump; Exception record; Bug check.

Crashed Process

Memory patterns. A system dump contains evidence that a process previously crashed or terminated unexpectedly, even though its active execution is gone. Retained process objects, reporting processes, and a later replacement instance can help identify the candidate. We correlate these signs with the application's expected lifecycle, because repeated terminated instances can also result from intentional shutdown and restart.

Ref. R797, Crash Dump Analysis Patterns (Part 223). See also: Zombie Processes; Error Reporting Fault; Last Object.

CrashLoopBackOff

Cloud. A Kubernetes-reported condition indicating repeated container restarts with a delay before another attempt. It is not a Pod phase and does not identify the original failure mechanism. An application exit, configuration error, failed liveness probe, or resource problem can produce the cycle. Examine the preceding termination reason, previous container logs, events, and startup configuration rather than diagnosing from the displayed status alone.

Ref. R292, Container restarts; CrashLoopBackOff. See also: Container log; Liveness probe; OOMKilled.

Crash Signature

Memory patterns. A selected combination of failure attributes used to recognize and compare crashes, such as the faulting instruction, stack, frame information, and module context. We choose attributes that are meaningful and sufficiently stable for the intended grouping. Similar signatures suggest related cases, while damaged context or overly broad attributes can merge different causes and unstable values can split one cause.

Ref. R757, Crash Dump Analysis Patterns (Part 114). See also: Stack Trace Signature; Fault Context; Exception Module.

Crash Signature Invariant

Memory patterns. A stable part of a crash signature survives changes in platform or representation. Equivalent failing operations and common call structure can relate crashes whose registers, addresses, or offsets differ. We identify the invariant explicitly and check the underlying operation and operand. Signature similarity is useful for grouping cases, but does not by itself establish a shared defect mechanism.

Ref. R667, Crash Signature Invariant pattern and diagnostic examples. See also: Crash Signature; Crash; Invariant.

crash utility

Native tools. A Linux kernel-analysis utility that combines kernel-aware commands with GDB facilities for supported live systems and dump formats. It can inspect tasks, stacks, structures, and memory using suitable kernel debug information. We establish the relationship between the kernel image and memory artifact before interpreting output. Supported formats and target changes matter; a readable dump file is not sufficient without a compatible interpretation of kernel structures.

Ref. R123, Abstract; Prerequisites; The Command Set. See also: vmcore; kdump; Symbol file.

C-Reduce

Testing. A test-case reducer that applies transformations to C and C++ programs while an external interestingness test decides which candidates retain the desired behavior. It is useful for simplifying compiler failures and other source-based reproducers. The interestingness test defines the diagnostic contract: a candidate that triggers a different crash or relies on newly introduced undefined behavior may be small but irrelevant.

Ref. R183, README; interestingness tests and reduction. See also: Test-case reduction; Delta debugging; Minimal reproducer.

Critical path

Performance. A longest-duration chain of dependent activities in a model of an execution, determining completion time under that model's dependency assumptions. It identifies work whose delay constrains the chosen outcome. The visually longest span is not necessarily the critical path, and overlapping parent and child durations cannot simply be added. Missing dependencies, incomplete instrumentation, and clock inconsistencies limit a path reconstructed from traces.

Ref. R343, Sections 4–5, pp. 658–660. See also: Distributed tracing; Response time; Causal order.

Critical Point

Trace patterns. A point or region at which the shape of one trace changes, for example in its activity distribution or organization. The topological terminology provides an analogy for locating change. Unlike a Bifurcation Point between traces, this pattern concerns a change within the trace under examination.

Ref. R002, p. 93. See also: Bifurcation Point; Intra-Correlation; Trace Shape.

Critical Region

Memory patterns. A code region lies between contending operations and may access shared data relevant to the incident. We compare participating stacks and inspect the intervening code and object references to identify that region. The analysis pattern does not require a conventional lock: its purpose is to locate the shared activity whose ordering or simultaneous execution needs further investigation.

Ref. R801, Crash Dump Analysis Patterns (Part 235, Linux). See also: High Contention; Stack Trace Race; Shared Structure.

Critical section

Execution. A region of execution whose shared-state operations require controlled concurrent access. On Windows, CRITICAL_SECTION also names a specific process-local mutual-exclusion object. We distinguish the code region from that implementation when examining a report. Ownership, recursion, and acquisition rules matter; a waiting stack identifies an attempted entry, but the current owner's work explains why entry remains unavailable.

Ref. R098, Process-local mutual exclusion and acquisition order. See also: Mutex; Deadlock; Spinlock.

Critical Section Corruption

Memory patterns. Damage to a Windows critical section or its associated bookkeeping that makes the synchronization state internally implausible. Broken links, impossible ownership relationships, or pointers into unrelated data can explain failures in lock inspection or use. We check the structure layout, initialization, and lifetime before diagnosing corruption. An unfamiliar counter value alone is insufficient because valid representations vary with implementation and state.

Ref. R387, Critical-section structure examples and comment on interpreting LockCount. See also: Critical section; Memory corruption; High Contention.

Critical Stack Trace

Memory patterns. A stack shows prolonged involvement in an operation whose timely completion is important to system progress, such as page-fault processing. We examine the unexpected wait and compare similar stacks to find the shared blocking component or dependency. The significance comes from the activity and its delay together; the presence of a normally brief system routine alone is insufficient.

Ref. R785, Crash Dump Analysis Patterns (Part 188). See also: Blocking Module; Wait Chain; Stack Trace Collection.

Cross-validation

AI and ML. Evaluation that repeatedly trains and assesses a model on specified partitions of data. It helps estimate performance variability and compare choices without relying on one arbitrary split. The partition scheme must match the intended use: records from the same incident, machine, or time period may require grouping or ordering. Otherwise, information shared between partitions can make generalization appear stronger than it is.

Ref. R237, Cross-validation; grouped and time-series data. See also: Data leakage; Overfitting; Precision.

Cumulative temporality

Metrics. An aggregation convention in which each reported metric point covers the interval from an identified start time to the point's end time. Consecutive points therefore overlap rather than describe independent increments. Summing them counts some activity repeatedly. Diagnostic interpretation must account for resets, changed start times, and missing points before comparing values or converting them into an interval rate.

Ref. R261, Temporality. See also: Delta temporality; Counter; Rate.

Custom Exception Handler
Memory patterns. Application or component exception handling that changes how an observed fault is reported, propagated, or recovered from. Such handling can explain why an earlier exception is absent from the final selected context or why execution continues into a later problem. We reconstruct the relevant handler path using available context and platform mechanisms. A custom handler can implement legitimate recovery as well as obscure evidence.

Ref. R373, Opening handler discussion and exception-handling examples. See also: Hidden Exception; Exception; Early Crash Dump.

Custom Exception Handler (kernel space)
Memory patterns. Kernel code handles an earlier exception and deliberately reports a failure through its own bugcheck path. The visible stack may therefore emphasize the handler rather than the original invalid operation. We recover the saved exception and context where available, relate them to the custom reporting code, and distinguish the original fault from the mechanism chosen to stop the system.

Ref. R726, Crash Dump Analysis Patterns (Part 39b). See also: Custom Exception Handler; Hidden Exception (kernel space); Self-Diagnosis (kernel mode).

D

Daikon
Program analysis. A tool for discovering likely invariants from values recorded during program executions. Its output can describe recurring relationships among variables at selected program points. These relationships help formulate contracts and compare behavior before and after a change. They remain observations generalized from the supplied executions; additional tests or reasoning are needed before adopting them as guarantees of correct behavior.

Ref. R145, Daikon tool overview and applications. See also: Dynamic invariant detection; Invariant; Contract.

Dangling pointer
Memory. A pointer that no longer refers to a live object of the intended kind. Releasing an allocation or ending the lifetime of a local object can leave such a pointer in other state. The pointer value itself may look unchanged and plausible. We investigate the lifetime relationship between the reference and its former target; dereferencing it can produce a use-after-free or another invalid access.

Ref. R046, Alternate Terms (Dangling pointer); Description. See also: Use-after-free; Null pointer dereference; Stack.

Dapper

Tracing. Google's distributed tracing infrastructure described in a 2010 technical report. Its design combined shared-library instrumentation, propagated trace information, and sampling to make cross-service behavior visible at large scale. Dapper is an influential concrete implementation of distributed tracing, not a generic name for all tracing. Its assumptions about overhead, coverage, and sampling must be considered when applying its lessons elsewhere.

Ref. R279, Abstract; Sections 1–2. See also: Distributed tracing; Instrumentation; Sampling.

Dashboard

Reliability. An organized view of selected telemetry and derived results for a particular operational purpose. A dashboard can expose a change quickly, but every chart reflects a query, aggregation, interval, and chosen population. Its apparent completeness is a presentation property. During diagnosis, inspect the underlying definitions and follow unusual aggregates into more detailed artifacts rather than treating a normal overview as proof that no problem exists.

Ref. R286, Dashboards. See also: Monitoring; PromQL; Aggregation.

Data Alignment (page boundary)

Memory patterns. A memory operand spans a page boundary, so validating its starting address does not validate the complete access. We inspect the instruction's operand width and every page it touches, including their accessibility in the current execution context. The pattern explains why an apparently valid pointer can participate in a fault when a later byte lies on another page.

Ref. R498, Pattern description: Data Alignment (page boundary). See also: Invalid Pointer (general); Page fault; False Effective Address.

Data Association

Trace patterns. An association between changing property-value pairs or larger tuples, possibly recorded in different messages and sources. We follow how the relationship changes across the trace. It generalizes comparison within a single message family and differs from merely following one unchanged value through successive records.

Ref. R002, p. 94. See also: Data Flow; Message Change; Gossip.

Database blocking

Data systems. A condition in which a database operation cannot proceed because another transaction or session holds a required incompatible lock. Lock tables and session activity help reconstruct the dependency. A long-running blocker can be idle inside a transaction, so the currently displayed statement may not explain when its locks were acquired. Ordinary blocking differs from a deadlock, which requires a cycle of dependencies.

Ref. R300, Viewing Locks. See also: Wait event; Transaction isolation; Slow query log.

Data Contents Locality

Memory patterns. Distinctive memory contents resemble data observed in earlier incidents, providing a way to narrow an investigation across otherwise different dumps. We compare meaningful strings or structures together with their location and execution context. The resemblance becomes more useful when code-path evidence also agrees, but shared contents alone do not establish that the same defect produced both cases.

Ref. R739, Crash Dump Analysis Patterns (Part 66). See also: Code Path Locality; Regular Data; Historical Information.

Data Correlation (CPU times)

Memory patterns. Relationships between thread CPU times, elapsed times, or creation times that suggest related activity. Similar ratios or paired ages can identify cooperating threads even when no explicit synchronization dependency is visible. We use the numerical correspondence to guide stack and process comparisons; correlation supplies a line of inquiry rather than establishing which thread caused another's behavior.

Ref. R520, Pattern description: Data Correlation (CPU times). See also: Spiking Thread; Coupled Processes; Thread Age.

Data Correlation (function parameters)

Memory patterns. Values in different stack frames support one another through a known semantic relationship, such as element count multiplied by element size matching an allocation request. We use that correlation to assess apparently uncertain function arguments in optimized code. The relationship supplies corroborating evidence about the operation; it should be checked against the function prototypes and relevant data types.

Ref. R765, Crash Dump Analysis Patterns (Part 134a). See also: False Function Parameters; Optimized Code (function parameter reuse); Frame Regularity.

Data drift

AI and ML. A change in the distribution or characteristics of incoming data relative to a reference population. New versions, workloads, collection settings, or user behavior can alter diagnostic features even when the target relationship remains stable. A detected difference identifies a changed observation population. It does not establish model failure or explain whether the underlying system became faulty.

Ref. R238, Checking data drift. See also: Concept drift; Training-serving skew; Feature extraction.

Data Flow

Trace patterns. A view obtained by following a data value as it passes between messages, components, threads, or processes. Unlike filtering for one message template, this view may connect different message types. Identity, representation changes, and value reuse require attention before the observations can be interpreted as one flow.

Ref. R002, p. 95. See also: Data Association; Linked Messages; Adjoint Thread of Activity.

Data-flow analysis

Program analysis. Analysis that propagates facts about values or program state through control flow. Examples include reaching definitions and possible value ranges. At a merge point, facts from different predecessors must be combined, often losing distinctions between paths. The result explains what the analysis permits at a location; interpretation must account for its transfer rules, merge operation, and treatment of calls.

Ref. R144, Data-flow facts, joins, and fixed-point computation. See also: Control-flow graph; Abstract interpretation; Taint analysis.

DataFlowSanitizer

Program analysis. A framework for dynamic data-flow analysis that associates labels with data and propagates them through instrumented operations. A client defines what the labels mean and which observations matter. It can therefore support taint-style experiments on a concrete execution. Propagation rules and library wrappers form part of the model; the framework alone does not identify which observed flows constitute defects.

Ref. R182, Introduction; labels and client interface. See also: Taint analysis; Data-flow analysis; Dynamic analysis.

Data Interval

Trace patterns. A trace interval selected using occurrences of data named in the problem description, such as a user or device identifier. The first and last relevant occurrences can define closed boundaries, or the analysis can use open boundaries. The selected interval remains a starting view, not a guarantee that all relevant events lie inside it.

Ref. R002, p. 96. See also: Basic Facts; Trace Window; Trace Mask.

Data leakage

AI and ML. Use of information during model development or evaluation that would not legitimately be available for the intended prediction. It can arise through preprocessing fitted on all data or related examples appearing on both sides of a split. Diagnostic datasets are especially vulnerable to duplicate incidents and post-resolution labels. Inflated evaluation results then describe access to leaked information rather than deployable predictive ability.

Ref. R235, Data leakage; preprocessing and train/test separation. See also: Cross-validation; Overfitting; Training-serving skew.

Data race

Execution. Conflicting memory accesses to the same location that lack the ordering required by the language memory model. In Go, read-write and write-write pairs race when unordered by happens-before and at least one access is non-synchronizing. We separate this precise condition from broader timing-dependent logic errors. Atomic accesses can avoid data races without making an entire operation logically correct; the relevant synchronization guarantees still require examination.

Ref. R405, Memory Model: formal definitions of read-write and write-write data races. See also: Race condition; Happens-before; Atomic operation; ThreadSanitizer.

Data Reversal

Trace patterns. An unexpected reversal in the order of values carried by the same message type, or in related variable fields. Nested enter and leave records can produce an expected reversal, so the interpretation depends on the operation. The pattern concerns value order rather than the order of different event types.

Ref. R002, pp. 97–98. See also: Message Invariant; Event Sequence Order; Opposition Messages.

Data Selector

Trace patterns. Data obtained from one trace and used to select relevant messages in another. A client identifier may locate the corresponding session among many server sessions. We first establish the identifier's meaning and lifetime, then construct the selected activity and inspect its surrounding messages.

Ref. R002, p. 99. See also: Inter-Correlation; Linked Messages; Identification Messages.

Dead-letter queue

Data systems. A separate destination for messages that could not be processed under the configured retry or delivery policy. It preserves work for investigation or later handling and makes a terminal policy decision observable. An empty dead-letter queue does not establish that processing is healthy: messages may still be waiting or retrying. Inspect original context, attempt history, message age, and the rule that redirected the message.

Ref. R319, Measuring availability and latency: dead-letter queues. See also: Consumer lag; Retry; Queueing delay.

Deadline

Distributed systems. A time by which an operation must complete to remain useful under its contract. A deadline can constrain a chain of calls through the remaining time budget rather than giving each dependency a fresh full timeout. In gRPC, propagation accounts for elapsed time. Client-side expiry does not necessarily stop separately spawned work; cancellation handling and downstream completion require their own evidence.

Ref. R322, Deadlines; deadline propagation; server cancellation. See also: Timeout [Distributed systems]; Retry; Critical path.

Deadlock

Execution. A state in which participants cannot make progress because their required dependencies cannot be satisfied by the blocked participants. In lock-based cases, a cycle of ownership and waiting is important evidence. We reconstruct the dependency structure rather than declaring deadlock from a collection of waiting stacks. Timeouts, cancellation, and external completion can change whether an apparent wait cycle is permanent.

Ref. R053, Description; Common Consequences. See also: Mutex; Livelock [Execution]; Condition variable.

Deadlock (critical sections)

Memory patterns. A circular wait involves Windows critical sections whose ownership connects the waiting threads. We combine lock records with thread stacks and follow each owned resource to its next waiter until the dependency closes. A collection of locked sections alone is insufficient: the ownership and waiting relationships establish the deadlock in the captured process context.

Ref. R460, Pattern description: Deadlock (critical sections). See also: Wait Chain (critical sections); Deadlock; Critical Section Corruption.

Deadlock (executive resources)

Memory patterns. A cyclic wait involving Windows executive resources and their owning threads. Resource listings expose exclusive or shared ownership and waiting threads, allowing us to follow dependencies until an owner waits back on an earlier participant. We then inspect the participating stacks and components. A large number of blocked threads alone does not establish the cycle that defines this specialization.

Ref. R584, Deadlock (executive resources) pattern and diagnostic examples. See also: Deadlock; Wait Chain; Wait Chain (executive resources).

Deadlock (LPC)

Memory patterns. A circular wait crosses process boundaries through LPC request processing and other synchronization objects. We follow message identifiers to server threads, then continue through the locks or mutexes those threads await. Closing the chain back to an earlier participant explains why separate process stacks can each look like an ordinary wait while the combined system cannot progress.

Ref. R461, Pattern description: Deadlock (LPC). See also: Wait Chain (LPC/ALPC); Coupled Processes; Distributed Wait Chain.

Deadlock (managed space)

Memory patterns. Managed threads participate in a circular synchronization dependency. In a CLR dump, synchronization-block ownership, managed stack traces, and relevant code identify the participants; an extension can assist with detection. We still inspect the underlying relationships, because the diagnostic result depends on the captured object state and on which synchronization mechanisms the tool understands.

Ref. R462, Pattern description: Deadlock (managed space). See also: Wait Chain (CLR monitors); Managed Stack Trace; Deadlock (.NET Finalizer).

Deadlock (mixed objects, kernel space)

Memory patterns. A cyclic wait that passes through different kinds of kernel synchronization objects. An executive resource, mutex, or thread-termination wait can connect participants that one object-specific listing does not reveal. We reconstruct each ownership and waiting edge and follow it across process boundaries. The process containing a blocked thread does not necessarily identify the component that introduced the dependency.

Ref. R585, Deadlock (mixed objects, kernel space) pattern and diagnostic examples. See also: Deadlock; Wait Chain; Mutex.

Deadlock (mixed objects, user space)

Memory patterns. A circular dependency combines different synchronization mechanisms, for example a critical section and an event. We reconstruct both lock ownership and the actions needed to signal or release the other object. Because events do not directly identify an owner, stack residue and code may be needed to establish the missing dependency before calling the observed waits a deadlock.

Ref. R706, Crash Dump Analysis Patterns (Part 9c). See also: Wait Chain; Deadlock (self); Execution Residue.

Deadlock (.NET Finalizer)

Memory patterns. A cyclic dependency involving the .NET finalizer thread. A thread waiting for pending finalizers may itself service work required by the finalizer, such as a COM or RPC call. We combine managed and native stacks and identify the destination thread to establish both directions of the wait. The finalizer's involvement alone does not establish that finalization is defective.

Ref. R586, Deadlock (.NET Finalizer) pattern and diagnostic examples. See also: Finalization; Deadlock; Managed Stack Trace.

Deadlock (self)

Memory patterns. A thread waits for progress that depends on a resource it already holds. The published kernel example involves shared ownership of an executive resource followed by an exclusive acquisition attempt. We check the object's acquisition and upgrade rules, together with thread identity and ownership, because repeated acquisition alone does not imply deadlock for synchronization primitives that explicitly support recursion.

Ref. R707, Crash Dump Analysis Patterns (Part 9f). See also: Wait Chain (executive resources); Deadlock (mixed objects, user space); Blocked Thread.

De Broglie Trace Duality

Trace patterns. A metaphorical relationship between temporal trace activity and spatial memory evidence, such as recurring errors accompanying heap growth. The pattern encourages joint analysis of logs and dumps when either artifact alone is insufficient. The analogy does not assert a physical wave-particle law or prove that the two observations share a cause.

Ref. R002, pp. 100–101. See also: Adjoint Space; Periodic Error; Counter Value.

Debug Event Subscription / Notification

Debuggers. An architectural debugging pattern in which a diagnostic consumer arranges to receive selected execution events. It connects an analysis question with facilities that report when relevant activity occurs. The unified debugging examples realize this through more specific designs and breakpoint usage. We retain the event selection and context when interpreting what the notification establishes.

Ref. R878, Pattern discussion. See also: Unified Debugging Pattern Language; Breakpoint; Punctuated Execution.

Debugger

Debuggers. A tool that inspects program state and, for a live target, can control execution. Its facilities may include breakpoints, memory examination, expression evaluation, and stack reconstruction. We distinguish these observations from the interpretation supplied by symbols and extensions. A debugger cannot recover every omitted or overwritten value, and attaching or stopping a live target can change the timing of the behavior under investigation.

Ref. R112, Summary; R119, Engine capabilities and extension libraries. See also: Breakpoint; Memory examination; Debugger context.

Debugger Bug

Memory patterns. The debugger or its extension introduces a failure or misleading result during capture, live debugging, or artifact analysis. A hard failure may crash or hang the tool; a softer defect may produce plausible but incorrect output. We compare independent views and relevant tool versions before attributing the anomaly to the target. Diagnostic output also requires validation.

Ref. R653, Debugger Bug pattern and diagnostic examples. See also: Debugger; Debugger extension; Inconsistent Dump.

Debugger context

Debuggers. The selected process, thread, register state, and local frame used to interpret debugger commands. These selections determine which address space and execution state an expression refers to. We make the context explicit when comparing observations. A valid address or variable name in one context may refer to different data in another, and selecting a thread does not always select an exception's saved register context.

Ref. R087, Process Context; Register Context; Local Context. See also: Register context; Address space; Stack frame.

Debugger extension

Debuggers. Additional debugger functionality that interprets target state through commands or APIs. An extension can decode operating-system structures, summarize objects, or automate repeated analysis. We treat its output as a derived representation whose correctness depends on supported target layouts, symbols, and available memory. A failed or partial extension command should be investigated at those dependencies before being interpreted as evidence of target corruption.

Ref. R119, Engine capabilities and extension libraries. See also: WinDbg; !analyze; Symbol file.

Debugger Omission

Memory patterns. A debugger command fails to report evidence that is present in the artifact. We compare its result with an independent inspection method before treating absence from the output as absence from memory. The original examples include image-signature scanning and exception selection; these illustrate a diagnostic failure mode, not a claim that every current debugger retains the same defects.

Ref. R541, Pattern description: Debugger Omission. See also: Multiple Exceptions (user mode); Quiet Dump; No Component Symbols.

Debugging

Foundations. Investigation of software execution and implementation to locate, explain, and correct defects. It may use live control, recorded execution, postmortem artifacts, source analysis, or deliberately constructed models. Diagnostic findings help choose where to investigate; a proposed change still needs verification against the failure mechanism and the behavior the software is expected to provide.

Ref. R001, p. 31; pp. 197–199; pp. 239–241. See also: Software diagnostics; Troubleshooting; Root cause analysis.

Debugware

Foundations. Software, architecture, and supporting facilities designed to make troubleshooting and debugging possible or more effective. Debugware patterns capture recurring design problems and solutions in these facilities. Treating them explicitly helps connect the construction of diagnostic tools with the investigations they must support, including how evidence will be acquired, presented, and interpreted.

Ref. R001, p. 45; p. 129. See also: Software diagnostics engineering; Analysis pattern; Diagnostics-driven development.

Declarative Trace

Trace patterns. The logging statements present in source code, considered as a potential trace. Some statements may never execute, while loops and repeated calls can emit others many times. Comparing this potential with recorded messages helps investigate instrumentation coverage, disabled paths, and changes between software versions.

Ref. R002, p. 102. See also: Generative Trace; Sparse Trace; Missing Message.

Defamiliarizing Effect

Trace patterns. The appearance of unfamiliar messages within a trace landscape whose blocks and activities are otherwise recognizable. The change attracts attention to an unexpected execution path or environment. Familiarity is an analyst-dependent guide, so the unusual messages still need interpretation in the current system and version.

Ref. R002, pp. 103–104. See also: Characteristic Message Block; Bifurcation Point; Significant Event.

Defect

Foundations. A discrepancy between a software system and the model or expectations against which it is evaluated. The discrepancy can concern implementation, behavior, requirements, or the model itself. We therefore state the expected behavior and relevant conditions explicitly; an observation becomes a defensible defect report when its relationship to those expectations is clear.

Ref. R001, pp. 33–34; p. 72. See also: Fault; Failure; Problem description.

Defect Group

Trace patterns. A group of messages associated with a known defect, a problem description, or a problematic configuration, even when no simple selection predicate is available. The group may come from previous analyses. Its members provide a recognition aid; their association with failure does not give every member a causal role.

Ref. R002, p. 105. See also: Causal Messages; Message Set; Small DA+TA.

Defect mechanism

Foundations. The process by which a defect produces particular structural or behavioral indicators. A leak, omitted termination condition, or incorrect synchronization relationship can explain a diagnostic pattern, but the pattern alone does not select the mechanism. We seek implementation and execution evidence that connects the proposed process to the observed result.

Ref. R001, pp. 239–241; pp. 369–370. See also: APM; DMP; Root cause analysis.

Definition Trace

Trace patterns. A trace-like representation of a trace schema or its defining elements. An ordering, such as field order or alphabetical order, can be imposed so that analysis patterns can be applied. This order describes the representation of the schema and must not be read as an execution history.

Ref. R002, p. 106. See also: Trace Schema; Declarative Trace; Message Set.

Delay Dynamics

Trace patterns. Activity elsewhere in the system during a delay in the selected thread or adjoint thread. We inspect the interval across other activities to distinguish possible waiting, competing work, and an uninstrumented path. The visible gap establishes the question; it does not by itself identify what the delayed thread was doing.

Ref. R002, p. 107. See also: Discontinuity; Time Delta; Coupled Activities.

Delta debugging

Testing. An experimental method that repeatedly changes parts of a failure-inducing input or configuration and retests the result. It can simplify the failing case or isolate a difference between passing and failing cases. The outcome depends on a reliable test and the chosen decomposition. Minimality is relative to the algorithm and reduction operations, rather than a guarantee of the globally smallest explanation.

Ref. R154, Simplification and isolation algorithms. See also: Test-case reduction; Minimal reproducer; Regression bisection.

Delta temporality

Metrics. An aggregation convention in which a metric point describes activity during its own start-to-end interval, normally following the previous interval. It is useful for representing increments directly. Missing exports create missing intervals rather than a later cumulative value that necessarily contains them. Overlapping windows, duplicated points, or a mismatch between producer and backend expectations can distort totals and rates.

Ref. R261, Temporality. See also: Cumulative temporality; OTLP; Telemetry loss.

Denormalized Message

Trace patterns. A textual message produced by combining structured fields, values, and trace-wide constants into a narrative form. This can support text-oriented analysis and search. The transformation should preserve field meanings and boundaries because reconstructing the original structured record from free text may be ambiguous.

Ref. R002, p. 108. See also: Text Trace; Trace Schema; Trace Constants.

Density Distribution

Trace patterns. The distribution of selected message groupings across a trace or between traces. Local concentration can be compared with the surrounding message population to locate recurrent or changed activity. The selection rule and interval must remain comparable; changing instrumentation can alter the apparent distribution.

Ref. R002, p. 109. See also: Relative Density; Statement Density and Current; Inter-Correlation.

DEP Data Execution Prevention
Memory. A Windows protection mechanism that uses processor no-execute page protection to prevent execution from memory regions not marked executable. A violation can expose an invalid control transfer into data memory. We examine the attempted execution address and preceding state. DEP explains why execution was blocked; it does not establish whether the transfer originated in a corrupted function pointer, an incompatible code-generation practice, or another mechanism.

Ref. R128, Introduction; How Data Execution Prevention Works. See also: Memory protection; Access violation; Program counter.

Dependability
Foundations. The ability to deliver service that can justifiably be trusted, encompassing availability, reliability, safety, integrity, and maintainability. Diagnostic evidence informs assessment of the attributes relevant to the intended service and operating context.

Ref. R003, §2.3. See also: Failure; Software diagnostics; Root cause analysis.

Dereference Fixpoint
Memory patterns. A memory location contains its own address, so dereferencing that address yields the same address again. The name borrows the mathematical idea of a fixed point and identifies an exact self-reference that we can recognize in raw memory. Its presence describes a relationship between stored value and location; determining its significance requires the surrounding layout and execution context.

Ref. R881, definition and pointer examples. See also: Dereference Nearpoint; Execution Residue; Pointer Class.

Dereference Nearpoint
Memory patterns. A pointer value refers to a nearby location in the same examined memory region, generalizing the exact self-reference of a Dereference Fixpoint. We look for these local relationships and repeated references when classifying raw stack data. They may reveal propagated exception or structure addresses, while their diagnostic meaning still depends on validating the referenced memory and surrounding operation.

Ref. R817, Crash Dump Analysis Patterns (Part 285). See also: Dereference Fixpoint; Execution Residue; Latent Structure.

Design Value
Memory patterns. A value resembles an intentional design constant, such as a power-of-two capacity, a boundary value, or a regular flag pattern. It can suggest a limit reached or a structural constraint behind the observed behavior. We interpret it in the object's context and compare surrounding evidence. Numerical regularity, or coincidental symbol resolution, is insufficient to establish the intended design.

Ref. R669, Design Value pattern and diagnostic examples. See also: Abnormal Value; Counter Value; Coincidental Symbolic Information.

Determinism

Testing. The property that the same specified inputs and initial state produce the same relevant outcome under a defined execution model. It makes comparisons between experiments easier to interpret. The boundary of what counts as an input must include randomness, dependencies, and any scheduling or device effects that influence the result. A deterministic configuration may change performance and still require version-specific verification.

Ref. R174, Deterministic algorithms and platform/release limitations. See also: Random seed; Flaky test; Reproducible build.

Deviant Module

Memory patterns. A module differs from expectations in its name, location, protection, size, resources, imports, or embedded identifying text. We investigate the deviation in context and compare it with legitimate loading and code-generation behavior. The anomaly is a reason to examine provenance and purpose, not an automatic malware classification or proof that the module caused the incident.

Ref. R782, Crash Dump Analysis Patterns (Part 179). See also: Hidden Module; Template Module; Module Variety.

Deviant Token

Memory patterns. A process or thread has a security token that differs from the identity, groups, privileges, or impersonation state expected for the operation under investigation. We inspect the actual token and compare it with the intended execution context. The discrepancy helps explain access behavior, but its significance depends on whether the identity change was required and correctly scoped.

Ref. R540, Pattern description: Deviant Token. See also: Changed Environment; Special Process; Shared Structure.

Diachronic Module

Memory patterns. The same module appears in relevant active or waiting stacks at different capture times, even when the participating thread identifiers change. We compare its recurring role across the snapshot collection and correlate available traces. Repetition can identify a persistent performance dependency; it supports a causal hypothesis that still requires context, rather than proving fault from frequency alone.

Ref. R555, Pattern description: Diachronic Module. See also: Ubiquitous Component (kernel space); Spiking Thread; Stack Trace Change.

Diagnostic Analysis Pattern Tool

AI and ML. A proposed Model Context Protocol tool that exposes a diagnostic method as a callable capability. It can coordinate lower-level operations and return structured evidence, an operational plan, explanatory guidance, and candidate problem patterns. We retain the method and its limitations when assessing the result; executing the tool does not make every candidate a confirmed diagnosis.

Ref. R880, Vocabulary; tool output. See also: Analysis Pattern Agent; Operational MCP Tool; Analysis Pattern Square.

Diagnostic artifact

Foundations. A recorded or retained representation that can be examined for signs of software structure and behavior. Examples include memory dumps, logs, traces, debugger transcripts, configuration, and problem descriptions. Its diagnostic value depends on what was captured, when and how it was captured, and which transformations occurred before analysis.

Ref. R001, pp. 244–246; pp. 264–266. See also: Diagnostic evidence; Memory dump analysis; Software diagnostic space.

Diagnostic evidence

Foundations. Observations from artifacts and controlled investigations that support or constrain an explanation. Evidence includes values, relationships, distributions, absences under known collection conditions, and changes across cases. We preserve the distinction between what the artifact shows and what we infer, so another analyst can assess the same explanation and its unresolved assumptions.

Ref. R001, p. 136; p. 164; pp. 239–241. See also: Diagnostic indicator; Diagnostic hypothesis; Negative evidence.

Diagnostic hypothesis

Foundations. A provisional explanation or prediction that guides the next analysis action. It should connect observed indicators to a possible mechanism and identify evidence that could support, weaken, or distinguish it from alternatives. Writing down the hypothesis prevents a familiar pattern name or an early suspicion from silently becoming the investigation's conclusion.

Ref. R001, p. 136; p. 175; pp. 253–254. See also: Abduction; Causal model; Diagnostic evidence.

Diagnostic indicator

Foundations. An observable sign used to recognize structure, behavior, or a possible problem. A blocked thread, repeated error, unusual distribution, or missing expected message becomes useful when interpreted in context. An indicator can participate in several patterns, and its presence alone may be insufficient to establish either a defect or its mechanism.

Ref. R001, pp. 216–219; pp. 264–266. See also: Diagnostic evidence; Problem pattern; Analysis pattern.

Diagnostic operad

Foundations. A proposed description of composed diagnostic operations used to extract indicators from artifacts. Operations may transform data, query several artifacts, or accept additional parameters, and a useful composition can cross the boundaries of named analysis patterns. The mathematical terminology motivates formalization; a particular implementation must specify its own composition rules and assumptions.

Ref. R001, pp. 279–281. See also: Elementary analysis pattern; Software codiagnostics; Analysis pattern.

Diagnostic port

Runtimes. An interprocess endpoint through which a .NET runtime accepts diagnostic connections, such as requests for events or dump collection. The transport and access rules depend on the operating system and configuration. A tool failing to connect may reflect endpoint discovery, namespace, permissions, or runtime settings rather than a failure of the target application. Connection details are part of the acquisition context.

Ref. R201, Transport, configuration, and security considerations. See also: EventPipe; dotnet-dump; dotnet-trace.

Diagnostic report

Foundations. An account of the artifacts examined, indicators found, interpretations supported, and actions recommended. It should make clear which conclusions are established and which require more evidence. A useful report also identifies collection gaps and the reason for the next action, allowing troubleshooting or debugging to continue without reconstructing the entire investigation.

Ref. R001, p. 164; p. 235; p. 264. See also: Analysis narrative; Diagnostic evidence; Problem description.

Diagnostics Canvas

Foundations. A structured view of a diagnostic investigation organized around input, analysis, and output. Input connects the problem description with artifact collection; analysis connects methods with problem patterns; output connects findings with recommendations. Each part can be considered at general and concrete levels, helping expose missing links in an otherwise plausible analysis.

Ref. R001, p. 235. See also: Analysis Pattern Square; Diagnostic report; APM.

Diagnostics-driven development

Foundations. A development approach that introduces diagnostic analysis while software is being constructed. Early trace statements and periodic memory analysis can expose state, ordering, resource, and functional problems before production incidents. The same pattern languages used after deployment can guide what developers instrument, what artifacts they inspect, and which observations deserve further debugging.

Ref. R001, p. 289. See also: Debugware; Declarative Trace; Software diagnostics engineering.

Dia|gram

Narratology. A graphical language proposed for representing diagnostic artifacts and analysis patterns, including memory structures, trace messages, activity, and relationships. Its visual forms support communication across different artifact types and can illustrate how transformations reveal patterns. The diagram remains a selected representation, so its meaning depends on the stated mapping from artifact data to visual elements.

Ref. R001, p. 251; pp. 355–360. See also: Trace Shape; Software narratology; Diagnostic artifact.

Diags Diagnostic Analysis Gestures

Foundations. Configurations of diagnostic actions across tools and time, viewed as diagnostic analysis gestures. A gesture can include commands, interface actions, techniques, and interpretation; its narrative records how those actions form an investigation. The concept makes workflow and tool transitions explicit, including how a familiar investigation is adapted to another platform or artifact.

Ref. R001, pp. 331–332. See also: Logue; Analysis narrative; Pattern repertoire.

Dialog Box

Memory patterns. A thread is engaged in dialog processing, and captured parameters or dialog-template data can reveal the interface's caption and controls. We reconstruct that information when the incident lacks a screenshot or the dialog was inaccessible to the reporter. The resulting text helps explain a modal wait or diagnostic prompt, subject to the available memory and target layout.

Ref. R763, Crash Dump Analysis Patterns (Part 128). See also: Message Box; String Parameter; Blocked Thread.

Dialogue

Trace patterns. An exchange in which sources and targets alternate roles through requests, responses, or other related messages. Participants may be hosts, processes, threads, or user-interface objects. A missing response raises a question about the exchange and the completeness of its recording, rather than establishing that no response occurred.

Ref. R002, pp. 110–112. See also: Linked Messages; Incomplete History; Opposition Messages.

Diamathics

Foundations. A proposed name for mathematics applied to diagnostics, including models of artifacts, transformations, relationships, and diagnostic reasoning. Integral Diamathics emphasizes connections among these mathematical perspectives and the investigation of causes. A mathematical analogy can generate a useful diagnostic question; an asserted mathematical result additionally requires a defined model and satisfied assumptions.

Ref. R001, pp. 290–291. See also: Metaphorical bijectionism; Diagnostic operad; Analysis pattern duality.

Diegetic Messages

Trace patterns. Messages that tell the broader story of computation, such as one module reporting another module’s integrity check. This differs from a local trace statement recording only a component’s own call, although self-status messages can also reveal the broader story. The narrative position helps us distinguish local execution evidence from observation and summary.

Ref. R002, pp. 113–114. See also: Gossip; Implementation Discourse; Software narratology.

Differential testing

Testing. Testing that compares results from multiple implementations or configurations on related inputs. A disagreement supplies evidence that their behaviors differ and can reveal defects where an independent expected result is difficult to compute. It does not immediately identify which implementation is wrong. Shared defects, permitted variation, undefined behavior, and incompatible configuration must be considered before interpreting the discrepancy.

Ref. R158, Differential comparison and test interpretation. See also: Test oracle; Metamorphic testing; Reproducer.

Directing Module

Memory patterns. A higher-level module directs work performed by other components and may retain the objects or handles that determine their resource lifetimes. We compare allocation paths and common callers to identify this controlling relationship. Memory attributed to a lower-level allocator can therefore require investigation of the directing module's requests and retention behavior rather than the allocator alone.

Ref. R764, Crash Dump Analysis Patterns (Part 131). See also: Coupled Modules; Memory leak; Pass Through Function.

Disassembly

Debuggers. A representation of machine-code bytes as processor instructions. It provides the executable operations behind a source statement or failing instruction address. We establish architecture, decoding mode, and a valid instruction boundary before interpreting the result. Bytes belonging to data or a damaged image can still produce plausible-looking instructions, so a readable disassembly alone does not establish a legitimate execution path.

Ref. R029, Section 9.6. See also: Executable image; Program counter; Source mapping.

Disassembly Ambiguity

Memory patterns. Backward disassembly admits different instruction boundaries and can produce misleading code around a candidate return address. The apparent sequence may change with the requested instruction count or debugger behavior. We validate it through forward decoding from a credible boundary and surrounding control flow. An implausible backward listing is not sufficient evidence of corrupted or unexpectedly executed code.

Ref. R685, Disassembly Ambiguity pattern and diagnostic examples. See also: Disassembly; Wild Code; Coincidental Symbolic Information.

Disassembly Summary

Memory patterns. A prose account of an instruction sequence condenses individual operations and proposes their combined role. The pattern includes summaries produced by language models from disassembly text. We validate the account against instructions, calling conventions, and available context, keeping observed operations separate from inferred purpose; an external summarizer may lack the memory and symbolic information available to the debugger.

Ref. R573, Pattern description: Disassembly Summary. See also: Address Representations; False Function Parameters; Caller-n-Callee.

Disconnected Network Adapter

Memory patterns. Captured network-adapter state indicates that the local interface is disconnected or its media link is unavailable. Kernel networking structures can supply this evidence when an application reports a communication problem. We correlate adapter state with the affected operation and capture time. A local link condition narrows the investigation, but does not describe every remote endpoint or network path.

Ref. R654, Disconnected Network Adapter pattern and diagnostic examples. See also: Network partition; Diagnostic evidence; Kernel memory dump.

Discontinuity

Trace patterns. A gap in the visible messages of a thread or adjoint thread of activity. It may correspond to waiting, computation without logging, altered collection, or an irrelevant background activity. We compare the gap with the reported interval and with other evidence before assigning a mechanism.

Ref. R002, pp. 115–116. See also: Delay Dynamics; Blackout; Visibility Limit.

Disk Packet Buildup

Memory patterns. Disk-transfer packets accumulate in outstanding or queued states instead of returning to their normal reusable population. We inspect storage-driver structures or the distribution of related IRPs, comparing the affected device with expected behavior. The buildup localizes stalled I/O and its associated requests; determining why progress stopped requires following the relevant storage path and dependencies.

Ref. R784, Crash Dump Analysis Patterns (Part 185). See also: Network Packet Buildup; Object Distribution Anomaly; Wait Chain.

Dispatch Level Spin

Memory patterns. Persistent execution in a spin or loop at Windows DISPATCH_LEVEL or a higher IRQL, where ordinary thread scheduling cannot preempt the activity on that processor. A damaged or unreleased spinlock is one possible explanation. We examine the processor's IRQL, current instructions, and synchronization state, then seek evidence of persistence. A single captured spin iteration can be normal; the level alone does not establish a hang.

Ref. R378, Opening elevated-IRQL spinning discussion and instruction examples. See also: IRQL; Spinlock; Spiking Thread.

Distributed Exception (managed code)

Memory patterns. An exception crosses a process or machine boundary while preserving information about its earlier stack. In applicable .NET implementations, an exception object's remote-stack string can expose that original context. We interpret remote frames in their originating environment; numeric addresses from another process cannot be resolved as though they belonged to the local dump's address space.

Ref. R552, Pattern description: Distributed Exception (managed code). See also: Coupled Processes; Coupled Machines; Managed Stack Trace.

Distributed Spike

Memory patterns. Processor consumption is spread among several threads and may extend across cooperating processes. We compare thread times and captured activity across the relevant group rather than expect a single dominant thread. Strong, weak, or semantic coupling can propagate work and help identify additional participants whose CPU use belongs to the same diagnostic episode.

Ref. R509, Pattern description: Distributed Spike. See also: Spiking Thread; Coupled Processes (weak); Coupled Processes (semantics).

Distributed tracing

Tracing. Recording related operations across process and service boundaries so that an execution can be reconstructed as a connected trace. Identifiers and propagated context associate local observations with a larger request or workflow. The trace is a model assembled from available instrumentation, not a complete recording of the system. Missing spans, sampling, asynchronous work, and clock differences constrain its interpretation.

Ref. R279, Sections 1–2. See also: Span; Context propagation; Dapper.

Distributed Wait Chain

Memory patterns. Similar wait chains recur across processes or sessions, exposing a repeated dependency structure within the captured system. We compare their shapes and endpoints to identify a shared blocking component or recurring relationship. Distribution here concerns the pattern across the process population; it does not require multiple hosts. Repetition strengthens a comparison but still needs an explanation of the blocking mechanism.

Ref. R666, Distributed Wait Chain pattern and diagnostic examples. See also: Wait Chain; Blocking Module; Coupled Processes.

Divide by Zero (kernel mode)

Memory patterns. A kernel-mode instruction faults because its integer divisor is zero. Different Windows bugchecks can expose the same underlying arithmetic exception, so we recover the exception context and inspect the actual instruction and operands. The bugcheck's outer classification and currently named process are insufficient to identify how the zero value reached the driver computation.

Ref. R499, Pattern description: Divide by Zero (kernel mode). See also: Divide by Zero (user mode); Exception Stack Trace [Memory patterns]; Fault Context.

Divide by Zero (user mode)

Memory patterns. A user-mode integer-division instruction faults because its divisor is zero. We verify the exception, instruction, and operand in the saved execution context, then trace how the divisor was computed or supplied. The immediate arithmetic condition is usually straightforward; understanding the application failure requires connecting that value to the input, state, or earlier operation that produced it.

Ref. R744, Crash Dump Analysis Patterns (Part 78a). See also: Exception Stack Trace [Memory patterns]; Fault Context; Data Correlation (function parameters).

DMP Defect Mechanism Patterns

Foundations. Named, reusable descriptions of mechanisms that can produce diagnostic problem patterns. They connect indicators with software internals and help choose a debugging strategy or construct a model of a defect. The relationship is many-to-many: one mechanism can produce several signs, while similar signs can result from several mechanisms.

Ref. R001, pp. 369–370. See also: Defect mechanism; APM; Software pathology.

DNS failure

Networks. Failure to obtain the name-resolution result an operation requires. Domain Name System responses distinguish conditions such as a nonexistent name and a server failure; a missing response is a different observation again. Resolver caches, search rules, and network paths affect what a caller sees. Preserve the queried name, record type, resolver, response code, and timing before interpreting an application's generic connection error.

Ref. R316, Section 4.1.1: Header section format; RCODE. See also: Timeout [Distributed systems]; Packet capture; HTTP status code.

Dominant Event Sequence

Trace patterns. A comparatively complete or extensive event sequence selected as a candidate for the reported activity when the problem description lacks identification details. The sequence helps form an initial hypothesis about the relevant instance. That hypothesis needs confirmation from basic facts or independent identifying evidence.

Ref. R002, p. 117. See also: Basic Facts; Event Sequence Phase; Use Case Trail.

Dominator tree

Runtimes. A representation of heap reachability in which an object dominates another when every relevant path from a root to the latter passes through the former. Immediate dominators organize objects into a tree useful for retained-memory analysis. A tree edge expresses a dominance relationship, not necessarily a direct object reference. Follow actual reference paths when explaining the application's retention mechanism.

Ref. R204, Domination, immediate dominators, and reference-graph distinction. See also: Retained size; GC root; Heap dump.

dotnet-counters

Runtimes. A tool for monitoring or collecting selected .NET runtime and application counters and metrics. It provides a relatively quick view of changes such as allocation pressure or exception activity before more detailed evidence is collected. Each measurement requires its own unit, interval, and aggregation interpretation. An elevated value identifies behavior to investigate; it does not identify the responsible call path by itself.

Ref. R197, Description; monitor and collect. See also: dotnet-trace; Garbage collection; Allocation profiling.

dotnet-dump

Runtimes. A .NET command-line tool for collecting process dumps and analyzing managed state with SOS commands. It supports investigations of threads, exceptions, object populations, and retention without requiring a full native-debugger workflow. It is not itself a native debugger, and available analysis depends on the collected artifact. Acquisition type, target runtime, and missing memory affect what can be reconstructed.

Ref. R195, Description; collect and analyze. See also: SOS; ClrMD; dotnet-gcdump.

dotnet-gcdump

Runtimes. A tool that requests garbage collection and uses runtime events to reconstruct a managed heap graph. The resulting .gcdump supports type and retention analysis with a different content and acquisition cost from a full process dump. It does not preserve arbitrary process memory or complete execution state. Collection effects, event loss, and target constraints can limit the graph available for interpretation.

Ref. R198, Description; collection and troubleshooting. See also: dotnet-dump; Heap dump; GC root.

dotnet-stack

Runtimes. A .NET tool that collects managed stack traces from a target process. It provides a focused view of what managed threads report at collection time and can help identify common waiting or execution contexts. A stack collection does not explain all preceding events or native activity. Repeated captures are useful only when differences in timing and acquisition scope are understood.

Ref. R202, Description; report. See also: Thread dump; SOS; dotnet-trace.

dotnet-trace

Runtimes. A tool for collecting diagnostic events from .NET applications using supported runtime and platform mechanisms. Selected providers, profiles, and collection duration determine which behavior the resulting trace can describe. A missing event may mean that its provider or keyword was not enabled. Runtime versions and operating-system integrations also affect the available data, so collection settings should accompany the artifact.

Ref. R196, Collection modes, profiles, and providers. See also: EventPipe; Diagnostic port; dotnet-counters.

Double free

Memory. A second release of storage that has already been released without a valid intervening allocation relationship. It can expose inconsistent ownership or duplicated cleanup paths and may damage allocator state. We identify the logical allocation instance, not just the numerical address, because an allocator can legitimately reuse an address. Release history is therefore more informative than two matching pointer values alone.

Ref. R047, Description. See also: Use-after-free; Heap corruption; Memory leak.

Double Free (kernel pool)

Memory patterns. Kernel code attempts to release a pool allocation that has already been freed. Pool state and the relevant bugcheck parameters can distinguish this from a first legitimate free or general corruption. We identify the allocation and current caller, then investigate ownership history: the component performing the second release may not be the component that first relinquished the block.

Ref. R718, Crash Dump Analysis Patterns (Part 23b). See also: Double Free (process heap); Dynamic Memory Corruption (kernel pool); Double IRP Completion.

Double Free (process heap)

Memory patterns. A process-heap allocation is released more than once, with detection depending on allocator state and intervening operations. Corruption may surface during a later legitimate free rather than at the repeated release. We inspect allocation history where available and use suitable free checking or page-heap instrumentation during reproduction. A delayed manifestation does not change the invalid lifetime operation.

Ref. R596, Double Free (process heap) pattern and diagnostic examples. See also: Double free; Dynamic Memory Corruption (process heap); PageHeap.

Double IRP Completion

Memory patterns. An I/O request packet is completed after it has already been completed, often because separate participants both believe they own completion responsibility. We inspect the reported IRP and device-stack information to reconstruct that ownership. The second completion may expose the failure after evidence of the first has disappeared, making the current reporting driver only part of the investigation.

Ref. R771, Crash Dump Analysis Patterns (Part 149). See also: IRP; Double Free (kernel pool); Self-Diagnosis (kernel mode).

DPC Deferred procedure call

Execution. A Windows mechanism for deferring work from an interrupt service routine or another suitable context to a later execution point. It allows the immediate interrupt handler to remain short. We examine queued and executing DPC work when reconstructing latency or watchdog failures. The presence of a DPC routine in a stack identifies current activity, while its duration and dependencies establish its significance.

Ref. R095, DPCs and interrupt service routines. See also: IRQL; Context switch; Bug check.

DPC Stack Collection

Memory patterns. Per-processor DPC stack regions are collected for examination of current activity and execution residue. Processor structures and the appropriate stack-size information establish the regions to inspect. We can then search them for symbolic references or rough traces beyond ordinary thread stacks. Captured bytes may describe earlier DPC activity, so residue requires the same validation as other reconstructed execution evidence.

Ref. R699, DPC Stack Collection pattern and diagnostic examples. See also: DPC; Stack Trace Collection (CPUs); Rough Stack Trace Collection (unmanaged space).

DRD

Program analysis. A Valgrind thread-error detector that analyzes memory accesses and recognized synchronization during program execution. It can expose races and misuse of supported threading operations, with reports linking the relevant contexts. The analysis describes the observed instrumented behavior. Custom synchronization, incomplete annotations, and unexplored schedules affect its coverage and can require additional reasoning about the program's intended ordering.

Ref. R187, Overview; detected errors and synchronization. See also: Valgrind; Helgrind; ThreadSanitizer.

Driver Device Collection

Memory patterns. A collection of driver and device objects examined together, often against a saved reference collection. Differences can identify objects requiring further investigation and help relate drivers, file systems, and devices. We retain object identity and collection scope during comparison. An added or changed object identifies a difference in the system; its role still requires interpretation.

Ref. R837, Pattern discussion. See also: Module Collection; IRP; Deviant Module.

Driver Verifier

Native tools. Windows facilities that subject selected drivers to additional checking and configured stress conditions to expose invalid behavior. Detected violations can cause a bug check that preserves useful evidence near the operation. We interpret that failure in the context of the enabled verification options. Verifier changes the environment deliberately, so its findings identify a checked violation rather than reproducing every aspect of the original workload.

Ref. R069, Introduction; testing and debugging requirements. See also: Bug check; IRQL; Memory corruption.

Drone Message

Trace patterns. A message repeated throughout a trace or activity at approximately regular intervals. Its recurrence can supply a relative timing reference when explicit timing information is limited. The reference is useful only while the repetition remains stable; scheduling delays and missing records can disturb its apparent rhythm.

Ref. R002, p. 118. See also: Periodic Message Block; Time Delta; Fourier Activity.

Dry Weight

Memory patterns. Loaded module images account for a substantial part of a process's memory footprint. A product upgrade can increase this image contribution through new frameworks or components, producing an immediate usage increase that resembles a leak. We separate image memory from other allocations and compare versions. A larger baseline explains some growth without excluding an independent allocation-retention problem.

Ref. R672, Dry Weight pattern and diagnostic examples. See also: Module Variety; Memory map; Memory leak.

dSYM

Artifacts. An Apple-platform bundle containing debug symbols for a particular binary. The binary and its dSYM must have matching build UUIDs for reliable symbolication. We retain the symbols for the exact distributed build, including dependent components. Rebuilding nominally identical source is not a substitute for the original matching artifact, because build settings or toolchain differences can produce a different image.

Ref. R099, Overview; Locate the build UUID of the binary. See also: Mach-O; Build ID; Symbolication.

Dual Interface

Debuggers. A diagnostic tool offering both graphical and command-line access to its capabilities, either within one component or through cooperating components. The two interfaces support interactive investigation and automation. We check how their options and defaults correspond, so a result obtained manually can be reproduced by the scripted path under equivalent conditions.

Ref. R853, Pattern discussion. See also: Tool Façade; Debugware; Reproducer.

Dual Stack Trace

Memory patterns. A pair of user-space and kernel-space stack traces representing the two sides of a thread's execution. We use the pair to connect a process dump with a kernel or complete dump, matching the thread and capture context before comparing frames. The additional side can expose a wait or dependency that the first artifact cannot show.

Ref. R515, Pattern description: Dual Stack Trace. See also: Stack Trace [Memory patterns]; Stack Trace Collection; Wait Chain.

Dump Context

Memory patterns. The surrounding information needed to interpret a memory dump: the reported issue, collection method, system setting, related records, and earlier dumps or analyses. We organize that context so the snapshot can be connected with the incident it is meant to explain. A consistent checklist improves analysis quality and reduces repeated requests for basic missing information.

Ref. R822, Crash Dump Analysis Patterns (Part 301). See also: Trace Context; Paratext; Diagnostic evidence.

Dumping Memory

Debuggers. Displaying or extracting the contents of a selected memory range as an elementary inspection action. We choose the range and representation according to the current question and preserve the address context. Here dumping means examining existing memory contents; acquisition of a new memory-dump artifact is a separate operation with its own scope and timing.

Ref. R001, pp. 268–270. See also: Memory dump; Memory Region; Searching Memory.

Dumping Object State

Debuggers. Displaying the fields or other available state of a selected object. The operation exposes data for subsequent interpretation and comparison with the object's expected relationships. We verify identity, type information, and address context, and distinguish unreadable or absent fields from valid values. A formatted structure is useful only to the extent that its interpretation is justified.

Ref. R001, pp. 268–270. See also: Memory examination; Structure Field Collection; Dumping Memory.

Dumping Thread State

Debuggers. Displaying the state associated with a selected thread, including available registers, execution status, and related structures. We preserve thread and process identity and distinguish the capture point from a reconstructed history. The resulting observation supports analysis of execution or waiting, but a single state does not by itself establish persistence or responsibility for a problem.

Ref. R001, pp. 268–270. See also: Thread; CPU register; Listing Stack Traces.

Duplicated Module

Memory patterns. The same module name appears in multiple loaded instances, often from different paths or versions. We compare image identity, deployment paths, and load order to investigate incompatible dependencies. An address appended to a debugger's module name disambiguates instances; it is not a reliable indication of which instance was loaded most recently.

Ref. R493, Pattern description: Duplicated Module. See also: Module Variety; Not My Version; Missing Component (general).

Duplicate Extension

Memory patterns. Two loaded debugger extensions provide overlapping commands, and an unintended version receives commands that previously worked. We inspect the extension list, versions, and command resolution before treating the resulting failures as properties of the target process. Explicitly selecting or retaining the appropriate extension restores a consistent analysis context; matching the runtime remains a separate requirement.

Ref. R527, Pattern description: Duplicate Extension. See also: No Component Symbols; Unrecognizable Symbolic Information; Incorrect Stack Trace.

DWARF

Artifacts. A standardized representation of debugging information describing program entities and their relationship to generated code. It can encode types, source locations, variable locations, and call-related information. We interpret it together with the executable and the supported format version. Debug information can describe where a value would be found without guaranteeing that the value remains available at every optimized instruction position.

Ref. R076, Sections 2.1, 2.6, 6.2, and 6.4. See also: Symbol file; Optimized-out variable; Stack unwinding.

Dynamic analysis

Program analysis. Analysis of behavior produced by executing a program, often with additional instrumentation. It can expose actual memory accesses, calls, allocations, or ordering relations for the observed workload. The resulting evidence is tied to those executions and to the analysis mechanism. Instrumentation can change timing and resource use, while unexecuted paths remain outside the observation.

Ref. R184, Introduction; dynamic binary instrumentation. See also: Static analysis; Runtime verification; Valgrind.

Dynamic invariant detection

Program analysis. Inference of candidate properties from observed program values. A detector may report that a value was always positive or that two fields maintained a relationship across the collected runs. These are likely invariants of the sample, not automatically universal properties. They can suggest assertions, reveal changes between runs, or expose a workload that never exercised an important boundary.

Ref. R145, Abstract; dynamic detection of likely invariants. See also: Daikon; Invariant; Assertion.

Dynamic Memory Corruption (kernel pool)

Memory patterns. Damage to dynamically allocated kernel pool memory, often detected when a later allocation or deallocation examines corrupted bookkeeping. We correlate the damaged block, pool tag, contents, and participating drivers, then use suitable instrumentation when reproduction is possible. The stack that reports corruption identifies its detection point; the earlier write responsible may belong to a different execution path.

Ref. R702, Crash Dump Analysis Patterns (Part 2b). See also: Dynamic Memory Corruption (process heap); Lateral Damage; Double Free (kernel pool).

Dynamic Memory Corruption (managed heap)

Memory patterns. Damage to managed-heap objects or their organization, often discovered while the CLR traverses the heap during garbage collection. We examine heap verification results, the last valid object, and surrounding raw data. A collection interrupted by a crash may itself leave analysis state incomplete, so inconsistent tool output calls for comparison with additional captures.

Ref. R457, Pattern description: Dynamic Memory Corruption (managed heap). See also: Dynamic Memory Corruption (process heap); Garbage collection; Corrupt Structure.

Dynamic Memory Corruption (process heap)

Memory patterns. Damage to dynamically allocated process memory that becomes visible during allocation, deallocation, or later use. Heap-management frames may identify where corruption was detected while the original write occurred elsewhere. We inspect the affected blocks and use suitable heap checking in a reproduction to move detection closer to the damaging operation, retaining the distinction between detector and offender.

Ref. R580, Dynamic Memory Corruption (process heap) pattern and diagnostic examples. See also: Heap corruption; PageHeap; Lateral Damage.

Dynamic slicing

Program analysis. Selection of the operations that contributed to a particular value or event in a particular execution. Unlike a static slice covering possible executions, the criterion includes concrete execution context. This can reduce a large trace to relevant dependencies. The result still depends on the slicing model, including its treatment of aliases, control dependencies, and interactions outside the captured execution.

Ref. R146, Abstract; computation-specific slicing criterion. See also: Program slicing; Dynamic analysis; Fault localization.

E

Early Crash Dump

Memory patterns. A dump obtained at an earlier significant failure event, before subsequent handling or secondary damage changes the available evidence. Capturing a relevant first-chance exception may preserve a clearer exception context than a later termination dump. We choose the trigger according to the suspected sequence and expected exceptions. Earlier evidence narrows the investigation, but even the first observed fault can follow an earlier corrupting operation.

Ref. R371, Opening early-capture discussion and exception-handling example. See also: First-chance exception; Hidden Exception; False Positive Dump.

eBPF Extended Berkeley Packet Filter

Native tools. A Linux facility for running verified programs at supported attachment points, including tracing-related hooks. It enables collection and aggregation near selected kernel or user-space events through the relevant interfaces. We document the program, attachment points, maps, and kernel capabilities as part of the measurement. Verification constrains program safety; it does not prove that the resulting metric or inferred explanation answers the diagnostic question correctly.

Ref. R040, Questions and Answers; ABI and tracing questions. See also: Kprobe; Uprobe; Kernel tracepoint.

Effect Component

Memory patterns. A component appears because it participates in handling or recording an event that has already occurred. A dump-writing driver on a bugchecking thread is one example. We reconstruct the direction of the event sequence before assigning causal significance, since a module left in execution residue by the response can be mistaken for the original offender.

Ref. R503, Pattern description: Effect Component. See also: Execution Residue; Custom Exception Handler; Diagnostic hypothesis.

Elapsed time

Performance. The duration between a defined start and end, including both execution and waiting within that interval. It answers how long an observer waited, whereas CPU time describes processor use. Parallel threads can consume more total CPU time than the elapsed interval. Identify the operation boundaries and clock semantics before comparing durations or attributing their difference to a particular wait.

Ref. R397, Real time and process time. See also: CPU time; Wall-clock time; Monotonic clock.

Elementary analysis pattern

Foundations. A basic diagnostic action such as identifying threads, listing modules, examining memory, or establishing symbol information. These operations form reusable building blocks for more elaborate investigations and can be described independently of a debugger’s command spelling. Their results still require interpretation in the context of the artifact and the problem.

Ref. R001, pp. 268–270. See also: ADC; Analysis pattern; Elementary diagnostic pattern.

Elementary diagnostic pattern

Foundations. A readily recognizable class of reported behavior that helps determine an initial investigation direction. Examples include a crash, a hang or delay, an error message, a counter deviation, or a use-case deviation. It describes the problem as observed before deeper analysis identifies detailed indicators and possible mechanisms.

Ref. R001, p. 170. See also: Problem description; Problem pattern; Elementary analysis pattern.

ELF Executable and Linkable Format

Artifacts. A binary format used by many Unix-like systems for executable files, shared objects, relocatable objects, and core files. The header identifies architecture, class, byte order, and file type, while other structures organize its contents. We establish these properties before decoding records. An ELF core file and an ELF executable share a container family but represent different diagnostic artifacts.

Ref. R077, Section 2.1; file types; Section 2.2, ELF Identification. See also: Core dump; Executable image; DWARF.

Embedded Comment

Memory patterns. A dump contains explanatory text or recorded observations added by its collection mechanism. These can describe the trigger, tool, or monitoring state that surrounded acquisition and help interpret the captured execution. We distinguish that external account from the memory evidence itself and retain its provenance, whether it occupies a formal comment stream or an accessible application buffer.

Ref. R514, Pattern description: Embedded Comment. See also: Dump Context; Manual Dump; Paratext.

Embedded Trace

Trace patterns. A trace carried inside the messages of an outer tracing framework. The outer record may supply uniform metadata while the embedded content has its own schema and boundaries. Both layers can contribute evidence, but parsing one layer must not silently discard the other.

Ref. R002, p. 119. See also: Container Trace; Trace Schema; Split Message.

Embedding

AI and ML. A learned vector representation that places related inputs near one another according to the model's training objective. Text embeddings can support retrieval of similar diagnostic descriptions or grouping of recurring reports. Proximity expresses learned similarity, not logical equivalence or causal identity. Tokenization, model version, and the information preserved by the representation affect which apparently related cases are returned.

Ref. R240, Sentence embeddings and semantic similarity. See also: Feature vector; Clustering; RAG.

Empty Stack Trace

Memory patterns. The debugger produces no useful call chain, perhaps showing only the current instruction and an unusable return address. We establish the actual stack region and inspect its raw contents for validated frame and return-address evidence. Manual reconstruction may recover a meaningful path even when the automatic trace is empty, but candidate addresses still require consistency checks.

Ref. R776, Crash Dump Analysis Patterns (Part 161). See also: Incorrect Stack Trace; Rough Stack Trace (unmanaged space); Execution Residue.

Empty Trace

Trace patterns. A trace with no usable messages, or far fewer messages than the recorded scenario should produce. The file can still be nonempty because of headers or preallocated storage. Analysis therefore begins with collection settings, capture timing, and decoding rather than interpreting emptiness as absence of execution.

Ref. R002, p. 120. See also: No Activity; Missing Component; No Trace Metafile.

Encoded Pointer

Memory patterns. A stored pointer contains encoding, authentication, tags, or other metadata that prevents its raw value from being used as an ordinary address. We identify the representation and apply the appropriate interpretation before resolving or following it. The required operation depends on the platform and scheme. Arbitrarily masking bits can manufacture a plausible address without recovering the intended pointer.

Ref. R700, Encoded Pointer pattern and diagnostic examples. See also: Invalid Pointer (general); Execution Residue; Pointer Class.

Environment Hint

Memory patterns. Environment information retained in an artifact provides a clue about configuration or installed software when supporting details are incomplete or inconsistent. Environment-variable values can suggest a product, path, or version worth checking. We use the clue to form a comparison or reproduction step, keeping it separate from proof that the hinted component participated in the observed failure.

Ref. R639, Environment Hint pattern and diagnostic examples. See also: Changed Environment; Inconsistent Dump; Module Hint.

Ephemeral container

Cloud. A container added to an existing Kubernetes Pod for a purpose such as interactive diagnosis. It can provide tools when an application image lacks a shell or the original container cannot be used. The visibility obtained depends on namespace targeting, runtime support, and permissions. Its observations describe the accessible Pod environment at that time and may not reproduce the terminated application's earlier state.

Ref. R294, Debugging with an ephemeral debug container. See also: Container log; Pod event; CrashLoopBackOff.

Equivalent Messages

Trace patterns. Messages treated as equivalent under an explicitly chosen meaning or analytical relation despite differences in wording or structure. This permits comparison and simplification across traces. The relation should preserve distinctions relevant to the diagnostic question; similar text or model similarity alone need not imply equivalent behavior.

Ref. R002, p. 121. See also: Quotient Trace; Ultrasimilar Messages; Inter-Correlation.

errno

Execution. A thread-local error indicator used by many C library and system-call interfaces. Its value is meaningful according to the return-value contract of the operation being examined; a successful operation need not clear it. We capture it at the relevant failure point. A later diagnostic print or unrelated library call can leave a stale or changed value that misdescribes the original failure.

Ref. R062, DESCRIPTION. See also: System call [Execution]; Thread-local storage; Signal [Execution].

Error

Foundations. A part of system state that can lead to service failure. Errors may be latent, detected, masked, or propagated. A logged error message requires interpretation before assigning the erroneous state or its cause.

Ref. R003, §2.2. See also: Fault; Failure; Error Message.

Error budget

Reliability. The amount of unreliability permitted by a service level objective within its evaluation period. For a success-ratio objective, it corresponds to the allowed fraction or count of unsuccessful eligible events. A budget makes the cost of failures explicit and can inform operational decisions. Its remaining value depends on the population and window; it is not a universal allowance independent of service usage.

Ref. R281, Managing Risk; error budgets. See also: SLO; Burn rate; SLI.

Error Distribution

Trace patterns. The placement and concentration of error messages across the trace, its activities, or its partitions. A cluster around one transition has a different diagnostic meaning from uniformly recurring background errors. We examine both the distribution and the underlying message contexts before choosing an investigation path.

Ref. R002, p. 122. See also: Error Message; Periodic Error; Trace Partition.

Error Message

Trace patterns. A message reporting an error explicitly or through a status value, exception, or return result. Its severity label is part of the logging design and may not agree with its diagnostic importance. The message must be related to the investigated occurrence, surrounding activity, and status-code convention.

Ref. R002, p. 123. See also: False Positive Error; Error Thread; Exception Stack Trace [Trace patterns].

Error Message Box

Execution. A named User Interface Problem Analysis pattern in which a visible error dialog starts the investigation. We record its text and context, then relate the dialog to the owning process, thread, and calling code. A dump can expose the path that displayed it. The message reports an observed condition; its wording alone does not identify the underlying defect mechanism.

Ref. R824, Pattern discussion. See also: Message Box; Error Message; Self-Diagnosis (user mode).

Error Powerset

Trace patterns. The subsets of the error codes or exception types observed in a trace, considered as candidate combinations for analysis and search. A useful subset can be more discriminating than one common error or the complete set. Selection follows context and activity; exhaustive enumeration can become impractical as the set grows.

Ref. R002, p. 124. See also: Error Message; Message Set; Defect Group.

Error Reporting Fault

Memory patterns. The mechanism that reports a failure itself fails, potentially causing repeated or recursive reporting. The published Windows example follows a chain of reporting processes, each handling a fault in another reporter. We trace those relationships toward the initial failure and account for the resources consumed by the reporting cascade, which can obscure or displace the original evidence.

Ref. R783, Crash Dump Analysis Patterns (Part 182). See also: Process Factory; Exception Reporting Thread; Lost Opportunity.

Error Thread

Trace patterns. The sequence of earlier messages from the same thread or adjoint thread that leads to a selected error. It provides a focused view of the local activity before failure was reported. A causal explanation may require leaving that thread to examine data or dependencies in other activities.

Ref. R002, p. 125. See also: Error Message; Back Trace; Thread of Activity.

ETW Event Tracing for Windows

Native tools. A Windows event-tracing infrastructure in which providers emit events, controllers configure sessions, and consumers process the recorded or live stream. It connects system and application observations through provider-defined event data. We retain session settings, provider identities, and time context with the trace. An absent event can reflect disabled instrumentation, filtering, or collection loss; it does not necessarily mean that the associated operation did not occur.

Ref. R070, Controllers; Providers; Consumers. See also: WPR; WPA; Process Monitor.

Evental Dumps

Memory patterns. Memory dumps are saved in response to selected events, such as thread creation or exit, forming snapshots around changes in execution. We inspect other threads and correlate the captures with logs to reconstruct surrounding behavior. The triggering event identifies why each artifact exists. It does not imply that the event itself caused the problem being investigated.

Ref. R677, Evental Dumps pattern and diagnostic examples. See also: Memory dump; Stack Trace Collection; Adjoint Space.

Event-loop blocking

Runtimes. A condition in which work executed on an event-loop thread prevents that loop from promptly handling other ready work. Long computations and synchronous operations can create this pattern even when the application contains asynchronous APIs. A delayed callback is evidence of scheduling delay, not automatically of its cause. Correlate callback execution, queued work, and blocking operations within the relevant runtime.

Ref. R227, Event loop; blocking callbacks and work. See also: asyncio debug mode; Node.js Inspector; cProfile.

EventPipe

Runtimes. A .NET runtime event-collection mechanism used by diagnostic tools and runtime or application event providers. It can preserve structured event sequences for later analysis, with the selected providers and buffers defining the observation. Available operating-system integrations vary across versions and platforms. Events omitted by configuration or lost during collection cannot be inferred from a complete-looking trace file.

Ref. R200, EventPipe overview; collection and integrations. See also: dotnet-trace; Diagnostic port; dotnet-counters.

Event Sequence Order

Trace patterns. The required or expected ordering of events for a particular operation. A departure can direct investigation toward startup dependencies or synchronization. We distinguish the order required by the system from the order in which records are buffered, collected, and displayed before interpreting an apparent violation.

Ref. R002, p. 126. See also: Data Reversal; Unsynchronized Messages; Use Case Trail.

Event Sequence Phase

Trace patterns. A relative position within recurring event sequences, used to separate intermingled instances when explicit instance identifiers are absent. Anchors and time intervals can help associate stages of the same activity. The reconstruction depends on assumptions about sequence order and timing and should retain ambiguous assignments.

Ref. R002, pp. 127–128. See also: Anchor Messages; Event Sequence Order; Dominant Event Sequence.

Event timestamp

Logs. The time attributed to an event by its originating system. In the OpenTelemetry log model, Timestamp describes occurrence time according to the source clock. It may differ from the time of collection or storage. Reconstructing order requires considering clock synchronization, resolution, buffering, and delayed emission; sorting event timestamps alone does not prove a causal order across independent machines.

Ref. R260, Timestamp. See also: Observed timestamp; Clock skew; Causal order.

Eviction

Cloud. In Kubernetes node-pressure handling, termination of Pods by the kubelet to reclaim resources when monitored pressure crosses configured thresholds. It differs from an application exit and from a kernel OOM kill. The relevant evidence includes node conditions, eviction reason, resource usage, and thresholds. Sampling can miss a fast pressure increase, so absence of a recorded high metric does not exclude the event.

Ref. R297, Node-pressure eviction; out-of-resource handling. See also: OOMKilled; Resource limit; Pod event.

Exception

Execution. An execution event that diverts control to special handling, such as a processor-detected fault or an explicitly raised software condition. An exception can be handled and execution can continue. We therefore inspect its type, handling stage, and surrounding activity before equating it with a crash. The place where an exception is detected can differ from the place where invalid state originated.

Ref. R018, Hardware and software exceptions. See also: First-chance exception; Second-chance exception; Signal [Execution].

Exception Collection

Memory patterns. Exception evidence is gathered across available memory spaces and representations rather than taken from one current exception view. Stored records, exception-processing stacks, managed exceptions, recorded heap failures, and raw-memory residue can contribute. We retain each item's context and relationship to other records. The collection may mix current, nested, handled, and historical events, so counting records does not count independent failures.

Ref. R694, Exception Collection pattern and diagnostic examples. See also: Stored Exception; Hidden Exception; Multiple Exceptions (managed space).

Exception Module

Memory patterns. A module contains the instruction or operation where an exception occurred. We identify it from the original exception context and validate the relevant stack or call evidence when necessary. This localizes the failure, but it does not settle responsibility: an invalid input, damaged state, or upstream caller can make a component fail while its own implementation is behaving as designed.

Ref. R795, Crash Dump Analysis Patterns (Part 217). See also: Exception Stack Trace [Memory patterns]; Fault Context; Well-Tested Module.

Exception record

Artifacts. A structured description of an exception, including its code, address, flags, and condition-specific parameters. On Windows, EXCEPTION_RECORD is distinct from the register context used to reconstruct the failing execution. We interpret parameters according to the exception type. Confusing the exception address with a referenced data address can direct investigation toward the wrong memory region.

Ref. R116, Members. See also: Exception; Register context; Access violation.

Exception Reporting Thread

Memory patterns. A thread performs failure reporting on behalf of another thread or operation. Its stack may show reporting infrastructure while the actual blocked or faulting execution appears elsewhere. We connect the reporter to the reported context and inspect both. This separation is particularly useful for asynchronous hang reports and handled exceptions whose reporting path does not identify the original failure site.

Ref. R808, Crash Dump Analysis Patterns (Part 257). See also: Exception Stack Trace [Memory patterns]; Error Reporting Fault; Blocked Thread.

Exception Stack Trace

Trace patterns. An exception-related call stack serialized into log output, either within one message or across several records. It supplies a local execution context for the reported exception. Wrapping, rethrowing, asynchronous boundaries, and interleaved output can affect interpretation, so the relationship between the frames and the report must be preserved.

Ref. R002, pp. 129–130. See also: Collapsed Message; Split Message; Error Thread.

Exception Stack Trace

Memory patterns. A stack trace contains exception-dispatch, handling, or reporting activity. Its frames can lead us to an exception record, saved context, and the operation that failed. We distinguish the processing stack from the original faulting execution, particularly when another thread reports the event. The absence of such frames in the current collection does not rule out a previous exception.

Ref. R754, Crash Dump Analysis Patterns (Part 105). See also: Exception Module; Exception Reporting Thread; Hidden Exception.

Exclusive time

Performance. Time attributed directly to a function or operation, excluding time attributed to its callees under the profiler's accounting model. It is commonly presented as self time. The quantity helps separate a wrapper from the work it invokes. CPU-based and elapsed-time profilers measure different things, and missing frames or attribution rules can change which operation receives the apparent cost.

Ref. R396, CPU utilization data columns: Self CPU. See also: Inclusive time; CPU time; Elapsed time.

Executable image

Artifacts. A binary representation prepared for loading and execution, with code, data, and metadata organized according to its platform format. Its file layout is not necessarily identical to its loaded memory layout. We retain section mappings, relocation context, and build identity when comparing the two. A numerical offset in a file cannot always be interpreted directly as an offset in process memory.

Ref. R074, General Concepts; headers; sections; Debug Directory. See also: PE; ELF; Mach-O.

Execution Residue

Memory patterns. Remnants of earlier execution that remain in memory after their original use has ended. Raw stacks may preserve code addresses, arguments, strings, or exception-related data beyond the currently reconstructed frames. We combine these fragments with code and architectural context to propose earlier activity. Their survival is selective, and a symbol-like value requires validation before it can support a reconstructed call or sequence.

Ref. R386, Opening definition and raw-stack reconstruction examples. See also: Historical Information; Coincidental Symbolic Information; Hidden Exception.

Execution Residue (managed space)

Memory patterns. Managed execution leaves recoverable call and object references in stack memory beyond the immediately displayed frames. Runtime-aware stack and object inspection can expose earlier operations, values, and exception objects. We distinguish these residual references from the active call chain and current ownership, using them as historical clues that require correlation with the present thread state and incident.

Ref. R736, Crash Dump Analysis Patterns (Part 60b). See also: Execution Residue; Hidden Exception (managed space); Managed Stack Trace.

Execution Residue (unmanaged space, kernel)

Memory patterns. Remnants of earlier kernel execution remain in a thread's stack region beyond its current call chain. Dump-generation and diagnostic components may overwrite that evidence after a bug check. We inspect the relevant boundaries and, where present, a preserved pre-bug-check stack copy. The available residue constrains reconstruction of past activity; its absence does not prove the activity never occurred.

Ref. R614, Execution Residue (unmanaged space, kernel) pattern and diagnostic examples. See also: Execution Residue; Past Stack Trace; Self-Diagnosis (kernel mode).

Exemplar

Metrics. An example observation retained alongside an aggregate, with context that can connect a metric point to a particular execution. An OpenTelemetry exemplar can preserve a value, timestamp, filtered attributes, and trace or span identifiers. It provides a route from an unusual aggregate to a concrete artifact. It does not establish that the linked request is representative or responsible for the entire aggregate.

Ref. R268, Exemplar. See also: Histogram; Trace identifier; Log correlation.

EXPLAIN

Data systems. A database command for displaying the plan and related information for a statement. In PostgreSQL, ordinary EXPLAIN principally reports planner estimates; cost units are not milliseconds. The output helps identify how the database expects to access and combine data. Options and product semantics matter, so distinguish an estimated plan from a plan accompanied by measurements gathered while actually executing the statement.

Ref. R298, EXPLAIN Basics. See also: Query plan; EXPLAIN ANALYZE; Slow query log.

Explainable AI

AI and ML. Methods that make aspects of a model's predictions or behavior intelligible to a person. Explanations may describe influential features, local approximations, or global model structure. In software diagnosis, they help inspect why a classifier favored a category. Explaining a prediction does not establish that the prediction is correct or that the highlighted features caused the underlying software failure.

Ref. R243, Explaining individual predictions; R244, Additive feature attribution. See also: LIME; SHAP; Feature vector.

EXPLAIN ANALYZE

Data systems. A PostgreSQL EXPLAIN mode that executes the statement and reports observed execution statistics alongside the estimates. It allows estimated and actual row counts or timing to be compared, but execution can change data or produce other side effects. Instrumentation adds overhead, and ordinary client delivery costs are not all included. The resulting artifact measures this instrumented execution under its particular conditions.

Ref. R298, EXPLAIN ANALYZE; caveats. See also: EXPLAIN; Query plan; Transaction isolation.

Explanation Trace

Trace patterns. A selected sequence or set of trace messages assembled to support an explanation of observed behavior. Different proposed mechanisms may require different explanation traces. The selected evidence should include qualifications and conflicting observations where relevant, so that the explanation can be assessed independently of the analyst's route through the data.

Ref. R002, p. 131. See also: CoTrace; Causal History; Message Annotations.

Exporter
Telemetry. In an OpenTelemetry pipeline, a component that sends telemetry to a configured destination using a supported format and protocol. Export can fail independently of instrumentation and collection. Queue limits, retries, destination rejection, and shutdown handling determine which data reaches the next stage. Success establishes the result of the exporter's own operation, whose acknowledgement and durability semantics must still be examined.

Ref. R257, Exporters. See also: OTLP; Prometheus exporter; Telemetry loss.

Expression evaluation
Debuggers. The interpretation of a language or debugger expression using the target context and available symbols. It reveals values and relationships that are inconvenient to inspect directly. We distinguish reading captured state from calling target functions or changing execution. Such calls can allocate, acquire locks, raise exceptions, or fail when the stopped process cannot progress.

Ref. R030, Section 10.3; R031, Section 17.5. See also: Debugger context; Memory examination; Deadlock.

Extended software trace
Narratology. A software trace considered together with supporting material such as screenshots, video, scenario descriptions, interviews, and other contextual records. These materials can explain what the user attempted and what the trace vocabulary means. We preserve their distinct provenance and timing instead of treating every supporting narrative as an event recorded by the software.

Ref. R001, p. 120. See also: Software narrative; Problem description; Diagnostic artifact.

F

F1 score
AI and ML. The harmonic mean of precision and recall, summarizing a selected balance between them. It excludes true negatives and does not express operational error costs. Multiclass averaging and zero-denominator conventions affect the score. Retain the underlying counts when comparing diagnostic models, evaluated populations, or decision thresholds.

Ref. R234, F-measures; averaging and zero-division behavior. See also: Precision; Recall; Confusion matrix.

Factor Group
Trace patterns. A partition of messages into disjoint groups according to their variable values or a predicate over those values. Comparing the groups can expose alternation, imbalance, or changed state. The name is a mathematical analogy; the groups here need not have the algebraic structure of a quotient group.

Ref. R002, pp. 132–133. See also: Message Invariant; Data Association; Relative Density.

Failure
Foundations. An event in which delivered service deviates from correct service at a stated system boundary. An internal error need not produce failure, while a component's failure can become a fault for a dependent system.

Ref. R003, §§2.1–2.2. See also: Fault; Error; Problem description.

Fake API

Debuggers. A workaround that substitutes behavior at an API boundary through a wrapper, hook, patch, or related implementation technique. The replacement may avoid an unwanted result or emulate an expected response. We specify the substituted contract and the cases it covers. Diagnostic observations made through the replacement describe the modified environment and need that context.

Ref. R848, Pattern discussion. See also: Workaround; Hooksware; Fault injection.

Fake Module

Memory patterns. A module that presents a familiar or legitimate-looking identity while differing from the expected component. We compare its path, size, version information, image structure, and other metadata with a suitable reference. A plausible filename is only one observation. Attribution requires the actual image and its context, including legitimate versions or deployments that may also differ.

Ref. R834, Pattern discussion. See also: Deviant Module; Unknown Component; Module Collection.

False Effective Address

Memory patterns. A debugger computes a misleading operand address from incomplete or invalid saved register values. The displayed address can therefore disagree with the fault address recorded during exception or bugcheck processing. We compare those sources and inspect context warnings before diagnosing a null pointer or other address problem from the disassembly annotation alone.

Ref. R791, Crash Dump Analysis Patterns (Part 205). See also: False Function Parameters; Fault Context; Invalid Exception Information.

False Frame

Memory patterns. An individual stack frame is reconstructed incorrectly even though surrounding frames appear useful. Missing unwind information, an unloaded module, or symbolic ambiguity can produce a plausible but invalid return address. We check the candidate against executable memory, instructions, symbols, and comparable traces before correcting the frame or using it to assign responsibility.

Ref. R566, Pattern description: False Frame. See also: Incorrect Stack Trace; Coincidental Symbolic Information; Injected Symbols.

False Function Parameters

Memory patterns. Values displayed as function parameters that do not reliably represent the arguments supplied at the relevant call. Calling conventions, register reuse, optimization, and stack reconstruction can all affect the interpretation. We verify the exact code and parameter locations before assigning semantic meaning to displayed values. Even a correctly located value may describe the capture moment rather than its original value at function entry.

Ref. R384, Displayed-parameter discussion and calling-convention examples. See also: Calling convention; CPU register; Incorrect Stack Trace.

False Local Address

Memory patterns. A debugger reports a local variable at the wrong address even though the relevant base register is correct. Nonstandard alignment in the ARM64 example exposes the mismatch between the displayed location and the initialization instructions. We reconstruct the actual storage address from disassembly and memory before interpreting the reported value as corruption or failed initialization.

Ref. R576, Pattern description: False Local Address. See also: Address Representations; False Function Parameters; Corrupt Structure.

False Memory

Memory patterns. Memory readable during dump analysis does not necessarily contain the bytes present when the recorded exception occurred. An address released before a fault may be reused during subsequent reporting activity, leaving unexpected replacement contents in the capture. We compare the exception, mappings, and lifecycle evidence. Present readability must not be treated as proof that the original access was valid.

Ref. R688, False Memory pattern and diagnostic examples. See also: Stored Exception; Invalid Pointer (objects); Memory map.

False Positive Dump

Memory patterns. A dump that directs analysis toward an apparent failure created by the collection conditions or an inappropriate capture trigger. An expected exception, for example, can become the most conspicuous event in the artifact. We check how the dump was obtained and whether its exception matches the reported symptom before treating the selected thread or module as the failing component.

Ref. R348, Opening definition and OutputDebugString capture example. See also: Manual Dump; First-chance exception; Early Crash Dump.

False Positive Error

Trace patterns. An error that is present in the artifact but does not explain the particular problem being investigated. A comparable working trace may contain the same error in the same context. This comparison weakens its diagnostic relevance but does not, by itself, prove that the error is harmless in every occurrence.

Ref. R002, p. 134. See also: Master Trace; Error Message; Bifurcation Point.

False sharing

Performance. Interference caused when processors access different data occupying the same coherence-managed cache line and at least one access writes. Coherence traffic can repeatedly invalidate or transfer the line despite the data being logically independent. This is a performance mechanism, not proof of an application data race. Correlate cache-line evidence with data layout and access patterns before attributing a slowdown to it.

Ref. R401, What is False Sharing; How to detect and analyze False Sharing. See also: Cache miss; CPU utilization; Hotspot.

Fat Process Dump

Memory patterns. A process dump is unusually large relative to comparable captures from similar deployments or repeated executions. Extra modules, queued work, or growing memory regions may explain the difference and make the artifact a useful starting point. We compare acquisition scope and workload before interpreting size; a larger file alone does not identify a memory leak or a failure mechanism.

Ref. R636, Fat Process Dump pattern and diagnostic examples. See also: Memory dump; File Size; Memory Leak (regions).

Fault

Foundations. A cause, observed or hypothesized, of an error. It may remain dormant until activated. Distinguish the cause from the resulting incorrect state and service deviation at the system boundary being investigated.

Ref. R003, §2.2. See also: Error; Failure; Defect mechanism.

Fault Context

Memory patterns. Different failures repeatedly occur within a common execution setting, such as the same process, despite varying instructions, modules, or apparent symptoms. We use that shared context to direct investigation toward related workloads, interactions, and resources. The recurring setting connects the cases and helps formulate a hypothesis without establishing that the component providing the context caused every fault.

Ref. R752, Crash Dump Analysis Patterns (Part 97). See also: Crash Signature; Data Contents Locality; Dump Context.

faulthandler

Runtimes. A Python module for emitting tracebacks when selected faults occur, when explicitly requested, or after a configured timeout. It can preserve useful thread context when normal exception reporting is unavailable. Its emergency output intentionally uses restricted formatting and does not reconstruct all native or interpreter state. A timeout-triggered traceback records where execution was observed, not proof that the displayed function caused the delay.

Ref. R218, Fault handlers; delayed traceback dumps; output limitations. See also: pdb [Runtimes]; Thread dump; asyncio debug mode.

Fault injection

Testing. Deliberate introduction of a selected fault condition to observe how software detects, contains, and recovers from it. An allocation failure, rejected operation, or unavailable dependency can exercise paths rarely reached by ordinary tests. The experiment needs a defined injection point and observable outcome. An artificial fault may omit timing or state interactions present in the naturally occurring problem.

Ref. R169, Fault-injection capabilities and controls. See also: Chaos engineering; Test harness; Test oracle.

Fault localization

Testing. Narrowing the search for faulty program elements using evidence from executions, tests, or analysis. A localization technique may rank statements rather than identify one definitive cause. The ranking helps allocate investigation effort; it needs interpretation against control flow, data dependencies, and the failure criterion. Code frequently executed by failing tests can be associated with a failure without containing its defect.

Ref. R156, Test information for locating faulty statements. See also: Spectrum-based fault localization; Statistical debugging; Program slicing.

Feature extraction

AI and ML. Transformation of raw observations into features suitable for a model. Parsing text, encoding categories, and computing numerical summaries each define a particular view of diagnostic evidence. The transformation can discard order, context, or rare details that distinguish mechanisms. Its version, fitted parameters, and handling of missing or unseen inputs should accompany model results so the representation can be reconstructed.

Ref. R236, Text and dictionary feature extraction. See also: Feature vector; Training-serving skew; Data leakage.

Feature of Activity

Trace patterns. Messages grouped by a selected feature identifier, including a feature derived during analysis. The grouping can span several execution or adjoint threads. It provides a feature-based perspective that differs from a fiber restricted to one activity, and its usefulness depends on the meaning of the selected feature.

Ref. R002, p. 135. See also: Fiber of Activity; Adjoint Thread of Activity; Trace Schema.

Feature vector

AI and ML. A numerical representation of an observation used by a model or similarity method. For software diagnostics, its coordinates might encode counts, categories, or text-derived properties. The vector makes selected aspects of an artifact comparable while omitting others. Two cases close in feature space need not share a mechanism; the representation determines which differences remain visible to the analysis.

Ref. R236, Feature extraction overview; dictionaries and text. See also: Feature extraction; Embedding; Clustering.

Feynman Trace

Trace patterns. A conceptual collection of feasible trace variants from possible executions, including empty traces, error-producing routes, and alternative activity paths. The name borrows the path-integral idea from quantum mechanics. Declarative Trace analysis can help identify contributions, while Renormalization can simplify repeated structures. We exclude Impossible Traces and avoid treating the analogy as a ready-made probability calculation over executions.

Ref. R440, Part 245. See also: Trace Amplitude; Impossible Trace; Declarative Trace; Renormalization.

Fiber

Execution. An application-scheduled execution unit that runs in the context of an operating-system thread. A thread can switch among fibers, so a thread identifier alone does not fully identify the logical execution being investigated. We preserve stack and scheduling context when following fiber-based activity. A debugger view centered on the current thread may expose only the fiber that is currently active.

Ref. R133, Scheduling and saved fiber state. See also: Thread; Stack; Debugger context.

Fiber Bundle

Trace patterns. A base trace with supplementary traces attached to individual messages, such as recorded call stacks or input/output stack information. The bundle lets us move from an event to its associated execution context. Fiber-bundle terminology expresses the structural analogy without asserting a particular mathematical bundle construction.

Ref. R002, pp. 136–137. See also: Exception Stack Trace [Trace patterns]; Trace Extension; Trace Presheaf.

Fiber of Activity

Trace patterns. A selection of one message type within a particular thread or adjoint thread of activity. This narrower view supports examination of recurrence, intervals, and data changes. The pattern names an analytical selection and should be distinguished from a runtime fiber used as a scheduling abstraction.

Ref. R002, p. 138. See also: Feature of Activity; Fourier Activity; Thread of Activity.

File descriptor

Execution. A process-local integer identifying an entry through which a Unix-like process accesses an open resource. The descriptor and the underlying open file description are different objects. We preserve process identity and lifetime when following it through logs. A repeated descriptor number can refer to a newly opened resource, so numerical equality alone does not establish continuity of one file or connection.

Ref. R061, DESCRIPTION (file descriptor and open file description); NOTES, Open file descriptions. See also: Process; System call [Execution]; strace.

File Size

Trace patterns. The size of a trace artifact used as an initial clue to its capture and storage history. Repeated identical limits may suggest circular recording or truncation. Size must be interpreted with file format, compression, preallocation, and recording settings; it is not a direct measure of execution duration or diagnostic value.

Ref. R002, p. 139. See also: Circular Trace; Truncated Trace; Empty Trace.

Finalization

Runtimes. Runtime-triggered cleanup associated with an object's lifetime, traditionally used after ordinary program use has ended. Its scheduling and execution guarantees differ from explicit resource release, making it a poor basis for timely cleanup. Queued or slow finalizers can contribute to retention and delayed resource recovery. Java has deprecated finalization for removal; that status should not be confused with universal removal from every runtime.

Ref. R217, Summary; motivation and finalization weaknesses. See also: Garbage collection; Weak reference; GC root.

First-chance exception

Execution. The initial debugger notification of a Windows exception, before the system completes its search for an application handler. The exception may subsequently be handled as part of normal execution. We use this earlier observation point to preserve state that later handling might alter. Its presence in a trace does not establish an unhandled failure or justify treating every occurrence as a separate crash.

Ref. R096, First-chance and last-chance notifications. See also: Exception; Second-chance exception; Breakpoint.

First fault

Foundations. The earliest relevant manifestation of a problem in the evidence available to an investigation. An initial anomaly may be handled or obscured before a later, more visible failure occurs. Preserving and following the earlier indicators can prevent the final symptom from dominating the analysis; the earliest recorded event may still follow an unrecorded initiating defect.

Ref. R001, p. 156; pp. 239–241. See also: Fault; Defect mechanism; Causal History.

First Fault Stack Trace

Memory patterns. The stack trace associated with the earliest recoverable fault in a sequence of subsequent exceptions or reporting failures. We examine nested contexts, hidden exceptions, and surviving execution residue to separate the initiating failure from later consequences. Recovery may be partial, and the earliest trace available in the artifact is not automatically the earliest fault that occurred.

Ref. R537, Pattern description: First Fault Stack Trace. See also: Hidden Exception; Nested Exceptions (unmanaged code); Problem Exception Handler.

Flag

Trace patterns. A marked message or activity region, or a nested sequence of message sets and regions organized by inclusion. The latter use borrows the mathematical idea of a flag. Both forms help organize attention and progressively narrowed views while preserving which larger context contains each selection.

Ref. R002, p. 140. See also: Message Annotations; Message Set; Activity Region.

Flaky test

Testing. A test that produces different outcomes without an intended relevant change to the tested behavior. Hidden state, timing, ordering, concurrency, and environmental dependencies can make its result unstable. Repetition establishes the instability but does not explain it. Diagnostic work should identify which uncontrolled condition separates passing and failing executions before using the test as evidence for a code-change boundary.

Ref. R167, Sources of flaky tests. See also: Hermetic test; Regression bisection; Determinism.

Flame chart

Performance. A stack-oriented visualization whose horizontal axis represents time, preserving the order of recorded activity. It helps reveal phases, bursts, gaps, and nested execution that aggregation can conceal. Unlike a flame graph, identical stacks need not be merged across the interval. Its detail remains limited by the underlying samples or events; the drawing does not recover operations that were never recorded.

Ref. R394, Variations: Flame charts. See also: Flame graph; Elapsed time; Sampling profiler.

Flame graph

Performance. A visualization of aggregated stack information in which rectangle width represents the associated weight and vertical position represents stack depth. The horizontal axis groups the profile population rather than showing chronological execution. A wide frame can include work in descendants, and color need not encode cost. Read the weight unit and attribution rule before treating width as elapsed time.

Ref. R394, Summary. See also: Flame chart; CPU profile; Inclusive time.

Focus of Tracing

Trace patterns. The semantic subject of the currently visible tracing activity, such as session initialization followed by database work. A focus can extend across several dense activity regions. Recognizing changes of focus helps interpret the trace in terms of functionality instead of treating every increase in message rate as the same activity.

Ref. R002, p. 141. See also: Activity Region; Use Case Trail; Background and Foreground Components.

Fooled by Abbreviation

Foundations. Inferring a component's identity or a symbol's meaning from a familiar-looking abbreviation without checking its actual context. The same prefix can denote unrelated products or structures. We verify the module, symbol information, and surrounding evidence before assigning a meaning. A plausible expansion is a hypothesis about the name, not evidence about the component's behavior.

Ref. R870, Antipattern discussion. See also: Unknown Component; Symbol file; No Component Symbols.

Fooled by Description

Foundations. Allowing the reported problem description to determine the interpretation of an artifact despite evidence pointing elsewhere. We use the report to frame questions, then reconcile it with the observed state, timing, and capture context. A discrepancy may reveal incomplete reporting, a changed problem, or an unsuitable artifact; it should remain visible rather than be explained away.

Ref. R867, Antipattern discussion. See also: Problem description; Diagnostic evidence; Diagnostic hypothesis.

Foreign Module Frame

Memory patterns. A stack frame belongs to a module supplied by a vendor other than the operating-system or main application vendor. We inspect its identity, path, and loading role to determine whether a plugin or hook participates in the failing activity. Its presence can reveal an unexpected remaining integration, but third-party origin alone does not make the frame culpable.

Ref. R558, Pattern description: Foreign Module Frame. See also: Hooked Functions; Problem Module; Module Product Process.

Foreign Stack

Memory patterns. A value in one thread's stack points into another thread's stack region. Such a reference can expose relationships between threads and may warrant scrutiny of local-variable lifetime in asynchronous work. We establish the target region and the referenced object's use before judging correctness. A cross-stack pointer is an investigative clue; it is not automatically evidence of an invalid reference.

Ref. R696, Foreign Stack pattern and diagnostic examples. See also: Stack; Use-after-free; Value References.

Fourier Activity

Trace patterns. Analysis of the recurrence frequency of selected message types and changes in their intervals. Periods of slower recurrence may expose delays or additional work hidden inside an aggregate performance symptom. The name is inspired by Fourier analysis; recognizing the pattern does not require computing a Fourier transform.

Ref. R002, pp. 142–143. See also: Fiber of Activity; Time Delta; Delay Dynamics.

FPU Exception

Memory patterns. A floating-point operation raises an exception that must be interpreted using the captured floating-point state as well as the ordinary instruction context. We inspect the exception code, faulting operation, and relevant FPU registers to identify the condition. The visible failure may occur inside runtime-generated or compiler infrastructure, so its module alone does not explain the arithmetic state.

Ref. R528, Pattern description: FPU Exception. See also: Invalid Exception Information; Divide by Zero (kernel mode); JIT Code (.NET).

Frame pointer

Memory. A register used by some calling conventions or compiler-generated code to locate a function frame consistently while the stack pointer changes. Compilers may omit it or use the register for another purpose. We therefore check the generated code and unwind information before following a presumed frame chain. A missing frame pointer does not by itself make a stack unrecoverable.

Ref. R130, Stack allocation; Function types; alloca. See also: Stack pointer; Stack unwinding; Calling convention.

Frame Pointer Omission

Memory patterns. A compiled-code pattern in which functions do not maintain the conventional frame-pointer chain. Apparent disagreement between frame and stack pointers can therefore reflect optimization rather than corruption, particularly in the illustrated x86 setting. We use the applicable calling convention, unwind information, and instructions to reconstruct frames. Assuming that every function preserves a frame pointer can turn a valid stack into a misleading trace.

Ref. R383, Opening frame-pointer discussion and optimized x86 code example. See also: Frame pointer; Stack unwinding; Incorrect Stack Trace.

Frame Regularity

Memory patterns. Similar invocations leave recurring arrangements of values in their stack frames or execution residue. We compare an unclear frame with another whose relevant value is already understood, using the regularity to suggest an interpretation. The proposed correspondence must be validated against the object and surrounding code; a repeated stack position is not automatically a genuine function argument.

Ref. R806, Crash Dump Analysis Patterns (Part 251). See also: False Function Parameters; Data Correlation (function parameters); Execution Residue.

Frame Trace

Memory patterns. A traversal of stack frames that collects their parameters, local variables, and associated frame information. We examine the resulting sequence to recover values across the calling path and correlate them with logs or other incident evidence. It extends a simple function-name stack view, subject to available symbols, optimization, captured memory, and the validity of each frame's context.

Ref. R811, Crash Dump Analysis Patterns (Part 267). See also: Managed Stack Trace; Historical Information; Annotated Disassembly (unmanaged code).

Frozen Process

Memory patterns. A process cannot make normal progress because its threads are frozen or suspended by a platform, debugger, or lifecycle mechanism. We inspect thread and process state before interpreting stationary stacks as a synchronization failure. Some mechanisms may leave a limited thread active. Establishing how execution was restricted is necessary before diagnosing the application's own progress behavior.

Ref. R661, Frozen Process pattern and diagnostic examples. See also: Suspended Thread; Hang; Debugger context.

Frozen Process

Debuggers. A workaround that suspends a process responsible for unwanted activity. The process remains present while its execution is interrupted, which can also affect components waiting for it. We distinguish this deliberate intervention from a process found frozen during analysis and record when suspension began. Stopping the activity does not establish that its underlying mechanism has been corrected.

Ref. R846, Pattern discussion. See also: Suspended Thread; Workaround; Unresponsive Window.

ftrace

Native tools. Linux kernel tracing infrastructure with interfaces for function tracing and other supported tracing facilities. Its trace buffers and controls expose selected kernel execution to subsequent analysis. We choose the tracer, filters, buffer configuration, and timestamps before interpreting a record. The name does not imply that every function or event is present, and collection overhead or overwritten buffer contents can constrain the reconstructed sequence.

Ref. R038, Introduction; The File System. See also: Kernel tracepoint; Kprobe; perf.

Full user-mode dump

Artifacts. A Windows process dump intended to preserve broad user-mode state for reconstruction. The legacy full-dump format and a minidump containing full-memory options are not identical, and the latter can preserve additional information. We identify the format and collection flags rather than infer completeness from the filename. Unreadable pages or unavailable external resources can still limit the resulting analysis.

Ref. R064, Types of user-mode dump files; Full user-mode dumps; Minidumps. See also: Minidump; Memory dump; Process.

Function Pointer

Memory patterns. A code address found in raw memory may designate a function entry or exception-handling entry rather than a return address. We examine forward disassembly and compare surrounding instructions to classify the reference before reconstructing a past call path. Missing symbols can display a real entry as another symbol plus an offset, so the printed name alone is insufficient.

Ref. R821, Crash Dump Analysis Patterns (Part 297, Linux). See also: Execution Residue; Past Stack Trace; Coincidental Symbolic Information.

Futex

Execution. A Linux synchronization facility that lets user-space code coordinate waiting and waking around a shared word. Uncontended synchronization can remain in user space, with kernel assistance when waiting is needed. We interpret a futex wait using the higher-level lock or condition protocol. The system-call name identifies the waiting mechanism, but does not by itself identify the owner or prove a deadlock.

Ref. R059, DESCRIPTION. See also: Mutex; Condition variable; strace.

Fuzzing
Testing. Repeated automated execution with generated or mutated inputs to expose unexpected behavior. A campaign needs an input interface, an execution mechanism, and a way to recognize interesting outcomes. Crashes, sanitizer reports, hangs, and violated properties are different findings and need separate interpretation. The fuzzer's reach depends on the harness, seed corpus, input model, and resources assigned to the search.

Ref. R161, Fuzzing usage and campaign documentation. See also: Coverage-guided fuzzing; Test harness; AFL++.

G

Galois Trace
Trace patterns. A correlated investigation of two traces in which movement through one is paired with movement in the opposite direction through the other. The pattern uses a Galois-connection analogy to organize this search. A formal Galois connection would additionally require specified orders and verified mathematical properties.

Ref. R002, p. 144. See also: Inter-Correlation; Back Trace; Message Context.

Garbage collection
Runtimes. Automatic reclamation of storage for objects that the runtime determines are no longer needed under its reachability rules. It changes the relationship between application allocation and actual memory release. Unwanted references can keep objects alive, so automatic collection does not eliminate retention problems. Diagnostic analysis separates allocation, live data, collection work, and the memory the runtime reserves from the operating system.

Ref. R191, Memory allocation; collection; generations. See also: Managed heap; GC root; Retained size.

Gauge
Metrics. A metric whose reported value can increase or decrease, such as queue depth or current memory use. A gauge usually describes a state at an observation time rather than an accumulated event count. Short-lived peaks between observations may be absent. Summing gauges is meaningful only when their units, populations, and time alignment permit the resulting quantity to be interpreted.

Ref. R270, Gauge. See also: Counter; Scrape interval; Queue length.

GC root Garbage collection root
Runtimes. A starting point from which a garbage collector or heap-analysis tool determines object reachability. Roots may arise from thread state, static storage, or native handles, depending on the runtime and artifact. A path from a root explains why an object remains reachable. Tool-created roots and incomplete root information need recognition before the path is interpreted as an application ownership relationship.

Ref. R206, Root categories; unknown and synthetic roots. See also: Garbage collection; Dominator tree; Heap dump.

GDB GNU Debugger

Debuggers. The GNU debugger, which supports inspection and execution control for supported programs, core files, and remote targets. Commands such as backtrace, print, and x construct different views of execution and data. We establish the executable, debug information, target architecture, and selected thread before interpreting those views. Supported languages and target facilities depend on how GDB and the target were built.

Ref. R112, Summary. See also: Core dump; Memory examination; Remote debugging.

General analysis pattern

Foundations. An analysis method expressed independently of a particular product, platform, or command language. It describes the diagnostic operation and its purpose at a level that can be realized in several environments. Porting it requires a concrete mapping of artifacts, fields, and tool behavior; superficial similarity of commands does not establish equivalent evidence.

Ref. R001, pp. 216–218. See also: Concrete analysis pattern; ADC; Pattern repertoire.

Generalized software trace

Narratology. A representation that extends the trace concept beyond a chronological log to data arranged by another index or relationship, including memory contents. This permits analysis patterns to move between artifact types. The chosen mapping determines which order, adjacency, and identity relationships are preserved and which claims the resulting trace can support.

Ref. R001, p. 149; pp. 206–208. See also: General trace; Memory dump analysis; Software narratology.

General problem pattern

Foundations. A recurrent diagnostic phenomenon described without the details of one platform or product. It captures recognizable relationships among indicators that can have several concrete realizations. The general description supports transfer of knowledge, while a real investigation must still identify the specific evidence and contextual conditions that make the pattern applicable.

Ref. R001, pp. 216–218. See also: Concrete problem pattern; Problem pattern; Structural pattern.

General trace

Narratology. An artifact treated as a sequence or structured collection of messages even when ordinary timestamp order is absent, partial, or inappropriate. Debugger output and memory-derived records can be examined this way. We choose an ordering or relationship suitable for the artifact rather than importing causal assumptions from a conventional time-stamped execution trace.

Ref. R001, pp. 206–208; pp. 314–315. See also: Special trace; Generalized software trace; Software codiagnostics.

Generative debugging

Foundations. A proposed approach to constructing models of defects by combining behavioral forms with structural parameters. A leak, overflow, or blocked execution can be modeled in a selected thread, heap, process, or other context. The resulting model helps investigate diagnostic indicators and mechanisms; the term here predates and does not specifically mean generative-AI-assisted debugging.

Ref. R001, pp. 130–131. See also: MdTL; Defect mechanism; Projective debugging.

Generative Trace

Trace patterns. Source code considered as a general trace capable of generating different execution traces. Logging statements form its Declarative Trace, while line order supplies a possible presentation order. Source order is not execution order: branches, calls, repetition, and concurrency determine the actual histories produced.

Ref. R002, p. 145. See also: Declarative Trace; Moduli Trace; Linked Messages.

Ghost Thread

Memory patterns. A synchronization structure refers to a thread identifier whose original thread has exited and whose value now belongs to a different thread, possibly in another process. The apparent owner therefore misdirects a wait-chain analysis. We compare process membership, creation history, and available traces to distinguish identifier reuse from a live dependency on the newly created thread.

Ref. R548, Pattern description: Ghost Thread. See also: Missing Thread; Wait Chain; Thread Age.

Glued Activity

Trace patterns. An apparent single activity formed by joining records from different execution entities that reuse an identifier or name. Reused process identifiers are a common example. We split the view using creation, termination, lifetime, or other identity evidence before interpreting the combined messages as one continuous activity.

Ref. R002, pp. 146–147. See also: Intrinsic ID; Identification Messages; Adjoint Thread of Activity.

Glued Stack Trace

Memory patterns. A stack trace is reconstructed by joining surviving or recovered fragments from one stack region. The assembled path can help interpret a truncated trace, but intermediate frames may remain missing. We record where each fragment came from and mark the joins and gaps. A useful reconstruction should not acquire the appearance of a complete, directly observed call stack.

Ref. R665, Glued Stack Trace pattern and diagnostic examples. See also: Truncated Stack Trace; Past Stack Trace; Stack unwinding.

Go execution trace

Runtimes. A recording of selected Go runtime events, including goroutine activity, scheduling, blocking, system calls, and garbage collection. Application tasks and regions can add structure to the runtime history. The trace helps relate a delay to execution and scheduling intervals. It differs from an aggregate CPU profile, and its usefulness depends on collection duration, event completeness, and the annotations available.

Ref. R231, Runtime events; tasks, regions, and logs. See also: pprof; Goroutine profile; Stop-the-world pause.

Golden signals

Reliability. Four monitoring dimensions described in Google SRE practice: latency, traffic, errors, and saturation. They connect user-visible behavior with demand and constrained resources. Each needs an operational definition. We preserve distinctions such as successful versus failed requests and healthy versus overloaded instances; these dimensions guide investigation without exhausting every useful metric.

Ref. R285, The Four Golden Signals. See also: RED method; Saturation; Latency.

Go race detector

Program analysis. Go's runtime race-detection facility, enabled for supported builds and tests with race instrumentation. Reports identify conflicting accesses and relevant execution context from the observed workload. The detector can find races only in behavior that actually executes, so realistic test coverage matters. A clean instrumented run does not establish absence of every race, deadlock, or other concurrency problem.

Ref. R229, Usage; report format; how to use. See also: ThreadSanitizer; Go execution trace; Code coverage.

Goroutine profile

Runtimes. A Go runtime profile describing the stack contexts of current goroutines. It helps recognize repeated blocked contexts, unexpected populations, or work accumulating behind a common operation. A large count alone does not establish a leak: the expected concurrency and workload matter. Comparing profiles over time can reveal persistent growth, but individual stack snapshots do not provide the full scheduling history.

Ref. R230, goroutine profile. See also: pprof; Go execution trace; Thread dump.

Gossip

Trace patterns. Repeated reporting of the same underlying information by different sources, possibly with changes of wording, representation, or meaning. Several messages may therefore describe one event rather than several events. Comparing the accounts helps detect distortion and avoids constructing a false event sequence by mixing duplicate reports.

Ref. R002, p. 148. See also: Diegetic Messages; Data Association; Event Sequence Order.

Graceful degradation

Distributed systems. Continuing to provide reduced or less costly service when resources or dependencies are constrained, for example by omitting optional work or using an approximation. A successful response can therefore represent degraded behavior. We examine activation, reduced guarantees, and recovery conditions before treating the response as the normal service contract.

Ref. R337, Graceful degradation. See also: Load shedding; SLI; Cascading failure.

Guard page

Memory. A page used to detect access at a boundary or otherwise trigger special handling. Windows PAGE_GUARD provides a one-shot access alarm whose protection modifier is cleared after the exception; an inaccessible PageHeap boundary uses different protection behavior. We identify the mechanism before interpreting the fault. A guard-page exception can be part of expected stack growth rather than evidence of corruption.

Ref. R092, PAGE_GUARD behavior. See also: PageHeap; Stack overflow; Memory protection.

Guest Component

Trace patterns. An unexpected component visible in a trace, often through a module-load event, whose presence distinguishes the investigated environment. It can reveal configuration, integration, or injected functionality worth examining. The component is a guest relative to the expected configuration; unexpected presence alone does not establish faulty or malicious behavior.

Ref. R002, pp. 149–150. See also: Missing Component; Master Trace; Background and Foreground Components.

H

Habitual Reply

Foundations. Recommending a familiar troubleshooting action without checking whether its detection mechanism or expected effect matches the current evidence. Repetition from earlier cases can make the recommendation appear justified when the conditions differ. We connect each proposed action to a specific question and predicted observation, then assess the result against that purpose.

Ref. R875, Antipattern discussion. See also: Troubleshooting; Diagnostic hypothesis; Fault injection.

Hallucination

AI and ML. A generated statement presented as factual although it is unsupported or incorrect in the relevant context. In debugging, examples include invented commands, nonexistent APIs, or an asserted mechanism absent from the evidence. Comparing multiple generated answers can reveal instability, but consistent answers can share the same error. Verification must return to executable checks, authentic documentation, and the actual diagnostic artifacts.

Ref. R247, Factual inconsistency detection through sampling. See also: RAG; LLM-as-a-judge; LLM-assisted debugging.

Handled Exception (kernel space)

Memory patterns. Kernel execution has passed through an exception handler, leaving recognizable handler activity or references in captured stack memory. We examine the surrounding residue and validate candidate return addresses to distinguish real handling from coincidental symbols. This can establish earlier exception processing even when the current stack does not display the event; handling alone does not show that the underlying condition was corrected.

Ref. R773, Crash Dump Analysis Patterns (Part 152c). See also: Hidden Exception (kernel space); Execution Residue; Coincidental Symbolic Information.

Handled Exception (.NET CLR)

Memory patterns. A managed exception has already been handled, but evidence of its processing remains in stack memory. Runtime handler frames and related residue can reveal that earlier activity beyond the current stack trace. We validate the relationships between candidate frames and exception data. Residue of a handled exception establishes past processing, not automatically the cause of the present problem.

Ref. R650, Handled Exception (.NET CLR) pattern and diagnostic examples. See also: Execution Residue (managed space); Hidden Exception (managed space); Handled Exception (user space).

Handled Exception (user space)

Memory patterns. Evidence that an exception was processed and execution continued, even when no current exception record remains. Raw stack memory may retain dispatcher, handler, unwind, or catch-related calls. We compare this residue with the surrounding execution context; subsequent calls can overwrite it, so absence of such traces does not establish that no exception was handled.

Ref. R526, Pattern description: Handled Exception (user space). See also: Hidden Exception; Execution Residue; Custom Exception Handler.

Handle Leak

Memory patterns. Handles or their associated objects accumulate because releases fail to balance acquisitions over the relevant workload. High process handle counts and an abnormal difference between object allocations and frees can direct the investigation even when the memory footprint is modest. We identify the object types and owning processes, then distinguish unintended retention from legitimate outstanding work.

Ref. R539, Pattern description: Handle Leak. See also: Reference Leak; Insufficient Memory (committed memory); Object Distribution Anomaly.

Handle Limit (GDI, kernel space)

Memory patterns. GDI resource creation reaches a limit associated with session graphics objects and their supporting allocations. We correlate failed creation requests, visual malfunction, and the relevant session-pool populations. The historical Windows examples distinguish handle exhaustion from general memory pressure; a later null-pointer failure may be a consequence of an earlier unsatisfied resource request.

Ref. R489, Pattern description: Handle Limit (GDI, kernel space). See also: Handle Limit (GDI, user space); Insufficient Memory (session pool); NULL Pointer (data).

Handle Limit (GDI, user space)

Memory patterns. A process approaches or reaches its GDI-object quota, with symptoms such as drawing failures, unresponsive dialogs, or allocation errors. In a user-space dump, the shared GDI handle table can help identify counts and object types associated with the process. We use the captured layout and platform context rather than assuming the historical example's structure offsets or quota are universal.

Ref. R613, Handle Limit (GDI, user space) pattern and diagnostic examples. See also: Handle Leak; Resource exhaustion; Main Thread.

Hang

Execution. A perceived absence of visual or functional response from a program, service, or system that remains present. We compare required behavior with thread states, dependencies, and repeated observations. Deadlock, livelock, resource exhaustion, and external waits are possible mechanisms; an unresponsive interface alone does not distinguish them.

Ref. R001, p. 38, Crashes and Hangs Differentiated; R424, Definition. See also: Crash; Deadlock; Livelock [Execution]; Blocked Thread.

Happens-before

Execution. A partial ordering used to reason about memory visibility and conflicting accesses. In Go, it is formed by taking the transitive closure of the sequenced-before and synchronized-before relations. We reconstruct these edges from language and library guarantees. Seeing one operation finish earlier in a trace does not, by itself, establish a happens-before relationship between that operation and another thread's access.

Ref. R405, Memory Model: synchronized-before and happens-before relations; Synchronization. See also: Data race; Memory barrier; Atomic operation.

Hard lockup

Execution. A Linux watchdog condition indicating an extended absence of expected interrupt-related progress on a CPU, as detected by the configured hard-lockup mechanism. It can accompany severe execution stalls or interrupts remaining disabled. We connect the warning to processor state and system context rather than infer a specific defect from the label. Detector implementation, hardware support, configuration, and virtualization can affect the available evidence.

Ref. R084, Configuration; Implementation. See also: Soft lockup; Kernel panic; CPU register.

Hardware Activity

Memory patterns. Captured stack contents retain evidence of driver activity associated with hardware operations, even when current interrupt or DPC state is inconclusive. We inspect these references and check for coincidental symbolic matches before using them as investigative leads. Historical device-related activity identifies a possible route for further analysis, without proving that the device caused the reported symptom.

Ref. R507, Pattern description: Hardware Activity. See also: Execution Residue; Coincidental Symbolic Information; Spiking Interrupts.

Hardware-assisted AddressSanitizer

Program analysis. A memory-error detector that uses tagged pointers and tagged memory to check whether an access belongs to the expected allocation. Tag mismatches expose many out-of-bounds and lifetime errors with a different overhead tradeoff from AddressSanitizer. Detection depends on tag granularity and available tag values, so some invalid accesses can escape a check. Platform and compiler support constrain its use.

Ref. R181, Algorithm; memory tags and tag checks. See also: AddressSanitizer; MemorySanitizer; Dynamic analysis.

Hardware breakpoint

Debuggers. A breakpoint implemented using facilities provided by the processor or target rather than by replacing an instruction in memory. It is useful where modifying code is undesirable or impossible. We account for the target's limited debug resources and supported conditions. Hardware assistance does not make all addresses, access widths, or execution contexts equally observable, and the debugger may reject requests that exceed those resources.

Ref. R114, Section 5.1.1; hardware-assisted breakpoints. See also: Breakpoint; Watchpoint; CPU register.

Hardware Error

Memory patterns. Captured failure information points toward a hardware-detected condition, such as a machine-check report. We examine the actual diagnostic record and correlate it with the observed execution. A surprising instruction, corrupted address, or generic fault is insufficient to identify hardware as the cause; software changes and corruption can produce similar symptoms, and an NMI may simply have triggered dump collection.

Ref. R734, Crash Dump Analysis Patterns (Part 57). See also: Self-Diagnosis (kernel mode); Lateral Damage; Dump Context.

Head sampling

Telemetry. A tracing decision made near the beginning of a trace, before its completed behavior is known. It can keep a predictable fraction of requests with low decision overhead, but cannot directly select a request because of a later failure or long duration. Interpretation must account for the policy and its propagation; downstream collection cannot reconstruct spans that were never recorded.

Ref. R258, Head sampling. See also: Sampling; Tail sampling; Context propagation.

Health check

Cloud. A defined test used to decide whether an application or dependency satisfies a particular operating condition. Its result is meaningful only within the scope of what it exercises and the success criteria it applies. A shallow endpoint may respond while business operations fail; an overly broad check may propagate a dependency outage. Identify the check's consumer and the action triggered by failure.

Ref. R327, Solution; issues and considerations. See also: Black-box monitoring; Liveness probe; Readiness probe.

!heap

Debuggers. A Windows debugger extension command for examining supported user-mode heap state, with options for statistics, blocks, and corruption investigation. Its capabilities depend on the heap implementation and debugging environment. We relate reported blocks to allocation lifetime and ownership. A heap summary can locate growth or damaged metadata, but a single snapshot does not identify the allocation history or prove that retained blocks are leaks.

Ref. R108, Segment and NT Heap Parameters. See also: Heap; Heap corruption; Memory leak.

Heap

Memory. A memory region managed for dynamic allocations whose lifetimes need not follow function calls. Heap metadata describes allocation boundaries and state according to a particular allocator. During analysis, we reconstruct these boundaries before assigning meaning to nearby bytes. A damaged heap header can be detected during a later allocation or release, far from the operation that originally overwrote it.

Ref. R014, Private heaps and fragmentation. See also: Heap corruption; Memory fragmentation; Memory leak.

Heap corruption

Memory. Damage to data or metadata belonging to dynamically allocated storage. The allocator may discover it while checking a block, linking a free list, or releasing memory. Such detection identifies an inconsistent heap state but does not automatically identify the original writer. We inspect block boundaries and neighboring allocations, then seek earlier evidence through PageHeap, sanitizers, or recorded execution.

Ref. R014, Private heaps and fragmentation; R068, Standard and full-page heap verification; R108, Segment and NT Heap Parameters. See also: Heap; Memory corruption; PageHeap.

Heap dump

Runtimes. An artifact recording managed objects and relationships at a selected collection point, with details determined by the runtime, format, and capture options. It supports analysis of types, object graphs, roots, and retention. It usually lacks the complete history of allocation and ownership changes. Capture may request collection or affect the process, so the artifact's timing and acquisition method belong in its interpretation.

Ref. R208, GC.heap_dump. See also: HPROF; Heap histogram; Dominator tree.

Heap histogram

Runtimes. A summary grouping heap objects by type or another category, commonly with instance counts and aggregate sizes. Comparing histograms can reveal which categories grow between observations. The summary does not identify all individual retaining paths or distinguish intended growth from a leak. Collection options, class identity, and whether the counted population includes unreachable objects affect the comparison.

Ref. R208, GC.class_histogram. See also: Heap dump; Shallow size; Retained size.

Heap snapshot

Runtimes. A point-in-time representation of objects and references captured by a runtime or heap-analysis interface. In JavaScript tooling it supports investigation of object populations, retaining paths, and detached structures. Creating a snapshot can suspend execution and consume substantial additional memory. A comparison shows differences between captured states, while allocation history and the sequence that produced those differences may require separate recording.

Ref. R224, How to obtain and interpret heap snapshots; capture cost. See also: Heap dump; Retained size; Allocation profiling.

Hedges

Trace patterns. Event sequences whose observed lengths vary between comparable occurrences. Variation can suggest that an external condition interrupts the sequence, rather than that one fixed internal stage always fails. We compare the sequence boundaries and timing, for example when a timeout permits different numbers of events in different runs.

Ref. R002, pp. 151–152. See also: Event Sequence Order; Time Delta; Timeout [Trace patterns].

Heisenbug

Debuggers. A defect whose manifestation changes or disappears when we attempt to observe or reproduce it. Debugging can alter timing or other conditions on which the failure depends. We treat the changed behavior as evidence about those dependencies and compare acquisition methods carefully. Failure to reproduce under a debugger does not establish that the original defect has been removed.

Ref. R404, Software faults are soft—the Bohrbug/Heisenbug hypothesis, printed pp. 11–12. See also: Bohrbug; Observer effect; Reproducer; Race condition.

Helgrind

Program analysis. A Valgrind tool for investigating threading errors, including recognized synchronization misuse, inconsistent lock ordering, and data races. It relates operations observed during an execution and reports potentially problematic relationships. Understanding custom synchronization and suppressed reports is necessary before drawing conclusions. A run that exposes no report does not establish that every possible schedule or concurrency property has been checked.

Ref. R186, Detected error classes and synchronization handling. See also: Valgrind; ThreadSanitizer; DRD.

Hermetic test

Testing. A test whose relevant inputs and dependencies are explicitly supplied or controlled, reducing reliance on ambient machine state and external services. Hermeticity supports repeatability and makes environmental differences easier to reason about. It is a property of the test boundary, not a claim that the simulated environment represents production completely. The boundary should preserve the interaction under investigation.

Ref. R168, Purpose of tests; hermeticity and declared dependencies. See also: Reproducer; Flaky test; Test double.

Hidden Call

Memory patterns. A call is missing from the displayed stack because optimization or uncertain reconstruction omitted its frame. We inspect the instructions associated with surviving return addresses and compare native and managed views to recover a plausible intervening operation. The additional call must be supported by code and surrounding context, rather than inserted solely because the stack appears unfamiliar.

Ref. R796, Crash Dump Analysis Patterns (Part 220). See also: Incorrect Stack Trace; Frame Pointer Omission; Managed Stack Trace.

Hidden Error

Trace patterns. An error inferred from indirect signs when no explicit error message is present. Loading an error-reporting component can be one such sign. The sign suggests a further search or a different acquisition strategy; its meaning must be checked because the same component may also be used during normal execution.

Ref. R002, p. 153. See also: Indirect Message; Error Message; Adjoint Space.

Hidden Exception

Memory patterns. An exception that is not apparent in the initially selected debugger context or automated analysis, but whose evidence remains elsewhere in the dump. Saved exception records, dispatcher frames, and raw stack contents may reveal an earlier event. We establish that the recovered context belongs to the relevant thread and episode; exception residue can describe a handled event rather than the present failure.

Ref. R351, Opening discussion and raw-stack exception reconstruction examples. See also: Exception record; Execution Residue; Custom Exception Handler.

Hidden Exception (kernel space)

Memory patterns. An earlier kernel exception recoverable from raw stack contents although it is not immediately presented as the current exception. Exception-dispatch residue can lead to a saved context and the faulting instruction. We check the recovered registers and memory access before interpreting that context, preserving the distinction between an earlier exception and the state selected when the dump opened.

Ref. R583, Hidden Exception (kernel space) pattern and diagnostic examples. See also: Hidden Exception; Execution Residue; Register context.

Hidden Exception (managed space)

Memory patterns. A managed exception is absent from the obvious current stack or exception display but remains discoverable through managed objects referenced in stack memory. We search the captured stack region and inspect candidate exception objects, including their recorded stacks. These references can recover earlier failure information even when the thread is now waiting, although retained objects do not establish a current unhandled exception.

Ref. R705, Crash Dump Analysis Patterns (Part 8c). See also: Hidden Exception; Managed Code Exception; Execution Residue (managed space).

Hidden Facts

Trace patterns. Relevant facts uncovered during trace analysis that were absent from the initial problem description. They may concern identities, dependencies, configuration, or the actual activity performed. These facts connect the artifact to subsequent troubleshooting recommendations and can require revision of the initial description or diagnostic hypothesis.

Ref. R002, p. 154. See also: Basic Facts; Indirect Facts; Data Flow.

Hidden IRP

Memory patterns. An outstanding I/O request is absent from the normal thread display but can be found by searching the broader IRP population or surviving stack references. We correlate the request's thread association, device, and current stack location to recover the missing I/O context. Its omission from one command's output should not end investigation of a suspected pending operation.

Ref. R792, Crash Dump Analysis Patterns (Part 208). See also: IRP; Execution Residue; Wait Chain.

Hidden Module

Memory patterns. An executable image exists in memory but is absent from the normal loaded or unloaded module lists. We scan for image structures and examine headers, resources, and mapped contents to identify the component and its role. Hidden presence can arise from legitimate software as well as malicious activity, so the discovery establishes an additional module to investigate, not its intent.

Ref. R743, Crash Dump Analysis Patterns (Part 75). See also: Deviant Module; Module Collection (predicate); Execution Residue.

Hidden Output

Debuggers. A workaround that suppresses an unwanted visible result, such as an error dialog or display artifact. It changes what the user sees while the condition producing that output may continue. We record the suppressed observation and the circumstances in which the workaround applies, so the altered presentation remains understandable during subsequent diagnosis.

Ref. R845, Pattern discussion. See also: Workaround; Error Message Box; Axed Code.

Hidden Parameter

Memory patterns. A useful function argument is missing from the displayed stack because it was passed in a register, optimized away, or no longer occupies the expected slot. We inspect raw memory around the relevant frames for surviving values or strings. Such residue can reveal a pipe name or other diagnostic clue, which we then validate against the operation.

Ref. R774, Crash Dump Analysis Patterns (Part 155). See also: False Function Parameters; String Parameter; Execution Residue.

Hidden Process

Memory patterns. A process structure exists in memory but is absent from the process list traversed by a conventional enumeration command. Searching suitable kernel allocations may reveal it. We inspect initialization, termination, parentage, threads, and related structures before interpreting the omission: a process being created can produce this pattern without deliberate concealment or malicious behavior.

Ref. R538, Pattern description: Hidden Process. See also: Missing Thread; Ghost Thread; Corrupt Structure.

Hidden Stack

Memory patterns. A stack-like memory region survives although its thread is absent from normal enumeration. We inspect virtual-memory layout and protection clues, then validate candidate stack contents to recover historical execution. Such regions can be especially useful in a nearly terminated process, but guard-page or layout resemblance alone does not prove the region belongs to a particular missing thread.

Ref. R810, Crash Dump Analysis Patterns (Part 263). See also: One-Thread Process; Missing Thread; Past Stack Trace.

Hidden Stack Trace

Memory patterns. A portion of the current logical call sequence is missing from an otherwise plausible stack trace. Exception-filter processing may leave an accessible context record that reconstructs the omitted frames. We inspect handler parameters and corroborating exception records before accepting a seemingly complete trace, distinguishing this hidden current context from historical stack residue left by earlier execution.

Ref. R553, Pattern description: Hidden Stack Trace. See also: Hidden Exception; Invalid Exception Information; Execution Residue.

High Contention

Memory patterns. A synchronization resource attracting enough competing activity to become a significant obstacle to progress. Owners, waiters, and contention measurements help us reconstruct how work is serialized around the resource. We distinguish cumulative contention counters from the current waiting population and relate both to the symptom interval. A frequently acquired lock is not necessarily inefficient; its protected work and throughput determine the significance.

Ref. R363, Critical-section contention discussion and allocator-lock example. See also: Critical section; Accidental Lock; Waiting Thread Time.

High Contention (executive resources)
Memory patterns. A large accumulated contention count on a Windows executive resource. The count can preserve evidence of repeated waiting even when few threads are blocked at capture time. We relate it to the owning thread, execution time, and other observations. It is a historical indicator within the object's lifetime, not a direct count of simultaneous waiters or proof of a current deadlock.

Ref. R601, High Contention (executive resources) pattern and diagnostic examples. See also: High Contention; Wait Chain; Contention.

High Contention (.NET CLR monitors)
Memory patterns. Several managed threads compete for a CLR monitor, while its owner may itself be blocked elsewhere. We correlate thread stacks, synchronization-block ownership, and recorded processor use to reconstruct the contention. A group captured waiting on one object identifies a shared dependency; the owner's subsequent wait can explain why the contention persists.

Ref. R478, Pattern description: High Contention (.NET CLR monitors). See also: Wait Chain (CLR monitors); Distributed Spike; Managed Stack Trace.

High Contention (.NET heap)
Memory patterns. Managed allocation and garbage-collection activity forms a contention point visible in native stack collections. Threads may wait for object allocation while other threads execute collection-related heap functions and consume CPU time. We compare those roles and the runtime's heap state to understand the delay. The presence of garbage-collector frames alone does not establish abnormal contention.

Ref. R602, High Contention (.NET heap) pattern and diagnostic examples. See also: Garbage collection; Managed heap; Stack Trace Collection.

High Contention (processors)
Memory patterns. More runnable threads compete for processor service than the available processors can immediately supply, for example after a common event wakes many equal-priority threads. We compare priorities, processor availability, wait objects, and stack similarity to explain bursts of contention. The relevant observation is simultaneous demand for execution; merely having many threads does not establish this pattern.

Ref. R723, Crash Dump Analysis Patterns (Part 29c). See also: High Contention; Thread Starvation (normal priority); Affine Thread.

High Contention (sockets)
Memory patterns. Many threads accumulate in socket operations such as send, receive, or select, suggesting a shared network-related source of delay or competition. We compare the stack population with a healthy case and investigate endpoints and dependencies. Network waits can be ordinary application behavior, so their frequency and relation to the reported slowdown matter more than the presence of a socket API alone.

Ref. R724, Crash Dump Analysis Patterns (Part 29f). See also: High Contention; Network Packet Buildup; Wait Chain.

Histogram

Metrics. A representation of an observation distribution through counts associated with value ranges, usually accompanied by a total count and sum. Histograms preserve more shape than an average while discarding individual observations. In Prometheus, classic and native histograms encode buckets differently. Bucket resolution and range influence the estimates obtainable from the artifact; a reported percentile should be interpreted with those limits.

Ref. R270, Histogram. See also: Histogram bucket; Quantile; Summary.

Histogram bucket

Metrics. A range of observation values and its associated count. Bucket definitions determine which distinctions remain visible after aggregation. Classic Prometheus histograms expose cumulative counts up to each upper boundary; summing these bucket values as though the ranges were disjoint counts observations repeatedly. Changes inside a wide bucket can remain hidden even when they matter to a latency objective.

Ref. R271, Histograms and summaries; quantiles. See also: Histogram; Quantile; SLO.

Historical Information

Memory patterns. Evidence of earlier activity retained in the current memory state. Request structures, recorded allocation stacks, unloaded-module information, and residual stack data can extend analysis beyond the instant of capture. We establish how each record was produced, retained, and possibly overwritten before reconstructing a sequence. Historical fragments are selective observations; their coexistence in one dump does not supply a complete or automatically ordered history.

Ref. R368, Opening discussion and list of historical evidence retained in dumps. See also: Execution Residue; IRP; Memory dump.

Hooked Functions

Memory patterns. Functions whose normal entry or execution path has been redirected by another component. Inspecting modified instructions and following their destinations reveals interactions that a module inventory alone can miss. We compare against the correct executable image and account for legitimate relocation or instrumentation. The presence of a hook establishes altered control flow; its purpose and connection to the failure require further evidence.

Ref. R372, Patched-function discussion and jump-destination examples. See also: Disassembly; Module Variety; Changed Environment.

Hooked Functions (kernel space)

Memory patterns. Kernel execution has been redirected through modified function code or service-table entries. We compare captured bytes and pointer destinations with the expected images, then identify the components receiving control. The modification establishes an interception mechanism; its diagnostic relevance and legitimacy require the surrounding product, version, and failure context rather than a conclusion based on patching alone.

Ref. R480, Pattern description: Hooked Functions (kernel space). See also: Hooked Functions; Hooked Modules; Hooking Level.

Hooked Modules

Memory patterns. Code changes detected across loaded modules when their memory images are checked against the expected images. Extending a single-function check to the module collection helps locate otherwise overlooked hooks or patches. We examine the changed locations and their context before assigning meaning: identifying modified bytes establishes an instrumentation or modification question, not automatically a malicious component or a defect.

Ref. R604, Hooked Modules pattern and diagnostic examples. See also: Hooked Functions; Module Collection; Hooking Level.

Hooking Level

Memory patterns. The extent of function patching in a captured environment, measured by the number of hooked functions. Comparing that extent between working and failing cases can reveal different instrumentation or product configurations. We inspect which functions differ as well as the count. More or fewer hooks alone does not determine correctness, since the affected operations and configuration matter.

Ref. R638, Hooking Level pattern and diagnostic examples. See also: Hooked Functions; Hooked Modules; Changed Environment.

Hooksware

Foundations. Software whose operation depends heavily on hooks introduced through supported hooking facilities or other mechanisms, including code patching. Its diagnostic footprint can involve modified functions, altered module layout, hidden components, or execution residue. We examine the specific hook and its effects. The classification describes an implementation dependence; both legitimate software and unwanted software can exhibit it.

Ref. R839, Pattern discussion. See also: Patched Code; Hooked Functions; Hidden Module.

Hotspot

Performance. A code region or call path accounting for a large share of a selected measurement during a profiled workload. It is a candidate for investigation rather than proof of inefficient code. High cost can result from many cheap calls or fewer expensive calls. A wide ancestor frame may chiefly represent descendant work, so inspect attribution before choosing an optimization target.

Ref. R392, Description. See also: CPU profile; Inclusive time; Exclusive time.

HPROF

Runtimes. A binary heap-dump format used by supported JVM tools for offline object analysis. A dump can encode objects, classes, references, and root information, with details determined by the producer and format version. Record the producing runtime and capture options when interpreting the artifact. A heap-format file does not supply a complete native process image or the history of object ownership changes.

Ref. R215, Binary heap-dump output. See also: Heap dump; jmap; GC root.

HTTP status code

Networks. A response code describing the server's result or response semantics for an HTTP request. Code classes provide a first classification, but the specific code, method, intermediary, and application contract determine its meaning. A successful HTTP response can still carry an incorrect business result, while a gateway error can describe a dependency failure. Retain response context and trace relationships before assigning cause.

Ref. R308, Section 15: Status Codes. See also: Span status; Idempotency; Log correlation.

Huge page

Memory. A memory page larger than the architecture's ordinary base page, supported through platform-specific mechanisms. Linux HugeTLB is one such mechanism with its own allocation and accounting behavior. We identify the actual page type before explaining reservation, usage, or allocation failure. A shortage in a configured huge-page pool can occur despite apparently available ordinary memory, and different large-page mechanisms have different operational constraints.

Ref. R125, Overview. See also: Page table; Memory pressure; NUMA.

Hyperdump

Memory patterns. A host-process dump contains memory regions that represent part or all of a guest system's memory, as can occur with a hosted hypervisor. We identify candidate regions from their allocation context, size, and contents, then analyze their internal artifacts with the guest relationship in mind. A large matching allocation alone does not establish a complete, directly interpretable guest physical dump.

Ref. R561, Pattern description: Hyperdump. See also: Virtualized Process (ARM64EC and CHPE); Value References; Region Clusters.

I

Iconic Trace

Trace patterns. A symbolic representation of trace messages, Activity Regions, Motifs, or an Activity Theatre. We choose a symbol profile and map the relevant structures into it, making their arrangement available for comparison. The same approach can represent a CoTrace. Interpretation depends on the stated meaning of the symbols; their appearance alone does not establish a diagnostic relationship.

Ref. R425, Part 230. See also: Dia|gram; Trace Sketch; CoTrace.

Idempotency

Distributed systems. A property whereby repeating an operation has the same intended effect as performing it once, under the operation's defined scope. It supports safe retries when completion is uncertain. It does not require identical response messages or prevent all ancillary logging effects. Diagnostic investigation should identify the idempotency key or operation semantics and distinguish a repeated request from a newly intended action.

Ref. R308, Section 9.2.2: Idempotent Methods. See also: Retry; Correlation identifier; HTTP status code.

Identification Messages

Trace patterns. Messages used to identify the object, component, or subsystem described in a problem report. Characteristic properties or behavior can identify an otherwise unknown handle or instance. Having established the identity, we can use its attributes to select the appropriate activity in the same or another trace.

Ref. R002, pp. 155–156. See also: Basic Facts; Data Selector; Intrinsic ID.

Implementation Discourse

Trace patterns. The vocabulary, structure, and assumptions of trace messages shaped by a component's language, runtime, interface, and implementation. Different components can describe similar operations in different discourses. Recognizing the discourse helps interpret status values and message sequences without imposing one subsystem's conventions on another.

Ref. R002, p. 157. See also: Trace Schema; Equivalent Messages; Software narratology.

Implicit Message

Trace patterns. An unrecorded message inferred from surrounding trace evidence and a model of the activity. Adding such messages can make a sparse narrative more explicit or support a proposed training representation. We mark them as inferred and preserve their basis. They differ from Silent Messages, which occupy resolution gaps, and must not be presented as records actually emitted by the system.

Ref. R444, Part 249. See also: Silent Messages; Sparse Trace; Trace Bias.

Impossible Trace

Trace patterns. A trace that appears inconsistent with the execution expected from the source, binary, or system model being used. Missing mandatory-looking messages can expose version mismatch, changed control flow, patching, or incomplete observation. The apparent impossibility is a conflict between evidence and model that requires investigation of both.

Ref. R002, p. 158. See also: Declarative Trace; Missing Message; Incomplete History.

Incident

Reliability. An operational event requiring coordinated response because a service is impaired or a significant risk is unfolding. Its response includes assessing impact, preserving useful observations, mitigating the condition, and keeping participants informed. Incident boundaries are organizational decisions as well as technical observations. The initial symptom, detected time, declared time, and restored-service time should remain distinguishable in the record.

Ref. R283, Incident management; live incident state document. See also: Incident commander; Postmortem; MTTD.

Incident commander

Reliability. The person responsible for coordinating an incident response, maintaining the overall state, and assigning responsibilities. This role creates a coherent account of what is known, what is being changed, and what remains uncertain. It need not be the person performing every technical investigation. Explicit delegation and handoff prevent gaps in ownership while operational work and communication proceed in parallel.

Ref. R283, Incident Command; Clear, Live Handoff. See also: Incident; Runbook; Postmortem.

Inclusive time

Performance. Time attributed to an operation together with the descendant work included by the profiler's model. A high inclusive value can identify an important execution route even when the operation itself performs little work. Inclusive totals at different nesting levels overlap and should not simply be added. State whether the underlying quantity is CPU time, elapsed duration, or an estimate from samples.

Ref. R396, CPU utilization data columns: Total CPU. See also: Exclusive time; Call graph; Flame graph.

Incomplete History

Trace patterns. A recorded history missing events expected from the relevant operation, such as a response after a logged request. The expectation depends on whether tracing covers normal returns and exception paths and whether capture continued long enough. The pattern makes missing evidence explicit instead of silently completing the narrative.

Ref. R002, p. 159. See also: Dialogue; Missing Message; Truncated Trace.

Incomplete Session

Memory patterns. A session lacks part of its expected process population, suggesting incomplete startup, incomplete termination, or another interrupted lifecycle transition. We compare the session with an appropriate normal state and inspect the remaining processes and dependencies. The expected population depends on platform and configuration. Absence becomes diagnostic evidence only when the missing process was expected at that stage.

Ref. R660, Incomplete Session pattern and diagnostic examples. See also: Missing Process; Process Factory; Coupled Processes.

Inconsistent Dump

Memory patterns. A dump whose related data represents incompatible moments because execution or modification continued during collection. A list and its objects can consequently disagree without the same inconsistency ever existing in a stopped target. We assess the acquisition method and validate relationships across independent structures. Useful evidence can remain, but conclusions requiring a single coherent system state need additional support.

Ref. R350, Opening definition, changing-data example, and comments on collection methods. See also: Memory dump; Corrupt Dump; Object Distribution Anomaly.

Incorrect Stack Trace

Memory patterns. A displayed call sequence that does not faithfully reconstruct the thread's execution frames. Corruption, unsuitable symbols, missing unwind information, or an inappropriate context can produce plausible but misleading callers. We compare return addresses, instructions, stack bounds, and available unwind metadata before interpreting the sequence. A debugger warning is a reason to investigate the reconstruction, not a complete explanation of its failure.

Ref. R353, Stack-walking warnings, implausible caller examples, and manual reconstruction. See also: Stack unwinding; Frame Pointer Omission; Coincidental Symbolic Information.

Incorrect Symbolic Information

Memory patterns. Symbolic output assigns an incorrect function, offset, or module identity to captured addresses. Causes can include stale user-space symbols after a process-context change or damaged unloaded-module metadata. We verify the context, symbol state, and underlying bytes before following the name. An apparent unfamiliar module can be a malformed fragment of another name rather than a real loaded component.

Ref. R631, Incorrect Symbolic Information pattern and diagnostic examples. See also: Symbol file; Debugger context; Unloaded Module.

Indexical Trace

Trace patterns. A trace whose messages point to relevant regions in another trace or trace collection. A broker log can serve as an index into large server logs when the problem interval is unknown. The links may use identifiers, timing, or recorded execution accounts and must preserve the identity of the indexed artifact.

Ref. R002, p. 160. See also: Data Selector; Intrinsic ID; Split Trace.

Indirect Facts

Trace patterns. Facts recovered indirectly from messages or other patterns when the problem description omits them. A message from another component may reveal the identifier of the process being investigated. The inference should retain its source and identity assumptions so that later selection does not conceal uncertainty.

Ref. R002, p. 161. See also: Basic Facts; Identification Messages; Hidden Facts.

Indirect Message

Trace patterns. A message located through an expected cause, effect, or side effect when direct messages for the reported activity cannot be found. Its surrounding context becomes a new search region. This is useful with sparse or unfamiliar tracing, provided the inferred relationship is treated as a hypothesis until checked.

Ref. R002, pp. 162–166. See also: Sparse Trace; Message Context; Hidden Error.

Indirect prompt injection

AI and ML. Instructions placed in external material that an LLM application encounters while carrying out another task. Diagnostic logs, repository files, or retrieved pages can carry such content. The failure occurs when source material is treated as authority to change the task or tool behavior. Preserving the distinction between evidence and instructions is necessary when automating analysis over untrusted artifacts.

Ref. R248, Indirect injection through externally supplied content. See also: Agentic debugging; RAG; Tool-augmented language model.

Injected Symbols

Memory patterns. Symbols supplied from an additional module provide otherwise missing names or type information for a debugging task. We make the source of that information explicit and verify that any reused layout fits the target context. The resulting symbolic interpretation is an analytical aid; it does not mean that the injected module actually participated in the captured execution.

Ref. R542, Pattern description: Injected Symbols. See also: No Component Symbols; Unrecognizable Symbolic Information; Coincidental Symbolic Information.

Inline Function Optimization (managed code)

Memory patterns. A JIT compiler incorporates a managed callee's operations into its caller, so the source-level method boundary need not appear as an ordinary call frame. We compare managed method information with the generated instructions before treating an absent frame as missing execution. The pattern concerns an optimization of representation, not proof that the source operation was skipped.

Ref. R620, Inline Function Optimization (managed code) pattern and diagnostic examples. See also: Inline Function Optimization (unmanaged code); JIT Code (.NET); Managed Stack Trace.

Inline Function Optimization (unmanaged code)

Memory patterns. Compiler inlining changes the visible call structure while retaining the relevant operation inside another function. We compare disassembly and surrounding frames across architectures or product versions instead of matching only the top function name. A missing explicit call frame can therefore explain different-looking signatures for the same operation without requiring a different defect mechanism.

Ref. R496, Pattern description: Inline Function Optimization (unmanaged code). See also: Optimized Code (function parameter reuse); Stack Trace [Memory patterns]; Constant Subtrace.

Instrumentation

Telemetry. Mechanisms that cause software activity to produce observations for later analysis. Instrumentation can be written explicitly in application code or supplied by libraries, agents, and runtime facilities. Its placement determines which operations and boundaries become visible. We should distinguish an operation that was not instrumented from an instrumented operation whose data was subsequently filtered, sampled, or lost.

Ref. R251, Code-based and zero-code instrumentation. See also: Automatic instrumentation; Manual instrumentation; Telemetry loss.

Instrumentation Information

Memory patterns. Evidence that diagnostic instrumentation was enabled and affected execution or capture. Verifier flags, allocation histories, or additional checking frames can explain why a reproduced failure appears differently from an uninstrumented one. We inspect the active settings and detector path before comparing artifacts. Instrumentation can bring detection closer to a defect while also changing the observable sequence and timing.

Ref. R633, Instrumentation Information pattern and diagnostic examples. See also: Instrumentation; Driver Verifier; PageHeap.

Instrumentation overhead

Performance. Additional work and resource consumption introduced by collecting observations, such as executing probes, capturing stacks, and recording events. Overhead depends on frequency, coverage, and workload, so a single percentage is not universal. It can change relative costs as well as total duration. Compare equivalent workloads and inspect collection settings before interpreting a profiled slowdown as ordinary application behavior.

Ref. R391, Instrumentation; Sampling versus instrumentation. See also: Instrumentation profiler; Observer effect; Profiling.

Instrumentation profiler

Performance. A profiler that inserts probes or uses callbacks at selected execution boundaries to collect events, counts, or timing. It can distinguish individual calls that sampling would miss, but its coverage follows the instrumented boundaries. Recorded durations describe the instrumented execution. Frequent short operations may be disproportionately affected by probe cost, so detailed measurement does not automatically imply an undisturbed workload.

Ref. R391, Instrumentation; Sampling versus instrumentation. See also: Sampling profiler; Instrumentation overhead; Observer effect.

Instrumentation scope

Telemetry. In OpenTelemetry, the identity of the logical software unit that produces telemetry through an instrumentation library, module, or component. Its name and optional version help distinguish two producers reporting on the same service. Scope describes the instrumentation origin; resource attributes describe the emitting entity. Confusing these dimensions can make a library upgrade appear to be a change in application identity.

Ref. R254, Instrumentation scope. See also: Resource attributes; Semantic conventions; OTel.

Instrumentation Side Effect
Memory patterns. A change in memory use, execution cost, or behavior introduced by diagnostic instrumentation itself. Full page heap and allocation-stack recording, for example, can substantially enlarge a process and add work to allocation paths. We inspect the enabled options and their retained data before attributing the observed resource growth entirely to the application's original problem.

Ref. R518, Pattern description: Instrumentation Side Effect. See also: Insufficient Memory (stack trace database); Memory Leak (process heap); Changed Environment.

Insufficient Memory (committed memory)
Memory patterns. The system has approached or exhausted its commit capacity, so further committed-memory requests can fail. We compare commit usage and limits with process, session, and allocation evidence. A later access violation may follow an unchecked allocation failure. The pattern identifies a resource condition; growth, workload, and capacity evidence are needed to distinguish its possible causes.

Ref. R463, Pattern description: Insufficient Memory (committed memory). See also: Insufficient Memory (physical memory); Memory Leak (process heap); Resource exhaustion.

Insufficient Memory (control blocks)
Memory patterns. A subsystem cannot obtain enough management structures for the resources it serves, even when a different memory measure appears adequate. The original example concerns cache-manager view-control blocks. We examine the specific control-block limit and dependent activity, because exhaustion can propagate into delayed requests, starvation, or a perceived system hang without a general heap shortage.

Ref. R465, Pattern description: Insufficient Memory (control blocks). See also: Resource exhaustion; Insufficient Memory (kernel pool); Blocked Thread.

Insufficient Memory (handle leak)
Memory patterns. Memory exhaustion caused by leaked handles retaining operating-system objects. The visible shortage may be in a kernel pool even when the accumulating handles belong to a user process. We correlate pool consumption, process handle counts, and object types, then inspect retained objects for origin clues. Terminated threads can remain allocated while handles continue to reference them.

Ref. R587, Insufficient Memory (handle leak) pattern and diagnostic examples. See also: Handle Leak; Resource exhaustion; PoolMon.

Insufficient Memory (kernel pool)
Memory patterns. Kernel allocation cannot be satisfied from the relevant pool, even when other memory categories still have capacity. We examine pool usage, allocation failures, and tag distributions to identify the exhausted resource and its principal consumers. Limits depend on the target configuration and operating-system generation; a large allocation total becomes a leak hypothesis only when ownership or growth supports it.

Ref. R709, Crash Dump Analysis Patterns (Part 13c). See also: Memory leak; Insufficient Memory (physical memory); Insufficient Memory (session pool).

Insufficient Memory (module fragmentation)
Memory patterns. Insufficient contiguous virtual-address space caused by the placement of loaded modules. An allocation can fail or shrink even when the total free address space appears substantial. We compare module distributions and region maps between working and failing configurations. Newly introduced or relocated modules guide further investigation, but their presence requires a demonstrated connection to the lost allocation opportunity.

Ref. R588, Insufficient Memory (module fragmentation) pattern and diagnostic examples. See also: Memory fragmentation; Memory map; Module Collection.

Insufficient Memory (physical memory)
Memory patterns. Available physical pages are severely constrained and memory management activity contributes to poor progress. We correlate available-page statistics, paging or fault activity, working-set operations, and runnable threads rather than relying on one large process alone. A manually triggered dump can capture this pressure; its trigger should be distinguished from the condition that made the system unresponsive.

Ref. R710, Crash Dump Analysis Patterns (Part 13f). See also: Busy System; Insufficient Memory (kernel pool); Working set [Memory].

Insufficient Memory (PTE)
Memory patterns. The available system page-table-entry resource is insufficient for operations that require virtual mappings. We inspect allocation-failure evidence and the operating system's PTE statistics in the captured configuration. The original examples concern constrained x86 Windows layouts; their boot-option and registry explanations must be interpreted within that historical platform, rather than transferred unchanged to other architectures.

Ref. R464, Pattern description: Insufficient Memory (PTE). See also: Insufficient Memory (committed memory); Page table; Virtual address.

Insufficient Memory (region)
Memory patterns. An expanding memory region reaches a limit imposed by surrounding allocations or the available address space. Moving or reallocating the region may postpone that limit without removing it. We compare successive region locations and sizes and check whether growth has stopped. The relevant shortage concerns the requested region and layout, rather than only aggregate free memory.

Ref. R590, Insufficient Memory (region) pattern and diagnostic examples. See also: Memory map; Memory fragmentation; Memory Leak (regions).

Insufficient Memory (reserved virtual memory)
Memory patterns. A shortage of usable virtual-address space while large regions remain reserved after allocations have been freed. A small live or managed heap therefore need not contradict an out-of-memory report. We inspect reservation and commitment separately and compare heap segments across relevant states. Residual allocations or module hints may identify where to investigate the reservation's origin.

Ref. R589, Insufficient Memory (reserved virtual memory) pattern and diagnostic examples. See also: Virtual memory; Reserved memory; Memory map.

Insufficient Memory (session pool)
Memory patterns. A session-specific allocation pool reaches its practical capacity while system-wide memory statistics may appear less remarkable. We inspect the affected session and its pool-tag consumers, comparing usage with the appropriate target limits and other sessions. The pattern localizes resource pressure; unusually high session consumption still requires further evidence before we attribute it to a particular leak or driver.

Ref. R711, Crash Dump Analysis Patterns (Part 13i). See also: Insufficient Memory (kernel pool); Memory leak; Object Distribution Anomaly.

Insufficient Memory (stack)
Memory patterns. The available stack region is insufficient for the execution path, sometimes producing a guarded failure or secondary diagnostic message before an ordinary stack-overflow exception becomes visible. We examine stack bounds, used regions, frames, and execution residue. A current stack pointer away from the boundary does not exclude earlier exhaustion, but a nearly full stack alone does not establish the root cause.

Ref. R712, Crash Dump Analysis Patterns (Part 13l). See also: Stack overflow; Stack Overflow (software implementation); Execution Residue.

Insufficient Memory (stack trace database)

Memory patterns. Memory retained for recorded allocation stack traces becomes a significant resource consumer. Heap totals may remain comparatively stable while the instrumentation database expands and recording consumes processor time. We compare region contents, heap statistics, and thread activity before attributing process growth to application allocations; the diagnostic acquisition mechanism can itself account for the observed pressure.

Ref. R466, Pattern description: Insufficient Memory (stack trace database). See also: Instrumentation Side Effect; Stack Trace (database); Memory Fluctuation (process heap).

Integer overflow

Execution. A computation whose mathematical result falls outside the representable range of its integer type. The resulting behavior depends on language, type, and execution rules; it is not universally wraparound. In diagnostics, we follow the value into sizes, offsets, counters, and bounds checks. A later memory access failure may originate in an earlier arithmetic error that produced a plausible smaller value.

Ref. R050, Description. See also: Buffer overflow; Out-of-bounds read; Disassembly.

Inter-Correlation

Trace patterns. Analysis of relationships between two or more traces, including concurrent captures and comparable working and nonworking runs. Shared facts, identifiers, and events establish the correspondence. We check capture scope, identity lifetimes, clock relationships, and version differences before interpreting apparently matching or conflicting records.

Ref. R002, pp. 167–168. See also: Intra-Correlation; Shared Point; Calibrating Trace.

Internal Stack Trace

Memory patterns. An uncommon but valid stack captures an internal operation that deserves attention in the reported context, even without a long wait or substantial accumulated CPU time. We inspect that path and its residue for associated components. The pattern can expose intermittent activity, such as message hooks, whose small individual delays are difficult to recognize from ordinary blocking or spiking indicators.

Ref. R798, Crash Dump Analysis Patterns (Part 226). See also: Message Hooks; Execution Residue; Special Stack Trace.

Interrupt Stack

Memory patterns. A platform-specific stack area used during interrupt handling, potentially distinct from an ordinary thread's kernel stack. We identify the applicable processor and operating-system mechanism before locating it. Besides the active interrupt context, the region can retain execution residue, so candidate symbolic addresses require the same validation as values recovered from other raw stack memory.

Ref. R564, Pattern description: Interrupt Stack. See also: Interrupt Stack Collection; Execution Residue; Rough Stack Trace (unmanaged space).

Interrupt Stack Collection

Memory patterns. The processor-specific interrupt and exception stack regions form another collection for stack and execution-residue analysis. On applicable Windows x64 systems, descriptor and task-state information can help locate stacks associated with events such as NMI, machine check, or double fault. We account for transition-stack indirection and platform layout before treating recovered addresses as frames.

Ref. R575, Pattern description: Interrupt Stack Collection. See also: Interrupt Stack; Execution Residue; Stack Overflow (kernel mode).

Interspace

Trace patterns. Regions between or around larger message spaces in a generalized trace. Such regions can link structured groups, as in debugger output that surrounds process or thread details with identifying information. Treating the interspace explicitly prevents connectors and enclosing context from being discarded during extraction.

Ref. R002, p. 169. See also: Message Space; Trace Schema; Linked Messages.

Intra-Correlation

Trace patterns. Analysis of relationships between different parts or activities within one trace. A state change in one component may correspond to another component's activity elsewhere in the log. The relationship supports an explanation only when its timing, identity, and mechanism fit the reported problem.

Ref. R002, pp. 170–172. See also: Inter-Correlation; Coupled Activities; Correlated Discontinuity.

Intrinsic ID

Trace patterns. An identifier drawn from the activity or use-case domain that helps identify the relevant trace, instance, or trail across artifacts. Examples include a session value, user, or computer name. Its scope and lifetime matter because a familiar value is not necessarily globally unique.

Ref. R002, p. 173. See also: Indexical Trace; Data Selector; Glued Activity.

Invalid Exception Information

Memory patterns. The exception information selected by a debugger is inconsistent or invalid and produces an implausible crash context. We compare registers and saved records with the actual exception-dispatch path, then reconstruct the trace from a corroborated context. A heuristic stack produced from bad exception pointers must not silently become the basis of the causal explanation.

Ref. R510, Pattern description: Invalid Exception Information. See also: Fault Context; Hidden Exception; Incorrect Stack Trace.

Invalid Handle (general)

Memory patterns. An invalid handle value is detected by an API, an explicit status check, or instrumentation. The visible exception may be raised directly or translated by a verifier into a different diagnostic stop. We inspect the call path and detection context, including whether corruption could have overwritten a formerly valid value. Detection identifies invalid use without automatically identifying its origin.

Ref. R615, Invalid Handle (general) pattern and diagnostic examples. See also: System Object; Handle Leak; Instrumentation Information.

Invalid Handle (managed space)
Memory patterns. Managed execution reaches native handle operations with an invalid handle, for example through interoperation or finalization. The visible failure depends on handle checking and instrumentation: it may appear as a native exception without a corresponding managed exception object. We inspect both stack views, the handle value, and enabled checking behavior before deciding which ownership or cleanup path is responsible.

Ref. R737, Crash Dump Analysis Patterns (Part 61b). See also: Managed Code Exception; Handle Leak; Execution Residue (managed space).

Invalid Parameter (process heap)
Memory patterns. A heap operation receives a parameter that does not designate a valid allocation for that operation. In the published example, an apparent block address contains unrelated character-like data. We check the actual argument and calling code before classifying the failure, separating an invalid supplied value from a legitimate pointer whose block was freed twice or whose heap bookkeeping was corrupted.

Ref. R758, Crash Dump Analysis Patterns (Part 117). See also: False Function Parameters; Dynamic Memory Corruption (process heap); Double Free (process heap).

Invalid Parameter (runtime function)
Memory patterns. A C runtime function receives an argument that violates its requirements and follows an invalid-parameter reporting path. We recover the call arguments using the applicable calling convention, stack, and disassembly, rather than rely on one displayed exception code. Different reporting mechanisms can expose the same class of rejected argument, while the caller explains how it arose.

Ref. R513, Pattern description: Invalid Parameter (runtime function). See also: Invalid Parameter (process heap); False Function Parameters; Self-Diagnosis (user mode).

Invalid Pointer (general)
Memory patterns. A value used as an address fails the required memory access in the captured execution context. We interpret the attempted operation, effective address, address-space mapping, and processor mode before classifying the failure. The pointer value can suggest a defect mechanism, but the access violation alone does not explain where that value originated.

Ref. R458, Pattern description: Invalid Pointer (general). See also: Invalid Pointer (objects); NULL Pointer (data); Wild Pointer.

Invalid Pointer (objects)
Memory patterns. An invalid object address used through an implicit member access or ordinary method call. Source code may show no explicit pointer dereference, while the generated instructions still use a pointer to reach the object's fields. We inspect the effective object address and faulting instruction rather than treating the apparent simplicity of the source expression as evidence of validity.

Ref. R582, Invalid Pointer (objects) pattern and diagnostic examples. See also: Invalid Pointer (general); Disassembly; Access violation.

Invariant

Program analysis. A property intended to hold at specified states or boundaries of a computation. A class invariant, for example, characterizes valid externally visible object states; temporary internal updates may pass through states that do not satisfy it. An observed violation is meaningful only with that boundary identified. Repeated observations supporting a property do not alone prove that it holds universally.

Ref. R143, Class invariants and assertion boundaries. See also: Assertion; Contract; Dynamic invariant detection.

IRP I/O request packet

Artifacts. A Windows I/O request packet carrying an operation through kernel components and device drivers. Its stack locations record information for participating drivers, while completion follows the I/O protocol. We examine the request's current state and ownership when investigating a stalled operation. The driver visible in one location is not automatically responsible for the entire elapsed delay or failure.

Ref. R103, I/O manager and I/O stack locations. See also: DPC; Kernel memory dump; Deadlock.

IRQL Interrupt request level

Execution. The Windows kernel's interrupt-priority level governing which interrupts and operations are permitted in the current processor context. It is distinct from ordinary thread scheduling priority. We interpret a driver's access and API calls against its current IRQL. An otherwise valid pageable address can become an invalid access in a context where the system cannot service the resulting page fault.

Ref. R094, IRQLs and permitted driver operations. See also: DPC; Nonpaged pool; Page fault.

ISA-Specific Code

Memory patterns. Code bytes require the instruction-set interpretation appropriate to their particular region. In Windows on ARM compatibility processes, x86 or x64 code may coexist with CHPE or ARM64EC code. We select the appropriate disassembly context for each candidate address before classifying undecodable bytes as corruption, wild code, or coincidental symbolic information.

Ref. R577, Pattern description: ISA-Specific Code. See also: Virtualized Process (ARM64EC and CHPE); Coincidental Symbolic Information; Incorrect Stack Trace.

J

jcmd

Runtimes. A JDK utility that sends diagnostic commands to a target JVM. Depending on the runtime and command, it can request recordings, thread information, histograms, heap dumps, or other diagnostics. Available commands and their impact are target-specific, so the target's help is part of the tool contract. A diagnostic command may change runtime state or pause execution while collecting evidence.

Ref. R208, Description; help and diagnostic commands. See also: JFR; Heap dump; Thread dump.

jdb

Runtimes. The JDK command-line debugger for Java programs, using the Java Platform Debugger Architecture. It can launch or attach to a supported target and inspect source-level execution. The displayed state depends on debug information, selected frames, and JVM capabilities. Evaluation and execution-control commands can affect the application, so conclusions should identify the state observed before diagnostic interaction changed it.

Ref. R216, Description; launching and attaching. See also: JPDA; JDWP; Thread dump.

JDWP Java Debug Wire Protocol

Runtimes. The protocol used to communicate debugging commands, replies, and events between components of the Java debugging architecture. It describes communication with the target rather than the source-language interface shown to the user. When debugging behavior differs across environments, transport, connection configuration, protocol capabilities, and target state are distinct diagnostic questions. The protocol connection itself does not establish semantic correctness of the application.

Ref. R213, Java Debug Wire Protocol component. See also: JPDA; JVM TI; jdb.

JFR Java Flight Recorder

Runtimes. An event-recording framework integrated into the Java runtime for collecting selected application and JVM behavior over time. Events can include execution samples, allocation, synchronization, and collection activity, depending on settings. A recording supports retrospective interpretation of an incident interval. Its evidence is bounded by enabled events, thresholds, sampling, buffers, and retention; recording existence does not imply every relevant event was captured.

Ref. R207, Flight Recorder. See also: jfr command; jcmd; NMT.

jfr command

Runtimes. The JDK command-line utility for inspecting and manipulating Flight Recorder files. It can summarize recording contents and print selected events, allowing a focused view without a graphical analysis tool. Filters and output formats change the presented view rather than the events originally recorded. Compare the report with recording metadata before treating an absent printed event as absent execution behavior.

Ref. R209, Description; print and summary commands. See also: JFR; jcmd; Thread dump.

JIT Code (Java)

Memory patterns. Java-generated machine code appears in a native process stack without an ordinary image-backed symbol for each frame. We inspect the candidate return addresses and preceding call instructions to establish that the addresses participate in a plausible execution path. The JIT interpretation explains missing module-style names while leaving the Java method identity and causal significance to further analysis.

Ref. R625, JIT Code (Java) pattern and diagnostic examples. See also: JIT Code (.NET); Stack trace [Artifacts]; Disassembly.

JIT Code (.NET)

Memory patterns. Native instructions were generated by the CLR at runtime and can differ from the compiler-generated code expected by the analyst. We use runtime context, surrounding instructions, and function boundaries to interpret an unfamiliar sequence before labeling it as wild execution. The absence of an ordinary image-backed function identity does not by itself indicate corruption.

Ref. R501, Pattern description: JIT Code (.NET). See also: JIT Code (Java); Managed Stack Trace; Wild Code.

Jitter

Distributed systems. In retry scheduling, a randomized variation in delay intended to spread attempts that would otherwise occur together. It can reduce synchronized bursts created by identical backoff policies or periodic work. The randomization distribution and bounds affect the resulting traffic. Here jitter describes scheduling variation; network delay variation is a related but separate use of the word that requires its own measurement definition.

Ref. R321, Adding jitter; comparisons of jitter algorithms. See also: Backoff; Retry storm; Latency.

jmap

Runtimes. A JDK utility for selected Java memory diagnostics, including class histograms and heap-dump generation. These views describe different aspects of a heap: an aggregate population and an object-level artifact. Java SE 25 documents jmap as experimental, and command support depends on the target. Collection options and effects must be recorded when comparing counts or interpreting what the dump contains.

Ref. R215, Description; histogram and dump options. See also: Heap histogram; HPROF; jcmd.

JMX Java Management Extensions

Runtimes. A Java management architecture that exposes resources through managed objects called MBeans. It provides attributes, operations, and notifications for inspecting and managing a running application or JVM. The exposed interface is a selected model of runtime state, not a complete execution record. Reading an attribute and invoking an operation have different effects and should be distinguished in a diagnostic procedure.

Ref. R212, JMX architecture; MBeans and agents. See also: JFR; jcmd; Diagnostic port.

JPDA Java Platform Debugger Architecture

Runtimes. The Java debugging architecture connecting debugger interfaces, the wire protocol, and the JVM's tooling interface. It separates the debugger front end from mechanisms observing and controlling the target. This separation helps locate an attachment or event-delivery problem within the diagnostic path. A successful connection proves that some communication works, not that every requested event or state view is available.

Ref. R213, Overview; front-end and back-end components. See also: JDWP; JVM TI; jdb.

jstack
Runtimes. A JDK utility that prints Java thread stacks and associated diagnostic information for a target JVM. Its output is useful for recognizing shared call contexts and waiting relationships. The command is documented as experimental in Java SE 25, where jcmd is the preferred general diagnostic utility. Stack visibility and collection behavior depend on the JVM, release, and available target state.

Ref. R214, Synopsis: experimental-tool note; Description; R207, Diagnostic Tools Overview: note recommending jcmd over jstack, jinfo, and jmap. See also: Thread dump; jcmd; jdb.

JVM fatal error log
Runtimes. A diagnostic artifact written when a JVM encounters a fatal error and can perform its error-reporting work. It may include the failing thread, native or Java context, runtime configuration, and process or system details. The artifact describes the failure boundary, not necessarily the initiating defect. Its contents and completeness depend on the JVM release and the state in which reporting occurred.

Ref. R210, Fatal-error log location, naming, and sections. See also: Thread dump; JFR; NMT.

JVM TI JVM Tool Interface
Runtimes. The JVM interface through which native tooling agents can inspect and control supported aspects of Java execution. Debuggers, profilers, and other tools use its events and capabilities to obtain runtime observations. Capabilities, execution phases, and callback restrictions constrain what an agent can do. Agent behavior and overhead also become part of the system being diagnosed.

Ref. R213, JVM Tool Interface component. See also: JPDA; JDWP; JFR.

K

Karnaugh Map
Trace patterns. A trace-analysis pattern that arranges observed condition combinations in a logical adjacency space to identify simpler descriptions of failure cases. Dimensions that vary without changing the observed outcome may be removable from that description. The simplification is bounded by the cases and variables examined; a compact Boolean condition is not, by itself, proof of causation.

Ref. R009. See also: Marked Message; Data Selector; Causal model.

kdump
Native tools. A Linux crash-dump mechanism that uses a reserved capture kernel to collect memory after the original kernel crashes. Its usefulness depends on prior configuration. We verify the capture path and retained content as part of diagnostic readiness. A reboot does not establish that a dump was saved, and filtering can limit later questions.

Ref. R035, Overview. See also: vmcore; Kernel panic; crash utility.

Kernel memory dump
Artifacts. A Windows dump emphasizing memory allocated to the kernel, drivers, and related kernel components at a system crash. It generally omits user-mode application memory. We can investigate kernel threads, locks, and driver structures within the captured scope, but a missing application object is not evidence that it never existed. Dump selection establishes a boundary for the questions the artifact can answer.

Ref. R066, Dump contents and excluded user-mode memory. See also: Complete memory dump; Active memory dump; Kernel mode.

Kernel mode

Execution. An execution mode that permits privileged operating-system operations and access to protected resources, subject to the platform's architecture. Kernel code and drivers execute within this environment. We interpret failures using kernel-specific execution, synchronization, and memory rules. The broad privilege of this mode increases the possible scope of corruption, while a kernel failure still requires evidence connecting the observed damage to its origin.

Ref. R120, User mode; Kernel mode. See also: User mode; IRQL; Kernel memory dump.

Kernel panic

Execution. A kernel-level termination of normal operation after an unrecoverable or deliberately escalated condition. Its report and any associated dump preserve evidence according to the platform and configuration. On Linux, kdump can capture memory after a panic through a separate kernel. We distinguish the immediate panic reason from earlier faults or corruption that made the final condition unavoidable.

Ref. R035, Overview; R084, Configuration; Implementation. See also: Bug check; kdump; vmcore.

Kernel space

Memory. The address-space region used for operating-system kernel code and data. The earlier Dictionary of Debugging blog entry illustrates it as an upper interval of a divided memory space. We distinguish this address-domain description from kernel mode, which describes execution privilege. The relevant mappings and dump contents determine which kernel-space locations can actually be inspected in a particular investigation.

Ref. R419, Definition and examples. See also: User space; Kernel mode; Kernel memory dump; Memory space.

Kernel tracepoint

Native tools. A statically defined instrumentation point in kernel code that allows registered tracing probes to receive the event's specified information. It provides a more explicit observation interface than probing an arbitrary instruction address. We interpret each event according to its documented fields and location in the operation. A tracepoint records the selected observation, which may represent an attempt, transition, or completion rather than the whole operation.

Ref. R039, Purpose of tracepoints. See also: ftrace; eBPF; Kprobe.

KLEE

Program analysis. A symbolic execution engine that operates on LLVM bitcode and generates concrete tests from explored paths. Symbolic inputs permit a single test harness to represent many candidate inputs. A generated test can turn a symbolic finding into a reproducible experiment. Coverage and error reports remain conditional on the compiled representation, environment models, resource limits, and search strategy used for that run.

Ref. R153, Tutorial One: symbolic inputs and generated test cases. See also: Symbolic execution; Path explosion; Test harness.

Kprobe

Native tools. A Linux dynamic probing mechanism that invokes a handler when execution reaches a selected supported kernel instruction location. Return probes provide related facilities around function returns. We use probes to observe arguments, state, or timing at points relevant to a hypothesis. Instrumentation restrictions, architecture behavior, and handler overhead matter, and a function's internal locations may change between kernel builds.

Ref. R085, Concepts: Kprobes and Return Probes. See also: eBPF; Kernel tracepoint; Uprobe.

L

Label

Metrics. A named dimension used to identify or group metric observations, such as operation, instance, or result. In Prometheus, each distinct combination of label values belongs to a separate time series. Labels make comparisons precise, but a query that removes them also removes distinctions. Aggregating across instances can conceal one unhealthy replica behind the normal behavior of many others.

Ref. R269, Metric names and labels. See also: Time series; Cardinality; Aggregation.

Large object heap

Runtimes. A .NET allocation area for objects meeting the runtime's large-object criteria, with collection and compaction behavior distinct from many smaller allocations. Large temporary objects can increase collection work and leave fragmentation. Interpretation requires the actual runtime version and configuration, because size thresholds and implementation details are not universal language guarantees. Heap totals alone do not reveal whether live data or free space dominates.

Ref. R192, Large-object allocation, collection, and compaction. See also: Managed heap; Garbage collection; Object pinning.

Last Activity

Trace patterns. The semantically relevant activity immediately preceding a significant event, disappearance, or termination. The final useful clue may come from another trace with different coverage. We examine this activity as a lead and distinguish the last recorded operation from the operation that caused the outcome.

Ref. R002, p. 174. See also: Significant Event; Discontinuity; Inter-Correlation.

Last Error Collection

Memory patterns. Last-error and status values gathered across a process's threads, or across available threads in a system dump. The collection can suggest investigation directions when the current stacks appear ordinary. We preserve the thread association and interpret each value with its API context. A retained last error can outlive the failing call and need not explain the reported incident.

Ref. R621, Last Error Collection pattern and diagnostic examples. See also: Coincidental Error Code; Stack Trace Collection; Diagnostic indicator.

Last Object

Memory patterns. The most recently added object in a relevant collection provides a useful first place to investigate an incident tied to a recent action. We inspect list tails or enumeration endings where the collection's insertion behavior supports that inference. This can narrow a large system dump quickly; if the latest candidates do not explain the problem, we expand the search.

Ref. R793, Crash Dump Analysis Patterns (Part 212). See also: Process Factory; Crashed Process; Stack Trace Collection.

Late Crash Dump

Memory patterns. A dump captured after the most useful state of an incident has disappeared, for example during process termination or after a diagnostic dialog was dismissed. We examine surviving raw-stack evidence, error values, and exception residue while recognizing the lost context. A visible termination stack can describe the aftermath without revealing the initiating failure.

Ref. R532, Pattern description: Late Crash Dump. See also: Early Crash Dump; Execution Residue; Hidden Exception.

Latency

Performance. Elapsed time between specified events in an operation. Its meaning depends on the chosen boundaries: network transfer, queue residence, server handling, and user-visible completion measure different intervals. We can divide a long interval into meaningful components to locate where time accumulated. The word latency alone is insufficient for comparison unless the observer, clock, start, end, and included work are identified.

Ref. R341, Time Division Method; By-Layer Method. See also: Response time; Queueing delay; RTT.

Latent Structure

Memory patterns. Raw memory shows incomplete signs of organization without enough evidence to assign a reliable structure. We investigate recurring values, partial strings, alignment, and pointer relationships while keeping candidate interpretations tentative. Checks of layout, internal coherence, and target conventions can support a testable hypothesis. The source names the proposed next stage Suspect Structure; premature casting can turn coincidence into misleading explanation.

Ref. R823, Crash Dump Analysis Patterns (Part 304). See also: Corrupt Structure; Regular Data; Pointer Class; Small Value.

Lateral Damage

Memory patterns. Damage to process information needed by ordinary analysis procedures, beyond the immediate application data implicated in the failure. Corrupted loader or thread structures can prevent familiar debugger commands from reconstructing a usable view. We then seek independent evidence in raw memory, registers, executable images, and other threads. Failure of the usual analysis route does not make every part of the dump unusable.

Ref. R349, Opening damaged-process examples and author comment dated November 5, 2008. See also: Memory corruption; Memory examination; Corrupt Dump.

Lateral Damage (CPU mode)
Memory patterns. A damaged or misleading CPU-mode context that makes debugger commands interpret a dump incorrectly. An x64 dump may initially present an x86 or segmented-addressing view, producing implausible parameters or stacks. We verify the effective machine and addressing mode before accepting those results. Correcting the interpretation can recover usable evidence without repairing genuinely missing registers or memory.

Ref. R581, Lateral Damage (CPU mode) pattern and diagnostic examples. See also: Lateral Damage; Debugger context; Rough Stack Trace (unmanaged space).

Layered Periodization
Trace patterns. A partitioning of trace history at several organizational levels, such as individual messages, threads, processes, and computers. The layers reveal how detailed events fit within longer-lived activities. Their boundaries need not coincide, and a component or function can participate in more than one execution context.

Ref. R002, pp. 175–177. See also: Trace Partition; Thread of Activity; Adjoint Thread of Activity.

LeakSanitizer
Program analysis. A runtime memory-leak detector available with AddressSanitizer or in a standalone configuration. It examines allocation reachability, commonly during a phase near process termination, and reports allocations classified as leaked. The report concerns the detector's reachability model and collection time. Memory that remains reachable through an unwanted cache or owner can still grow excessively without appearing as an unreachable leak.

Ref. R179, Introduction; integrated and standalone use. See also: AddressSanitizer; Memcheck; Retained size.

Least Common Frame
Memory patterns. A shared stack frame that becomes useful when an exact crash signature finds no comparable incident. We move through the trace, search candidate frames, and constrain results by relevant application or module context. A matching frame and subsequent subtrace can connect related incidents, but common runtime infrastructure alone provides weak evidence of a common cause.

Ref. R519, Pattern description: Least Common Frame. See also: Stack Trace [Memory patterns]; Special Stack Trace; Module Variety.

libFuzzer
Testing. An in-process, coverage-guided fuzzing engine used with an instrumented fuzz target. It mutates a corpus and retains inputs that provide useful execution feedback. Crashes and sanitizer findings can become repeatable test cases. Because many inputs run in one process, persistent state and incomplete cleanup in the target can distort later cases; the harness must make its intended state model explicit.

Ref. R162, Introduction; fuzz target; corpus. See also: Coverage-guided fuzzing; Test harness; AddressSanitizer.

LIME Local Interpretable Model-agnostic Explanations
AI and ML. A method that approximates a model's behavior near a selected example with an interpretable local model. The resulting feature weights describe the approximation under the chosen perturbations and neighborhood. They can help examine a diagnostic prediction's sensitivity. Local fidelity does not imply a faithful global description, and perturbed examples may differ from realistic software observations.

Ref. R243, Local interpretable model-agnostic explanations. See also: Explainable AI; SHAP; Feature vector.

Linked Messages

Trace patterns. Messages connected through a shared value or an explicit relationship between values, such as a process identifier and its parent's identifier. Such links support navigation across otherwise separate message families. The relation must respect identifier scope and lifetime so that reuse does not join unrelated activities.

Ref. R002, pp. 178–179. See also: Data Flow; Intrinsic ID; Braid of Activity.

Listing CPUs

Debuggers. Enumerating processor contexts represented by the target or artifact. The collection supports inspection of per-processor state and comparison with running threads or interrupt activity. We identify whether the view concerns physical, logical, or virtual processors, and preserve capture timing. The existence of a processor context does not establish how much work it performed.

Ref. R001, pp. 268–270. See also: CPU time; Stack Trace Collection (CPUs); Dumping Thread State.

Listing Heap Entries

Debuggers. Enumerating allocation entries within a selected heap so that their sizes, states, and contents can be examined. We relate each entry to its allocator and acquisition context before counting, grouping, or interpreting it. The result provides a basis for identifying relevant allocations; allocation metadata alone does not establish the application-level object or ownership.

Ref. R001, pp. 268–270. See also: Heap; Listing Heaps; Dumping Memory.

Listing Heaps

Debuggers. Enumerating the heaps represented within the selected analysis context. This provides candidates for examining allocation structure, contents, and resource use. We preserve heap identity and distinguish runtime or allocator families before comparing them. The enumeration describes the heaps visible through the chosen artifact and tool; additional address spaces or incomplete capture can limit that view.

Ref. R001, pp. 268–270. See also: Heap; Listing Heap Entries; Memory map.

Listing Memory Regions

Debuggers. Enumerating memory ranges and their available descriptive properties within a selected address space. The result supplies a map for interpreting addresses and choosing ranges to inspect. We preserve context, boundaries, and capture availability; a listed mapping does not guarantee that its contents are readable in the artifact being analyzed.

Ref. R001, pp. 268–270. See also: Memory map; Memory Region; Dumping Memory.

Listing Modules

Debuggers. Enumerating executable components represented in the selected target or artifact. The collection supports address attribution, version comparison, and symbol preparation. We record the enumeration’s scope and source before treating it as complete, then investigate components found through other structures or memory searches that do not appear in the ordinary module list.

Ref. R001, pp. 268–270. See also: Module Collection; Hidden Module; No Component Symbols.

Listing Object Names

Debuggers. Enumerating names associated with objects in the diagnostic scope. Names can connect handles, resources, and application roles, and help identify objects for further inspection. We preserve the namespace and object type when interpreting a name. Similar spelling does not establish identical identity or ownership across processes, namespaces, or captures.

Ref. R001, pp. 268–270. See also: Memory examination; Dumping Object State; Handle Leak.

Listing Processes

Debuggers. Enumerating the processes represented by the current diagnostic target or artifact. The resulting collection establishes candidates for selecting an analysis context and relating activities across process boundaries. We retain process identity and capture scope, because a list produced from a restricted artifact need not represent every process that existed on the original system.

Ref. R001, pp. 268–270. See also: Process; Setting Processes; Hidden Process.

Listing Stack Traces

Debuggers. Collecting reconstructed call stacks for selected threads or execution contexts. This elementary action supplies material for stack comparison, clustering, and dependency analysis. We retain thread identity and any unwinding limitations. Producing a list is the acquisition step within the analysis; the relationships among the stacks require further interpretation.

Ref. R001, pp. 268–270. See also: Stack Trace [Memory patterns]; Stack Trace Collection; Listing Threads.

Listing Threads

Debuggers. Enumerating the threads represented in a selected process, system, or captured artifact. This elementary action supplies identities for subsequent state and stack inspection. We preserve the scope and selection rules of the enumeration, then distinguish an absent thread from one omitted by the artifact or by the tool's current view.

Ref. R001, pp. 268–270. See also: Thread; Setting Threads; Stack Trace Collection.

Little's law

Performance. A relationship between average population, average arrival or throughput rate, and average time spent within the same system boundary. With appropriate long-run conditions, average population equals rate multiplied by residence time. We can use it to check whether queue, throughput, and latency measurements describe a consistent model. Mixing queue-only waiting with total in-flight work, or incompatible intervals, invalidates the comparison.

Ref. R317, p. 417; limiting-average conditions. See also: Queue length; Queueing delay; Throughput.

Livelock

Memory patterns. Threads repeatedly acquire and release resources, consume execution time, and change state without advancing their intended work. We correlate CPU time, contention, context switches, priorities, and similar stack traces to distinguish this activity from a stable wait. Repeated captures or other temporal evidence help establish the absence of progress; a single busy stack is insufficient.

Ref. R516, Pattern description: Livelock. See also: High Contention; Distributed Spike; Busy System.

Livelock

Execution. A failure of useful progress in which participants continue taking actions in response to one another. Repeated retries or state changes can keep the system active while the intended operation never completes. We compare execution activity with progress indicators and dependency behavior. CPU consumption and changing stack samples help distinguish this situation from simple blocked waiting, but do not alone establish livelock.

Ref. R102, Starvation; Livelock. See also: Deadlock; Starvation; Context switch.

Liveness probe

Cloud. A Kubernetes check used to decide whether a running container should be restarted after the configured failure conditions are met. It should identify a condition for which restarting is appropriate. A probe failure describes the check's result, not necessarily an application crash. Aggressive timing or dependency-sensitive checks can create restart cycles during slow startup, overload, or an external outage.

Ref. R293, Liveness probe. See also: Readiness probe; Startup probe; CrashLoopBackOff.

LLDB

Debuggers. The debugger associated with the LLVM project. It provides target, process, thread, frame, and expression facilities for supported platforms, with commands organized around these objects. We use this structure to make the selected context explicit during analysis. A convenient command abbreviation should not obscure whether we are examining the current frame, another thread, or data that requires the target to execute.

Ref. R033, Command Structure; Examining Thread State; Examining Stack Frame State. See also: Debugger context; Expression evaluation; Symbolication.

LLM-as-a-judge

AI and ML. Use of a large language model to evaluate another response or artifact against stated criteria. It can help compare diagnostic explanations or identify missing evidence, but its score is another model output. Position, verbosity, and preference biases can affect judgments. Agreement with a model judge should not replace checking commands, references, and conclusions against the underlying software evidence.

Ref. R242, LLM judges; limitations and biases. See also: LLM-assisted debugging; Hallucination; Test oracle.

LLM-assisted debugging

AI and ML. Use of a large language model to help interpret artifacts, propose explanations, retrieve relevant material, or plan diagnostic experiments. The model's contribution is a candidate analysis that must be connected to observed software behavior. Separate supplied evidence, retrieved information, generated inference, and verified tool results. This preserves a reviewable path from the original problem to any accepted explanation or change.

Ref. R245, Reasoning and observations; diagnostic application; R241, Grounding generation in retrieved material. See also: Hallucination; Tool-augmented language model; RAG.

Load shedding

Distributed systems. Deliberately rejecting selected work to keep a service within useful capacity and protect admitted work. We examine the admission policy, priority, and offered load. Lower latency among accepted requests can accompany a worse overall user outcome, so rejection must remain part of the assessment.

Ref. R337, Load Shedding and Graceful Degradation. See also: Saturation; Graceful degradation; Throughput.

Local Buffer Overflow

Memory patterns. A memory pattern in which writing beyond a local stack buffer damages neighboring data or control information. Character fragments in a saved return address or frame pointer can reveal the kind of data involved in the overwrite. We interpret the evidence using the actual architecture and stack layout. This pattern concerns an overrun within stack storage, distinct from exhausting the available stack space.

Ref. R370, Local stack-buffer discussion and overwritten control-data example. See also: Buffer overflow; Stack overflow; Incorrect Stack Trace.

Local Buffer Overflow (kernel space)

Memory patterns. An overwrite of a kernel thread's local stack buffer damages adjacent data or control information. A later fault may expose a null execution target or an unreliable stack trace. We establish the stack bounds and saved context, then inspect raw contents for surviving call evidence; the visible crash can occur after the write that caused the damage.

Ref. R725, Crash Dump Analysis Patterns (Part 36b). See also: Local Buffer Overflow; Incorrect Stack Trace; Execution Residue.

Lock convoy

Execution. A persistent queue of threads whose progress becomes serialized through a contended lock and repeated scheduling of its owners. A delayed owner or expensive handoff can allow further waiters to accumulate, even when the protected work is brief. We compare ownership transitions, scheduling delays, and arrival rates. The mechanism concerns sustained throughput loss, rather than the mere presence of several waiting threads.

Ref. R410, Lock handoff and scheduling-delay explanation; worked arrival-rate example. See also: Contention; Context switch; High Contention; Priority inversion.

Lockdep

Native tools. The Linux kernel's runtime locking validator, which records lock-class dependencies and checks locking rules to identify potentially unsafe combinations. It can expose a possible deadlock from observed dependency relationships without waiting for that complete deadlock to occur. We interpret a report with its lock classes, acquisition paths, and configuration. Absence of a report does not establish that unexecuted paths or untracked synchronization are correct.

Ref. R041, Lock-class; dependency validation. See also: Deadlock; Spinlock; Reader-writer lock.

Log correlation

Logs. Associating log records with one another or with traces, metrics, and resources through shared context. Correlation can join observations that are scattered across files and services into a more useful execution narrative. Matching time alone is often insufficient under concurrency. Identifiers, resource identity, and operation context strengthen the association, while reused values and incomplete propagation can still produce misleading groupings.

Ref. R339, Logs; correlation with traces and resources. See also: Correlation identifier; Trace identifier; Resource attributes.

Logical clock

Distributed systems. A mechanism that assigns logical values to events so that known causal order is respected without requiring synchronized physical clocks. Lamport clocks ensure that an event preceding another in the happened-before relation receives a smaller value. The converse does not hold: smaller values do not prove causality. A total display order produced from logical clocks must therefore be distinguished from an established dependency.

Ref. R305, The Partial Ordering; Logical Clocks. See also: Causal order; Vector clock; Clock skew.

Log injection

Logs. The insertion of untrusted content into a log in a way that changes the apparent record structure or meaning. Newlines, delimiters, and improperly encoded values can create misleading events or corrupt parsing. This is both a security issue and an evidence-quality problem. Use encoding and validation appropriate to the log format; parameterized message construction alone does not necessarily neutralize control characters.

Ref. R332, Event collection; protection. See also: Structured logging; Audit log; Log record.

Log level

Logs. A severity classification assigned to a log record, such as informational, warning, or error. The classification controls selection and suggests how the producer interpreted an event. It is not a universal measure of user impact: a recovered operation may log an error, while a harmful omission may produce no record. Compare level semantics and configured thresholds before comparing logs from different components.

Ref. R260, SeverityText; SeverityNumber. See also: Log record; Span status; Alert.

Log record

Logs. An individual recorded observation emitted by software or collected from an event source. A record may contain free text, structured fields, timestamps, and execution context. It is an artifact produced by a particular code path and logging configuration. It can describe an attempted action, a completed action, or a diagnostic interpretation; these meanings should not be treated as interchangeable.

Ref. R339, Logs. See also: Structured logging; Event timestamp; Log level.

Log retention

Logs. The policy and implementation governing how long logs remain available, in which form, and with what access and integrity protections. Retention determines the historical interval that an investigation can examine. A missing older event may reflect expiry, relocation, or a changed policy rather than absence of activity. Retained aggregates and indexes may not preserve the detail of the original records.

Ref. R331, Section 3.2 (Functions—Storage), p. 3-3; section 5.4 (Manage Long-Term Log Data Storage), p. 5-9. See also: Log rotation; Audit log; Telemetry loss.

Log rotation

Logs. Managing a log stream by closing, renaming, replacing, or archiving its active files according to a size, time, or other policy. Rotation bounds local storage and changes the physical boundaries of the artifact. An investigation that reads only the current file can miss the beginning of a problem. File naming, compression, writer behavior, and collector position all affect continuity.

Ref. R331, Section 3.2 (Functions—Storage), p. 3-3; section 5.1.2 (Log Storage and Disposal), p. 5-3. See also: Log retention; Container log; Telemetry loss.

Logue

Narratology. A narrative of diagnostic actions: the commands, transformations, inspections, and decisions that form an investigation. It provides a textual counterpart to a diagnostic gesture and can itself be analyzed as a log. Recording the logue makes both the artifact interpretation and the analyst's path through tools and hypotheses available for review.

Ref. R001, pp. 331–332. See also: Diags; Analysis narrative; CoTrace.

Loop Construct

Program analysis. A source-code loop examined for the state that governs continuation, termination, and progress. Shared or overwritten control data can connect the construct with a spiking thread or unusually dense trace output. We inspect those relationships before inferring a mechanism. The pattern directs analysis toward the loop's conditions and dependencies; the existence of a loop alone does not identify a defect.

Ref. R877, Pattern discussion. See also: Spiking Thread; Statement Density and Current; Data race.

Lost Opportunity

Memory patterns. The dump was captured after the most informative phase of the incident had passed. A diagnostic dialog may have closed, a thread may have terminated, or shutdown may have replaced the failing execution path. We search remaining residue and contextual records while recognizing the loss; the observation also helps specify an earlier capture point for the next occurrence.

Ref. R742, Crash Dump Analysis Patterns (Part 72). See also: Early Crash Dump; Execution Residue; Dump Context.

Loxel

Narratology. An element of a log or trace used in a constructed graphical representation, usually corresponding to one message. A message identifier may determine its color, and a sequence of loxels can be arranged as an image for inspection or processing. This representation reflects the chosen encoding rather than a direct image of the original memory bytes.

Ref. R001, pp. 307–312. See also: Mexel; Message Invariant; Trace Shape.

M

Mach-O

Artifacts. The executable-image format used on Apple platforms. Executables, dynamic libraries, frameworks, and bundles can contain loadable Mach-O images. Their build UUIDs support identification and matching to debug information. We preserve the image identity and load addresses when interpreting a crash report. An application name or release label is insufficient to select the corresponding symbols for every included image.

Ref. R100, Overview; Use unique build UUIDs. See also: Executable image; dSYM; Symbolication.

Macrofunction

Trace patterns. Several messages interpreted together as one higher-level semantic operation, such as connecting to a database, issuing a command, and closing the connection. Its constituent messages may cross components and threads. The grouping exposes the operation represented by the messages without requiring a matching source-code function.

Ref. R002, p. 180. See also: Milestones; Motif; Use Case Trail.

Main Thread

Memory patterns. The application's primary thread considered as an architectural clue during dump analysis. Its stack may reveal initialization, the principal input loop, shutdown, or a dependency that controls the application's lifetime. We identify its actual responsibilities rather than assuming that every application assigns the same work to it. A main thread waiting normally can coexist with a failure in another participant.

Ref. R366, Primary-thread discussion and application-lifecycle examples. See also: Thread; Passive Thread; Missing Thread.

Malnarrative

Narratology. A software narrative intentionally altered for a malicious purpose, for example by inserting, changing, or suppressing material to mislead its interpretation. It is distinct from an ordinary incomplete or corrupt artifact. Analysis must establish both the changes and their context before inferring intent; preserved originals and integrity evidence help assess competing explanations.

Ref. R001, pp. 200–201. See also: Malware narrative; Palimpsest Messages; Narrachain.

Malware narrative

Narratology. A software narrative containing signs of malicious software or its activity. The artifact itself need not have been maliciously modified: ordinary logs and memory snapshots may faithfully record the relevant behavior. The distinction from a malnarrative separates what an artifact describes from whether someone has manipulated the artifact to influence the investigation.

Ref. R001, pp. 200–201. See also: Malnarrative; Software narrative; Diagnostic indicator.

Managed Code Exception

Memory patterns. An exception originates in managed execution, while the native dump view may chiefly show runtime or exception-dispatch frames. We load compatible runtime support, inspect the managed stack, and examine the exception object and inner exceptions. This connects the reported native failure to managed operations and data without treating the presence of CLR infrastructure as evidence that the runtime caused it.

Ref. R714, Crash Dump Analysis Patterns (Part 17). See also: Managed Stack Trace; Runtime Thread (CLR); Version-Specific Extension.

Managed Code Exception (Python)

Memory patterns. An exception reported by Python execution rather than a native processor fault. The exception type and message identify the language-level operation that failed, while its traceback supplies the associated call path. We preserve both kinds of evidence and distinguish them from a crash in the interpreter or native extension. The article's example is an out-of-range list access.

Ref. R593, Managed Code Exception (Python) pattern and diagnostic examples. See also: Managed Stack Trace (Python); Exception; faulthandler.

Managed Code Exception (Scala)

Memory patterns. An exception reported by Scala's managed execution environment, with its language-level type, message, and stack trace. The example uses an empty optional value whose access raises a JVM exception. We preserve that managed context when investigating a failure; an operating-system exception or a native debugger stack describes a different layer of the execution.

Ref. R469, Pattern description: Managed Code Exception (Scala). See also: Managed Stack Trace (Scala); Managed Code Exception; Exception.

Managed heap

Runtimes. Memory used for objects whose allocation and reclamation are coordinated by a language runtime. Object references, collector rules, and runtime metadata determine how this memory is interpreted. Heap growth can reflect live application state, delayed collection, or unintended retention. A managed-heap total does not describe all process memory; native allocations and other runtime structures may require separate evidence.

Ref. R211, Automatic storage management and object allocation. See also: Garbage collection; Heap dump; NMT.

Managed Stack Trace

Memory patterns. A stack trace expressed in managed runtime terms, showing managed methods and their calling relationships. Runtime-aware commands can expose this view directly or as part of a combined native and managed trace. We use it to communicate the managed execution path and relate application frames to native runtime activity, exceptions, and the data available in the dump.

Ref. R767, Crash Dump Analysis Patterns (Part 139). See also: Managed Code Exception; Runtime Thread (CLR); Mixed Exception.

Managed Stack Trace (Python)

Memory patterns. A Python traceback represents calls in the language runtime, typically identifying files, functions, and source lines associated with an exception. We read the traceback together with the exception and examine how values reached the failing operation. This managed view describes Python execution; a native interpreter or extension failure may require an additional native stack and memory context.

Ref. R593, Managed Stack Trace (Python) pattern and diagnostic examples. See also: Managed Code Exception (Python); Stack trace [Artifacts]; Exception.

Managed Stack Trace (Scala)

Memory patterns. The sequence of managed method calls reported for Scala execution, commonly through the JVM. We interpret method names, source locations, and exception context at that managed level before relating them to native runtime frames. The pattern preserves the distinction between a language-level call sequence and the operating system's native stack representation of the same execution.

Ref. R469, Pattern description: Managed Stack Trace (Scala). See also: Managed Code Exception (Scala); Stack Trace [Memory patterns]; Runtime Thread (Python, Linux).

Manifold Memory Space

Memory. A proposed combined view of several virtual memory spaces that makes shared code and data explicit. One physical region can correspond to regions in different process views, including the same virtual range in several processes. We follow those mappings when relating observations across dumps. The geometric name organizes the representation; it does not establish mathematical manifold properties for every memory system.

Ref. R413, Definition and examples. See also: Orbifold Memory Space; Physical memory; Virtual memory; Memory map.

Manual Dump

Memory patterns. A dump deliberately requested during investigation rather than automatically produced by an unhandled failure. Collection-related threads, breakpoints, or comments may explain the context selected when the artifact is opened. We confirm the acquisition history whenever possible and interpret those clues accordingly. Some capture methods leave little distinctive evidence, so the absence of a recognizable collection signature does not establish automatic crash capture.

Ref. R375, Manual-capture indicators and limits of identifying capture origin. See also: Memory dump; False Positive Dump; Self-Dump.

Manual Dump (kernel)

Memory patterns. A kernel or complete dump was deliberately triggered to capture an unresponsive system or another condition of interest. The trigger can produce a bugcheck and stack belonging to the acquisition mechanism. We distinguish that terminal capture event from the earlier problem and use trigger codes, driver frames, and collection history as corroborating evidence.

Ref. R481, Pattern description: Manual Dump (kernel). See also: Manual Dump; False Positive Dump; Dump Context.

Manual instrumentation

Telemetry. Instrumentation added deliberately through telemetry APIs at selected points in application code. It can describe domain operations, record a decision, or attach context that automatic instrumentation cannot infer. The resulting evidence reflects the programmer's chosen boundaries and names. A span around an entire function and a span around one blocking call answer different diagnostic questions even when their labels look similar.

Ref. R251, Code-based instrumentation. See also: Instrumentation; Automatic instrumentation; Span.

Marked Message

Trace patterns. A message distinguished by a selected feature relevant to the investigation. Marking can record activity, its explicit absence, or another semantic distinction. When comparing traces, we preserve the difference between a feature not detected and a feature established as absent; both can otherwise acquire the same visual mark.

Ref. R002, p. 181. See also: Feature of Activity; Message Annotations; Master Trace.

Massif

Program analysis. A Valgrind heap profiler that records memory use over time and attributes selected snapshots to allocation call paths. It helps distinguish a high transient peak from persistent growth and identify allocation sites contributing to a peak. Allocation origin does not necessarily identify the owner retaining memory. Snapshot selection, allocator overhead, and optional stack accounting affect the measured totals.

Ref. R188, Heap profiling; snapshots and allocation trees. See also: Valgrind; Allocation profiling; Retained size.

Master Trace

Trace patterns. An expected trace or model of expected activity used to interpret an observed trace. It may be a recorded working example or an explicitly constructed expectation. A useful master matches the relevant version, configuration, and use case while allowing normal variation rather than requiring identical incidental values.

Ref. R002, p. 182. See also: Bifurcation Point; Equivalent Messages; Marked Message.

MC/DC Modified Condition/Decision Coverage

Testing. A coverage criterion requiring evidence that each condition in a decision can independently affect the decision's outcome, with the applicable treatment of other conditions made explicit. It distinguishes condition influence from merely visiting both decision outcomes. A complete MC/DC result concerns the selected decisions and test executions; it does not establish complete path coverage, correct requirements, or an adequate test oracle.

Ref. R140, MC/DC instrumentation and reports. See also: Branch coverage; Code coverage; Test oracle.

MdTL Metadefect Template Library
Foundations. A proposed library of parameterized defect models in which a behavioral template, such as leaking or overflowing, is combined with a structural context. Its purpose is to support experiments and generative debugging using reusable defect constructions. The name describes the proposal in the cited work and does not imply a currently supported general-purpose library distribution.

Ref. R001, p. 131. See also: Generative debugging; Defect mechanism; DMP.

Measurement
Trace patterns. A quantity obtained from trace content or structure under a stated rule, such as an interval, event count, density, or change rate. The trace is a medium for measurement, not only a narrative. We state the unit, selected population, and relevant observation boundaries so that the result can be interpreted.

Ref. R002, pp. 183–184. See also: Counter Value; Time Delta; Statement Density and Current.

Memcheck
Program analysis. A Valgrind tool that tracks addressability and definedness to report invalid accesses, uses of uninitialized values, and allocation-related problems. Its leak analysis distinguishes reachability classes rather than treating all remaining allocations alike. Reports connect observed operations to available call context. Missing reports do not exclude unexecuted defects or every form of memory misuse, and retained but reachable data needs separate interpretation.

Ref. R185, Error messages; definedness; leak detection. See also: Valgrind; MemorySanitizer; LeakSanitizer.

Memiotics
Foundations. The interpretation of signs and symbolic structures in computer memory. As a branch of memoretics, it asks how bytes, addresses, names, and relationships acquire diagnostic meaning in a particular software context. Recognizing a familiar value or structure is therefore an interpretive step that depends on the model used to read the artifact.

Ref. R001, p. 55. See also: Memoretics; Memory dump analysis; Diagnostic indicator.

Memoretics
Foundations. The study of computer memory snapshots and their evolution, including both normal and abnormal structure and behavior. Memory dump analysis and memory forensics are applications within this broader view. It encourages us to examine what a snapshot represents and how snapshots relate over time before reducing memory analysis to a search for one faulting address.

Ref. R001, pp. 54–56; pp. 133–134. See also: Memiotics; Memory dump analysis; Structural pattern.

Memory acquisition pattern
Artifacts. A reusable, context-dependent approach to obtaining memory evidence. The catalog separates structural scope, such as process or physical memory, from acquisition strategy, such as external, self-triggered, conditional, or sequential capture. We select the scope and strategy together so that the artifact can answer the intended question while preserving the timing and consistency conditions of collection.

Ref. R001, p. 176. See also: Memory dump; Missing Space; Snapshot Collection.

Memory barrier

Execution. An operation that constrains memory ordering according to a defined execution model. It helps establish when one participant's accesses can become visible relative to other accesses. We interpret barriers together with matching synchronization and the data protocol. Adding a barrier at a suspicious instruction does not by itself repair ownership, lifetime, or an incorrect assumption about atomicity.

Ref. R082, API; SEMANTICS; ordering. See also: Atomic operation; Race condition; RCU.

Memory corruption

Memory. An unintended change to memory that violates the expected structure or value of program state. Typical mechanisms include writes outside an object and writes through expired references. We separate the observed damaged bytes from the proposed corrupting operation. A dump usually preserves the result of corruption; additional allocation history, instrumentation, or replay may be needed to reconstruct when and where it occurred.

Ref. R043, Description; R046, Description. See also: Heap corruption; Buffer overflow; Use-after-free.

Memory dump

Artifacts. A captured representation of selected memory and associated execution metadata. Its scope is determined by the acquisition mechanism, options, and conditions at capture time. We use the preserved bytes to reconstruct objects and relationships, while distinguishing missing information from absent runtime state. A dump can be collected from a running or hung system as well as after a crash.

Ref. R064, Types of user-mode dump files; Full user-mode dumps; Minidumps; R422, Definition. See also: Crash dump; Core dump; Memory map.

Memory dump analysis

Artifacts. The extraction and interpretation of software structure and behavior from a captured memory artifact. We impose suitable models to identify processes, threads, modules, objects, and relationships, then use the resulting indicators to form and test explanations. A snapshot can retain traces of earlier execution, but it does not automatically provide a complete execution history.

Ref. R001, p. 37; p. 152. See also: Memoretics; Diagnostic artifact; Analysis pattern.

Memory examination

Debuggers. Reading and displaying target memory as bytes, words, instructions, strings, or typed objects. The selected format is an interpretation applied to the captured bytes. We establish address space, byte order, width, and object lifetime before assigning meaning. An unreadable region may be unmapped, inaccessible, or absent from the dump, while readable bytes can belong to freed or repurposed storage.

Ref. R024, Section 10.6. See also: Memory map; Memory protection; Use-after-free.

Memory Fibration

Memory patterns. A process memory space is associated with external trace or log activity that helps reconstruct its earlier behavior. We align the artifact's process identity and capture time with the relevant activity stream, including the appropriate portion of a combined log. This association extends a static memory observation with historical evidence, provided identities and time relationships remain consistent.

Ref. R559, Pattern description: Memory Fibration. See also: Paratext; Historical Information; Thread Age.

Memory Fluctuation (process heap)
Memory patterns. Process-heap usage increases and decreases across captures, making a persistent leak difficult to distinguish from changing allocation demand. We compare allocation sizes, counts, and call paths over suitable intervals, using an early baseline where available. Short-term differences identify activity; a leak hypothesis requires evidence that relevant allocations remain retained beyond their expected lifetime or continue accumulating.

Ref. R670, Memory Fluctuation (process heap) pattern and diagnostic examples. See also: Memory Leak (process heap); Allocation profiling; Step Dumps.

Memory fragmentation
Memory. A condition in which available storage is divided or rounded in ways that reduce its usefulness for a requested allocation. External fragmentation concerns separated free regions; internal fragmentation concerns unused space within allocated units. We compare requested sizes with allocator and address-space structure. A large total of free bytes does not guarantee that a suitably sized contiguous region is available.

Ref. R014, Private heaps and fragmentation. See also: Heap; Address space; Reserved memory.

Memory Hierarchy (General)
Memory patterns. Layers of memory representation connected by mappings, such as physical memory, virtualized physical memory, and a process's virtual view. The structural pattern also accommodates architectural address layers. We state which layer an address belongs to and follow the relevant translation. This general hierarchy describes representations and mappings; it is broader than a processor cache hierarchy.

Ref. R831, Pattern discussion. See also: Physical memory; Virtual memory; Memory map.

Memory leak
Memory. Allocated memory that remains unavailable for reuse after its effective purpose has ended. In diagnostics, the term concerns retention rather than disclosure of memory contents. We look for growth that persists after comparable work and expected cleanup, then identify the retaining ownership. A single large heap or a temporary rise in resident memory is insufficient to establish a leak.

Ref. R048, Description; Notes. See also: Heap; Working set [Memory]; Use-after-free.

Memory Leak (I/O completion packets)
Memory patterns. I/O completion packets accumulate as kernel-pool allocations. In the illustrated Windows case, the corresponding pool entries also identify the process associated with the growth, allowing the investigation to move from a global resource symptom to a process context. We verify packet population and ownership evidence before treating the largest pool consumer as an unrelated system-wide shortage.

Ref. R471, Pattern description: Memory Leak (I/O completion packets). See also: Insufficient Memory (kernel pool); Object Distribution Anomaly; Resource leak.

Memory Leak (.NET heap)
Memory patterns. Uncontrolled growth of retained objects on a .NET managed heap. Garbage collection can reclaim unreachable objects while a reference chain keeps unwanted objects alive. We compare successive heap statistics and reference relationships, separately from native heap growth. A large but stable heap, or several heaps associated with the runtime configuration, does not by itself demonstrate a leak.

Ref. R594, Memory Leak (.NET heap) pattern and diagnostic examples. See also: Managed heap; GC root; Retained size.

Memory Leak (page tables)
Memory patterns. Memory remains committed to page-table structures beyond the useful lifetime of the corresponding activity. The published example connects retained process objects with persistent page-directory allocations, even after ordinary process memory has disappeared. We correlate page-table usage with process and handle lifecycles and repeated observations; a high page-table count alone can also reflect a legitimate population of address spaces.

Ref. R716, Crash Dump Analysis Patterns (Part 20c). See also: Zombie Processes; Insufficient Memory (physical memory); Memory leak.

Memory Leak (process heap)
Memory patterns. Process-heap allocations remain allocated and accumulate across captures. We first confirm which heap and allocation population grow, then use recorded allocation stacks or retained block contents to investigate their origin. Growth in total process size alone does not establish a heap leak, and an allocation site identifies the producer without necessarily explaining why release did not occur.

Ref. R470, Pattern description: Memory Leak (process heap). See also: Memory Leak (regions); Stack Trace (database); Insufficient Memory (committed memory).

Memory Leak (regions)
Memory patterns. A growing population of virtual-memory regions across successive dumps. The regions may be reserved without substantial committed contents, so ordinary heap-byte totals can miss the growth. We compare region counts, sizes, and addresses and inspect any retained contents for origin clues. The pattern concerns accumulated address-space allocation and can precede other forms of memory exhaustion.

Ref. R595, Memory Leak (regions) pattern and diagnostic examples. See also: Virtual memory; Memory map; Insufficient Memory (reserved virtual memory).

Memory map
Memory. A description of the virtual-memory regions belonging to a process, including address ranges and properties such as access permissions and backing files. It supplies structural context for raw addresses. We use it to place stacks, heaps, and modules before examining individual values. A map captured at a different time may describe a region that has since been released or reused.

Ref. R037, Process-Specific Subdirectories; process memory maps and status. See also: Address space; Virtual address; VMMap.

Memory-mapped file

Memory. A file whose contents are made accessible through a virtual-memory mapping. The mapping connects address ranges with file offsets, but shared and private mappings have different visibility and writeback behavior. We retain the file identity, offset, and mapping mode when interpreting bytes. A value visible in memory may not yet be present in the file on persistent storage.

Ref. R058, DESCRIPTION; MAP_SHARED; MAP_PRIVATE; MAP_ANONYMOUS. See also: Memory map; Copy-on-write; Page cache.

Memory pressure

Memory. A condition in which demand for memory constrains useful execution and prompts reclamation, paging, or allocation failure. The symptom can appear as latency or reduced progress rather than a direct out-of-memory message. We correlate workload, residency, commitment, and reclaim activity. Available-memory counters alone do not describe the cost of maintaining the current workload under the operating system's memory policy.

Ref. R126, Pressure interface. See also: Working set [Memory]; Page fault; OOM killer.

Memory protection

Memory. The access permissions and related mechanisms controlling how memory may be read, written, or executed. They can expose invalid access, but operate at a granularity different from many program objects. We compare the attempted operation with the region's protection and intended object boundary. A writable page can still contain a buffer that the current operation has no right to overrun.

Ref. R012, Copy-on-Write Protection. See also: Page fault; Guard page; Buffer overflow.

Memory Region

Memory patterns. A selected range in a memory space. The structural catalog calls a region open when inspection can extend beyond the chosen range, and closed when unreadable surroundings limit it. We use these topology-inspired terms to describe the observable boundary. Readability in a particular artifact must still be distinguished from allocation, ownership, and access in the original process.

Ref. R829, Pattern discussion. See also: Region Boundary; Memory map; Memory space.

MemorySanitizer

Program analysis. An instrument that detects uses of uninitialized values and can track their origins. It distinguishes whether a value is initialized from whether its address is valid, so it complements address-based checks. Diagnostic interpretation includes the path by which undefined data propagated to the reported operation. Instrumentation coverage and correct treatment of external libraries are essential to the result.

Ref. R178, Introduction; origin tracking; handling external code. See also: AddressSanitizer; Memcheck; DataFlowSanitizer.

Memory Snapshot

Memory patterns. A structural representation of memory at a selected observation point. The catalog distinguishes a structured snapshot, whose headers or formatting support interpretation, from a BLOB snapshot containing raw memory ranges. We retain that distinction when choosing an analysis method: the bytes available and the descriptive structure supplied with them determine which relationships can be recovered.

Ref. R826, Pattern discussion. See also: Aggregate Snapshot; Anchor Region; Memory Region.

Memory space

Memory. A memory-address domain viewed as a linear range and divided into regions. This is the geometric vocabulary used in the earlier Dictionary of Debugging blog entry for describing user, kernel, virtual, and physical memory. We separate the address range from the contents available for inspection: some regions may be inaccessible or absent from the artifact even though their addresses belong to the model.

Ref. R421, Definition and examples. See also: Address space; Memory map; User space; Kernel space.

Memory tagging

Memory. A protection mechanism that compares a tag associated with a pointer against a tag associated with the addressed memory allocation granule. Support and reporting depend on the architecture and operating system. A mismatch can help expose invalid lifetime or spatial access. We preserve tag information during analysis; removing tag bits merely to display an address can discard part of the diagnostic evidence.

Ref. R032, Section 10.7. See also: Use-after-free; Memory protection; CPU register.

Message Annotations

Trace patterns. Notes attached to messages or their parts during analysis. Several annotations may accumulate at different times and can themselves be ordered into an annotation trace. This records interpretation alongside evidence while preserving which wording came from the original artifact and which was added by the analyst.

Ref. R002, p. 185. See also: CoTrace; Explanation Trace; Flag.

Message Bond

Trace patterns. A shared-message connection between message complexes formed through different adjoint activities. The common message or traceme links structures that would otherwise be viewed separately. The bond terminology is borrowed from chemistry and describes an analytical relationship, whose actual meaning comes from the message attributes and context.

Ref. R002, p. 186. See also: Message Complex; Trace Molecule; Traceme.

Message Box

Memory patterns. A thread is waiting inside message-box processing, potentially displaying diagnostic text or blocking progress that other threads depend on. We recover the caption and message from available parameters or stack data and relate the dialog to the incident. The dump can reveal an error report that was never captured in a screenshot, including dialogs outside the user's visible session.

Ref. R732, Crash Dump Analysis Patterns (Part 51). See also: Dialog Box; String Parameter; Blocked Thread.

Message Change

Trace patterns. A change in the variable part of a recurring message, such as a status changing from success to failure. Tracking the message family reveals the transition and directs attention to activity between occurrences. Comparing values requires the same interpretation of the message invariant and its fields.

Ref. R002, pp. 187–188. See also: Message Invariant; Data Association; Abnormal Value.

Message Complex

Trace patterns. A structure built by connecting a selected message with nearby messages sharing thread or adjoint identifiers, then repeating the construction for other identifiers. Shared messages connect the resulting local structures. The simplicial-complex analogy exposes relationships that a simple chronological neighborhood does not capture.

Ref. R002, pp. 189–190. See also: Message Context; Message Bond; Trace Molecule.

Message Context

Trace patterns. Messages surrounding or otherwise related to a selected message under a stated relationship. Context can follow time, thread, data, or another diagnostic association. Choosing it explicitly avoids interpreting an isolated error line while also preventing an arbitrary volume of neighboring output from being mistaken for relevant evidence.

Ref. R002, p. 191. See also: Activity Quantum; Message Complex; Back Trace.

Message Cover

Trace patterns. A relationship in which one adjoint activity covers the intervals or regions of another, suggesting a correlation worth examining. Repeated error activity may, for example, cover gaps in a different activity. The cover organizes comparison; a mechanism is still needed to explain why the observations correspond.

Ref. R002, p. 192. See also: Adjoint Thread of Activity; Discontinuity; Intra-Correlation.

Message Directory

Trace patterns. A multilevel selection of messages used to navigate a large trace. A coarse selection leads to a finer selection and eventually to the surrounding original context. The hierarchy reduces navigation effort while preserving a route back to detail; it is not a replacement for the omitted messages.

Ref. R002, pp. 193–194. See also: Message Set; Message Context; Trace Frames.

Message Embedding

Trace patterns. A vector representation assigned to a trace message, treated as a particular Trace Field. A sequence of such vectors can itself form a derived trace whose columns represent latent Features of Activity. Text Trace interpretation also permits sentence-embedding methods. We retain the embedding model and transformation context because proximity in the vector space is a constructed relationship requiring diagnostic interpretation.

Ref. R448, Part 253. See also: Trace Field; Feature of Activity; Text Trace; Embedding.

Message Essence

Trace patterns. The diagnostically useful part of a message retained for searching comparable cases. Variable identifiers are often removed while stable wording and discriminating constants remain. The appropriate essence depends on the search: removing too little introduces incidental noise, while removing too much collapses unrelated problems into one query.

Ref. R002, p. 195. See also: Message Invariant; Trace Constants; Equivalent Messages.

Message Flow

Trace patterns. An aggregate count of messages selected by one or more activity attributes, such as a process-thread pair or source-destination pair. The pattern extends flow-style summaries to general software narratives. Counts expose distribution and imbalance, but lose the individual order and context needed for some explanations.

Ref. R002, p. 196. See also: Message Metadata; Statement Density and Current; Message Set.

Message Hooks

Memory patterns. Windows message processing passes through an installed hook callback that can influence the receiving thread's behavior. We identify hook-related frames and surviving callback references to connect the activity to its module. Such callbacks may explain a hang, spike, or failure in an otherwise unrelated application, but their participation must be distinguished from ordinary message dispatch and code patching.

Ref. R753, Crash Dump Analysis Patterns (Part 100). See also: Hooked Functions; Internal Stack Trace; Pervasive System.

Message Interleave

Trace patterns. The superposition of different sequences of anchor messages, such as process creation and termination or module loading and unloading. Examining the combined sequence reveals boundaries and relationships that one anchor family alone may conceal. The ordering must still be interpreted according to the trace's recording semantics.

Ref. R002, p. 197. See also: Anchor Messages; Event Sequence Order; Trace Partition.

Message Invariant

Trace patterns. The stable part of messages emitted by the same logging structure, distinguished from variable values such as addresses, versions, and status codes. It supports message-family matching and comparison of variable fields. An invariant can change with software versions, so textual stability is an analytical assumption to check.

Ref. R002, p. 198. See also: Message Change; Message Essence; Abnormal Value.

Message Kinetics

Trace patterns. Relationships among changing message densities observed across traces or within selected regions. By analogy with chemical kinetics, we examine how increases or decreases in one message population accompany changes in another. These relationships can suggest mechanisms for investigation, including during normal execution with different inputs. Unlike Relative Density, the relevant semantic relationship need not already be known.

Ref. R439, Part 244. See also: Statement Density and Current; Relative Density; Activity Region.

Message Metadata

Trace patterns. Attributes associated with a message or measurement that provide dimensions for analysis. Metric labels, for example, can be treated as adjoint identifiers and used to construct activity views. Interpretation depends on label meaning and population, not merely on the fact that two records share the same attribute name.

Ref. R002, pp. 199–200. See also: Adjoint Thread of Activity; Counter Value; Trace Schema.

Message Pattern

Trace patterns. An ordered collection of message invariants within a thread or adjoint activity, used to recognize a corresponding sequence in the same or another trace. It captures more structure than searching for one line. Matching can require attention to omitted records, normal variation, and changes in the emitting software.

Ref. R002, p. 201. See also: Message Invariant; Intra-Correlation; Inter-Correlation.

Message Set

Trace patterns. Messages selected by a predicate or other stated criterion, such as an error class, identifier, or fact from the problem description. Members can become anchors or pivots for further analysis. The set makes selection explicit; any ordering subsequently used should be stated separately.

Ref. R002, p. 202. See also: Case Messages; Anchor Messages; Pivot Message.

Message Space

Trace patterns. A relatively large ordered region or selection that preserves the organization of a trace and serves as a space for further analysis. Kernel and managed-runtime output can form different message spaces in one artifact. Smaller message sets are then extracted without losing their relationship to the containing space.

Ref. R002, p. 203. See also: Interspace; Message Set; Adjoint Space.

Message Type

Trace patterns. A classification of messages by a declared shared property. Messages may have the same metadata value, such as a severity label, the same invariant structure, or membership in the same Trace Schema. We specify the classification criterion before combining or comparing populations. A shared label and a shared structure define different kinds of sameness even when they select overlapping messages.

Ref. R443, Part 248. See also: Message Metadata; Message Invariant; Trace Schema.

Metamorphic testing

Testing. Testing relations between executions when a direct expected answer is unavailable or expensive. An input transformation implies a relation between outputs, such as a permutation preserving an aggregate. A violated relation supplies a diagnostic observation even without knowing either exact answer. The relation must follow from the intended semantics, and satisfying selected relations does not establish overall correctness.

Ref. R159, Source and follow-up cases; metamorphic relations. See also: Test oracle; Differential testing; Property-based testing.

Metaphorical bijectionism

Foundations. A method of inquiry that maps concepts from one domain into another and develops the correspondence to generate new questions and models. In diagnostics, mathematical, physical, and narrative analogies can suggest patterns or analysis operations. The usefulness of the analogy is assessed against artifacts and mechanisms; the name alone does not establish a mathematical bijection.

Ref. R001, pp. 83–85. See also: Diamathics; Software narratology; Analysis pattern duality.

Meta Trace

Trace patterns. A narrative of how a trace and its logging structure evolve across software changes. Component replacement, refactoring, and message-format changes become its subject. This differs from a master trace describing expected execution and from a cotrace recording an analyst's path through existing evidence.

Ref. R002, p. 204. See also: Master Trace; CoTrace; Trace Schema.

Metric

Metrics. A measured or derived quantity used to describe system behavior over a defined scope and interval. A useful metric has a meaning, unit, population, and aggregation rule, not only a name and a number. Its movement can reveal a change that deserves investigation. The same value may have different implications when the workload, collection interval, or resource capacity changes.

Ref. R340, Metrics. See also: Time series; Aggregation; SLI.

Mexel

Narratology. A variable part of a trace message represented as an element in a constructed log image. Addresses, handles, counter values, and other changing fields can form separate layers across message elements. Such layers enable comparisons and anomaly analysis, provided that the encoding preserves the distinctions needed for the diagnostic question.

Ref. R001, pp. 311–312. See also: Loxel; Message Pattern; Trace Constants.

Milestones

Trace patterns. Explicit messages marking higher-level stages of design, architecture, or a use case. They provide recognizable boundaries above the detail of individual implementation operations. A milestone is itself a designed statement, whereas a macrofunction is an analytical grouping of several messages into one semantic operation.

Ref. R002, p. 205. See also: Macrofunction; Anchor Messages; Use Case Trail.

Minidump

Artifacts. A Windows dump format whose streams and collection options determine the information included. The name does not guarantee a small file: a minidump can include extensive process memory and other state. We inspect its actual contents before choosing an analysis path. Failure to read a heap object may reflect omitted memory, even when the recorded thread and exception data are complete.

Ref. R064, Types of user-mode dump files; Full user-mode dumps; Minidumps. See also: Full user-mode dump; Memory dump; Heap.

Minimal reproducer

Testing. A reproducer from which details unnecessary for the selected failure have been removed. Its smaller state and input space make the relevant interaction easier to inspect and test. Minimality must be stated relative to permitted simplifications and the failure predicate; a locally irreducible case may still have a much smaller alternative. Preserve evidence that the reduced case represents the intended problem.

Ref. R154, Minimality and failure-inducing input simplification. See also: Reproducer; Delta debugging; Test-case reduction.

Minimal Trace

Trace patterns. The trace produced by software in its default configuration without additional interaction or input activity. It provides a baseline for understanding the system's own background behavior. Minimal is relative to that configuration and scenario; initialization, timers, discovery, and maintenance can still generate substantial activity.

Ref. R002, p. 206. See also: Master Trace; Use Case Trail; Background and Foreground Components.

Mirror Dump Set

Memory patterns. A set combines a consistent mirrored kernel dump with complementary captures that may contain additional process or system memory. We use each member for the evidence it reliably preserves, such as recovering kernel stacks missing from an inconsistent live capture. The collection improves coverage, but its members must still be related carefully because they need not represent exactly the same instant.

Ref. R805, Crash Dump Analysis Patterns (Part 248). See also: Inconsistent Dump; Stack Trace Surface; No Process Dumps.

Missing Component

Trace patterns. An expected component whose messages are absent from the collected trace. The component may not have loaded or executed, may have ceased to exist, or may not have been selected for tracing. Other artifacts can distinguish these possibilities, including loader evidence and messages from components that depend on it.

Ref. R002, pp. 207–208. See also: Guest Component; No Activity; Visibility Limit.

Missing Component (general)

Memory patterns. An expected executable component or required export is unavailable when the application attempts to use it. Loader or exception information, stack context, and nearby strings can identify the missing dependency. We verify the candidate against the target environment and deployment state, since the apparent failure site may be a delayed-load helper rather than the component whose absence matters.

Ref. R735, Crash Dump Analysis Patterns (Part 59). See also: Missing Component; Hidden Parameter; Module Collection (predicate).

Missing Component (static linking, user mode)

Memory patterns. A load-time dependency cannot be found or initialized, leaving a process in loader-related failure handling. Here static linking denotes an imported dependency required at process initialization. We recover loader exceptions and retained path strings when the selected debugger event is uninformative, then compare the required module with the actual search environment and captured load order.

Ref. R490, Pattern description: Missing Component (static linking, user mode). See also: Missing Component (general); Special Stack Trace; Hidden Exception.

Missing Data

Trace patterns. An expected data value missing from a recorded operation, such as a result that the tracing system fills in only after a synchronous request completes. The absence can direct investigation toward a blocked request or incomplete capture. A related stack or memory snapshot may supply the missing execution context.

Ref. R002, p. 209. See also: Incomplete History; Fiber Bundle; Adjoint Space.

Missing Message

Trace patterns. An expected message absent from the trace under examination. Comparison with an earlier or working occurrence can make the absence diagnostically useful. Its interpretation depends on capture coverage, version, and logging conditions; an unrecorded event and an event that did not occur remain different possibilities.

Ref. R002, p. 210. See also: Incomplete History; Sparse Trace; Master Trace.

Missing Process

Memory patterns. A process required for the expected system function is absent, has exited, or no longer contains the necessary active threads or components. We check identity, service dependencies, startup order, and retained process objects instead of relying only on an image name. An exited object can remain visible, while several services may use the same executable name.

Ref. R628, Missing Process pattern and diagnostic examples. See also: Process; Missing Thread; Coupled Processes.

Missing Space

Foundations. Acquiring an artifact that omits the address space needed to investigate the reported behavior. A kernel-only capture, for example, may not supply the user-space state required to explain a process's computation. We map the question to the necessary capture scope. Unavailable state cannot be interpreted as evidence that the corresponding activity or structure was absent.

Ref. R874, Antipattern discussion. See also: Wrong Dump; User space; Kernel memory dump.

Missing Thread

Memory patterns. The absence of a thread that the application's expected structure or earlier observations lead us to anticipate. Premature termination, failed creation, or changed configuration can leave dependent work without its expected participant. We first establish the expected thread population and the dump's coverage. An absent thread supplies a structural discrepancy; it does not by itself reveal when or why that thread disappeared.

Ref. R359, Expected-thread discussion and thread-exit example. See also: Thread; Main Thread; Stack Trace Collection.

Missing Thread (kernel space)

Memory patterns. An expected kernel thread is absent from the normal active-thread view. We use identifying context, terminated thread objects, exit information, and surviving stack evidence to determine whether the thread has already gone. Its absence may explain a missing service or completion path, but we first distinguish termination from enumeration problems and limitations in the captured dump.

Ref. R717, Crash Dump Analysis Patterns (Part 298). See also: Missing Thread; Hidden Stack; Historical Information.

Mixed Exception

Memory patterns. Managed and native exception mechanisms participate in the same failure, for example when a managed exception wraps a native fault. We inspect inner exceptions, managed frames, and surviving native exception context to reconstruct the relationship. A later breakpoint or reporting path may obscure the earlier event, so the currently displayed exception need not be the original failure.

Ref. R749, Crash Dump Analysis Patterns (Part 90). See also: Managed Code Exception; Hidden Exception; Exception Stack Trace [Memory patterns].

Mock

Testing. A configurable replacement that records interactions and can return values or raise exceptions selected by the test. It helps check whether a component called a dependency with expected arguments and handles selected responses. The mock's behavior is an assumption of the experiment. If that assumption differs from the real dependency, a passing interaction check may support an unrealistic execution.

Ref. R172, Mock objects; calls, return values, and side effects. See also: Test double; Unit test; Test fixture.

Model checking

Program analysis. Systematic exploration of a state-transition model to determine whether specified properties hold. A violation may produce a sequence of states that demonstrates how the model reaches an unwanted condition. This sequence is valuable diagnostic evidence about the model. Applying it to software requires checking that the relevant implementation behavior and environmental assumptions are represented faithfully.

Ref. R147, Verification models, assertions, and temporal properties. See also: Bounded model checking; SPIN; Runtime verification.

Module

Artifacts. A loaded executable component, such as an executable image, shared library, or driver, identified within an execution context. Its address range and build identity help connect raw instruction addresses to code. We distinguish a module’s presence from its involvement in failure. The component containing the current instruction may be processing invalid state created by a different participant.

Ref. R074, General Concepts; headers; sections; Debug Directory; R118, Parameters; verbose module information. See also: Module list; Executable image; Symbol file.

Module Collection

Memory patterns. Loaded modules are collected across processes to compare component presence, versions, vendors, or other properties. A complete memory dump can provide the wider process population, but different execution environments may require different module views. We retain process associations and avoid counting the same process repeatedly. The collection supports environment comparison and discovery of common or unexpected components.

Ref. R663, Module Collection pattern and diagnostic examples. See also: Module list; Same Vendor; Module Collection (predicate).

Module Collection (predicate)

Memory patterns. A module collection selected by a stated condition, such as vendor, function, or semantic relationship. We can use the selected group as a unit in more complex analysis, including dependencies between different module populations. Making the predicate explicit keeps the comparison reproducible and distinguishes a meaningful grouping from an arbitrary list of binaries that happen to be loaded.

Ref. R786, Crash Dump Analysis Patterns (Part 190b). See also: Module Collection; Stack Trace Collection (predicate); Pleiades.

Module Hint

Memory patterns. A module name or recognizable name fragment appears in memory and suggests a component worth investigating. The clue may survive in a corrupted allocation, a string, or other execution residue even when the module is absent from the current stack. We connect the name to surrounding data and product configuration before treating it as evidence of participation or responsibility.

Ref. R644, Module Hint pattern and diagnostic examples. See also: Execution Residue; String Parameter; Module.

Module list

Artifacts. An inventory of loaded or otherwise recorded executable components, often including addresses, paths, build information, and symbol status. It establishes part of the software environment captured by a debugger or report. We use it to check component versions and address ownership. A current list can omit earlier unloaded modules, so historical addresses require additional care.

Ref. R118, Parameters; verbose module information. See also: Module; Build ID; Symbol path.

Module Product Process

Memory patterns. A process belonging to the product associated with a module found elsewhere in a complete memory dump. Module version information, paths, and vendor metadata help locate that related process. We then examine its own activity and dependencies, since a component loaded as a hook or plugin may participate in behavior coordinated by another part of the product.

Ref. R543, Pattern description: Module Product Process. See also: Coupled Processes; Problem Module; Module Variety.

Module Stack Trace

Memory patterns. A stack trace identifies frames mainly by module and offset, with few or no reliable internal function names. The offsets retain positional information that a fully collapsed module sequence would omit. We use the view for comparison while checking reconstruction quality: missing symbols, misleading exported names, or platform unwinding limits can still leave individual frames incorrect or incomplete.

Ref. R804, Crash Dump Analysis Patterns (Part 245). See also: Reduced Symbolic Information; No Component Symbols; Stack Trace Signature.

Module Variable

Memory patterns. A symbol names a module's data variable rather than executable code. Reading the variable can expose counters, configuration, or names that complement higher-level debugger output. We establish its type, units, and scope before interpreting the value, and use symbols appropriate to the captured binary. Resolving an address to a name is only the beginning of that interpretation.

Ref. R652, Module Variable pattern and diagnostic examples. See also: Symbol file; Memory examination; Counter Value.

Module Variety

Memory patterns. A large or heterogeneous collection of loaded components whose combinations enlarge the space of possible interactions. Grouping modules by role, supplier, and expected membership helps us identify unusual dependencies and deployment differences. The resulting map guides further investigation of calls, hooks, and shared resources. Membership in an unfamiliar group, or the presence of many third-party modules, does not itself identify a defect.

Ref. R356, Opening component-variety discussion and module grouping. See also: Module list; Hooked Functions; Not My Version.

Moduli Trace

Trace patterns. The executed source-code statements that give rise to internal logging output or observations made by an external tracing tool. These include explicit logging calls and intercepted operations. The moduli-space analogy relates code to the different traces it can produce, while retaining the distinction between the generating execution and its recorded outputs.

Ref. R002, pp. 211–213. See also: Generative Trace; Declarative Trace; Trace Schema.

Monitoring

Reliability. Repeated collection and evaluation of information about a running system to understand its condition, detect changes, and support operational decisions. Monitoring usually embodies selected questions and expected behavior in measurements, dashboards, and rules. It also supplies evidence for investigations that were not anticipated. Its coverage should be judged against user-visible outcomes and relevant failure modes, not the number of charts or alerts.

Ref. R286, Introduction: purposes of monitoring; Desirable Features of a Monitoring Strategy. See also: Observability; Alert; Dashboard.

Monotonic clock

Performance. A clock whose successive readings do not move backward, suitable for measuring intervals without discontinuous civil-time corrections. Its epoch need not identify a calendar date, and monotonic does not mean perfectly accurate or immune to rate adjustment. Suspend behavior is platform-specific: Linux CLOCK_MONOTONIC excludes suspended time, while CLOCK_BOOTTIME includes it. Choose the clock whose interval semantics match the investigation.

Ref. R398, CLOCK_MONOTONIC; CLOCK_BOOTTIME. See also: Elapsed time; Wall-clock time; Clock skew.

Motif

Trace patterns. A recognizable architectural or behavioral form in trace activity, such as request-response interaction, publication-subscription, or plugin initialization. Motifs connect implementation messages with higher-level organization. One use case or master trace can contain several motifs, and a motif may recur across different products and environments.

Ref. R002, p. 214. See also: Macrofunction; Activity Theatre; Master Trace.

Motivic Trace

Trace patterns. A smaller trace obtained by replacing selected message groups with representative messages or semantic summaries. The groups may express motifs rather than share one simple attribute. This transformation supports analysis at a higher level while requiring a retained correspondence to the detailed evidence that each summary represents.

Ref. R002, p. 215. See also: Motif; Quotient Trace; Collapsed Message.

MTBF Mean time between failures

Reliability. The average duration of normal operation before the next failure under a stated convention. In the cited AWS availability model, the clock starts when normal operation resumes and ends at the next failure; repair time is treated separately. Values depend on what counts as failure and on the observed population. A historical mean does not predict the next failure time for one instance.

Ref. R344, Understanding availability. See also: MTTR; Availability; SLO.

MTTD Mean time to detection

Reliability. The average interval between incident occurrence and its detection under an explicitly chosen convention. Detection may mean a monitor's decision or human recognition; these boundaries should not be mixed. The occurrence time is often reconstructed later and can itself be uncertain. State the population and boundary rules, preserve individual durations, and distinguish improved detection from changes in which incidents were included.

Ref. R346, Recovery and response metrics: Mean time to detection. See also: MTTR; Alert; Event timestamp.

MTTR Mean time to repair / recovery / restore / resolution

Reliability. An ambiguous mean-time metric: repair, recovery, restoration, and resolution name different possible intervals. Using the recovery convention here, measure from service failure to restoration of full operation, then average those durations over a declared incident population. A repair-only clock or a clock extending through preventive follow-up answers a different question. Always state the expansion, start event, end event, population, and treatment of ongoing incidents before comparing MTTR values.

Ref. R345, Mean time to repair; mean time to recovery; mean time to resolve. See also: MTTD; MTBF; Incident.

Multidimensional Message

Trace patterns. A message containing several distinct pieces of information, possibly with different invariants or activity meanings. Splitting it into derived messages can expose separate analytical dimensions. The derived records should retain their common origin and timing so that the split does not create fictitious independent observations.

Ref. R002, p. 216. See also: Message Invariant; Combed Trace; Trace Schema.

Multiple Exceptions (kernel mode)
Memory patterns. Several kernel-mode exceptions or faults visible within the same captured system state. They may involve different processors and differ from nested exception handling on one execution path. We compare trap frames, running threads, and raw stacks, including when a manual crash obscures the earlier activity. The final bug-check code need not describe every fault already present.

Ref. R579, Multiple Exceptions (kernel mode) pattern and diagnostic examples. See also: Multiple Exceptions (user mode); Hidden Exception; Spiking Thread.

Multiple Exceptions (managed space)
Memory patterns. Several managed exception objects are associated with threads in the same captured process. We enumerate CLR threads and inspect their exceptions, including inner exceptions, instead of assuming that the currently selected thread tells the whole story. The collection helps separate concurrent failures, wrappers, and retained exception information; it does not establish that every exception was unhandled.

Ref. R701, Crash Dump Analysis Patterns (Part 1c). See also: Managed Code Exception; Hidden Exception (managed space); Multiple Exceptions (user mode).

Multiple Exceptions (stowed)
Memory patterns. Several saved exception descriptions are referenced by the array associated with a Windows stowed-exception record. An automatic report may display only its first element. We enumerate the array, identify the structure version, and inspect each stored status and stack trace. The visible wrapper and the first reported exception need not identify the complete failure sequence.

Ref. R456, Pattern description: Multiple Exceptions (stowed). See also: Stored Exception; Mixed Exception; Exception Collection.

Multiple Exceptions (user mode)
Memory patterns. Several exception contexts coexist in a process dump, possibly on different threads. The exception selected by automatic analysis can be a debugger break-in while an earlier failure remains on another stack. We inspect the whole Stack Trace Collection and reconstruct the relevant exception contexts before deciding which event explains the reported crash.

Ref. R455, Pattern description: Multiple Exceptions (user mode). See also: Hidden Exception; Stack Trace Collection; Exception.

Mutation testing
Testing. Evaluation of tests by making selected small changes to program behavior and checking whether tests detect them. A detected change is a killed mutant; a surviving mutant may expose a weak test or represent equivalent behavior. Mutation results examine the sensitivity of the tests and their oracles. They do not directly measure the frequency or severity of real production defects.

Ref. R163, Killed, surviving, and equivalent mutants. See also: Test oracle; Code coverage; Regression test.

Mutex
Execution. A synchronization object that grants exclusive ownership to one thread at a time. Other participants wait or fail an acquisition attempt according to the API. We inspect the owner and its dependencies before interpreting the waiting threads. Many waiters can be a normal consequence of serialized work; they become diagnostic evidence when ownership duration or progress differs from the expected protocol.

Ref. R017, Synchronization object types. See also: Critical section; Deadlock; Priority inversion.

Myopic Troubleshooting and Debugging
Foundations. Abandoning an investigative direction after a brief or unsuccessful attempt without establishing what was actually tested. A failed probe can reflect unsuitable instrumentation, scope, or expectations rather than an impossible mechanism. We document the question, observations, and stopping condition so that the next step follows from evidence instead of premature exhaustion of one approach.

Ref. R872, Antipattern discussion. See also: Troubleshooting; Diagnostic hypothesis; Reproducer.

N

Namespace
Memory patterns. A group of API or module names used to infer a component's possible functionality when detailed symbols are unavailable. Names may appear in imports, dynamic-resolution strings, or a collection of modules. We distinguish available capabilities from observed execution and interpret them within the larger application. A legitimate component can also participate in a malicious workflow.

Ref. R841, Pattern discussion. See also: No Component Symbols; String Hint; Module Collection.

Narrachain
Narratology. A proposed integrity structure for software narratives in which message blocks are linked by hashes, potentially with a distributed record. It can make later changes detectable when the relevant hashes and trust conditions are preserved. Integrity protection concerns the recorded narrative; it does not establish that the original messages were truthful or that every event was recorded.

Ref. R001, pp. 287–288. See also: Malnarrative; Software narrative; Diagnostic artifact.

Narralog
Narratology. A proposed declarative language for modeling software narratives and generating traces with specified patterns. Its purpose is to describe the structure and behavior of synthetic trace data for analysis and experimentation. The cited proposal belongs to the author's diagnostic language work; it should not be assumed to be an interoperable trace format or a current supported tool.

Ref. R001, p. 143. See also: Generative Trace; Software narrative; Riemann RCA Pattern Language.

Narreme
Narratology. A proposed basic unit of software narrative whose useful granularity depends on the analysis. A repeated message structure, a small action, or a larger meaningful sequence can serve as a candidate unit. Identifying narremes helps organize a narrative into interpretable parts without assuming that one log line always corresponds to one meaningful computation step.

Ref. R001, p. 142. See also: Message Invariant; Macrofunction; Software narratology.

Near Exception

Memory patterns. Captured execution suggests that continued progress may reach an exception, although the exception has not yet occurred in the observed state. Growing recursive call sequences are one example. We inspect the path and the conditions required for the anticipated failure. The pattern records a supported prospect, with its uncertainty, rather than turning a possible future exception into an observed event.

Ref. R695, Near Exception pattern and diagnostic examples. See also: Stack overflow; Constant Subtrace; Stack Trace Change.

Need the crash dump. Period.

Foundations. Rejecting useful available evidence until the actual dump file is supplied. Partial stacks, logs, raw addresses, module information, and matching symbols may already support a bounded analysis. We state what those artifacts can establish and what additional capture would resolve. Requesting a dump remains appropriate when the unanswered question actually requires state absent from the current evidence.

Ref. R868, Antipattern discussion. See also: Diagnostic artifact; Stack Trace [Memory patterns]; No Component Symbols.

Negative evidence

Foundations. The absence of an expected indicator under collection and interpretation conditions that would make its presence observable. Missing output becomes informative only after considering filters, sampling, buffering, retention, clock context, and artifact completeness. We state the expectation and those conditions explicitly; otherwise, the observation may describe a visibility limit rather than absent execution.

Ref. R001, p. 219; pp. 264–266. See also: Diagnostic evidence; Visibility Limit; Missing Message.

Nested Exceptions (managed code)

Memory patterns. A managed exception retains another exception as its inner exception, potentially forming a chain of wrapped or rethrown failures. We follow the chain and distinguish the original operation from later messages and termination decisions. The outer exception can describe handling policy while the inner object preserves a more specific failure, so inspecting only the last message can lose the useful context.

Ref. R618, Nested Exceptions (managed code) pattern and diagnostic examples. See also: Exception; Managed Code Exception; Translated Exception.

Nested Exceptions (unmanaged code)

Memory patterns. Exception processing contains evidence of another exception raised or processed within it. We inspect dispatcher and handler frames, raw stack records, and the associated contexts to distinguish the nested events. The currently selected exception need not be the earlier one retained in a saved record, and capture timing affects which processing stages remain visible.

Ref. R494, Pattern description: Nested Exceptions (unmanaged code). See also: Multiple Exceptions (user mode); Hidden Exception; Exception Stack Trace [Memory patterns].

Nested Offender

Memory patterns. The component presenting an error or diagnostic message handles an exception raised in another, less immediately visible component. Raw-stack exception residue can reveal the inner activity and redirect investigation toward its origin. We distinguish the reporting component, its handler, and the operation that failed. This nesting concerns attribution across components, rather than an exception handler necessarily raising a second exception.

Ref. R626, Nested Offender pattern and diagnostic examples. See also: Hidden Exception; Execution Residue; C++ Exception.

Network Packet Buildup

Memory patterns. Network packet-related allocations accumulate to a degree that correlates with impaired network progress. We compare packet-pool usage with a normal case and inspect the allocators or drivers associated with the growing population. The pattern identifies retained or outstanding work; counts alone do not determine whether the cause is stalled transmission, a leak, or legitimate workload demand.

Ref. R777, Crash Dump Analysis Patterns (Part 164). See also: Disk Packet Buildup; High Contention (sockets); Object Distribution Anomaly.

Network partition

Distributed systems. A communication failure that prevents some parts of a distributed system from exchanging the messages required by its protocol. Processes on either side may continue running and serving different observations. Partition behavior depends on the consistency and availability guarantees the system attempts to maintain. A timeout can be consistent with a partition, but also with overload, delay, or a failed endpoint.

Ref. R323, Partition tolerance; consistency and availability. See also: Partial failure; Timeout [Distributed systems]; Causal order.

News Value

Trace patterns. The relevance of a trace or trace region to the current problem, considered against its timing, scenario, and accompanying artifacts. A detailed older log may have little value for correlating a later occurrence. The pattern encourages assessment of each artifact before investing in detailed cross-artifact analysis.

Ref. R002, pp. 217–218. See also: Basic Facts; Calibrating Trace; Inter-Correlation.

NMT Native Memory Tracking

Runtimes. A HotSpot facility that accounts for selected categories of memory used by the JVM itself. It helps separate Java-heap use from other runtime memory and compare category changes against a baseline. Its scope does not include every allocation made by third-party native code. Reserved, committed, and process-resident memory describe different quantities and should not be treated as interchangeable totals.

Ref. R207, Native Memory Tracking. See also: Managed heap; jcmd; Heap histogram.

No Activity

Trace patterns. The absence of visible messages from an activity known to have been selected for tracing. A contemporaneous dump may show why the activity is silent, such as threads waiting on an unavailable owner. Selection for tracing strengthens the question but does not eliminate gaps in instrumentation or recording.

Ref. R002, pp. 219–221. See also: Discontinuity; Missing Component; Adjoint Space.

No Component Symbols

Memory patterns. The absence of usable symbolic information for a component that matters to the investigation. Module boundaries, exports, imports, callers, and disassembly may still reveal its role and interactions. We distinguish what these observations establish from what a function name or source mapping would have established. Missing symbols reduce the resolution of the reconstruction; they do not establish that the component is faulty.

Ref. R354, Opening missing-symbol discussion and component-functionality investigation. See also: Symbol file; Public symbols; Disassembly.

No Current Thread

Memory patterns. A dump opens without a usable current-thread context even though other thread information may remain. Commands that depend on the current thread can then fail or mislead. We inspect the available threads and select a suitable context for further analysis, while treating any stored exception separately. Selecting another thread does not recreate a missing exception thread or register context.

Ref. R657, No Current Thread pattern and diagnostic examples. See also: Debugger context; Stored Exception; Missing Thread.

No Data Types

Memory patterns. Available symbols identify code or variables but lack the type information needed to interpret a structure. This is a specific limit on analysis, not necessarily a complete symbol-loading failure. We check symbol coverage and binary identity before seeking compatible type information. An apparently plausible layout from another build can still give a misleading interpretation of memory.

Ref. R645, No Data Types pattern and diagnostic examples. See also: Symbol file; Private symbols; Incorrect Symbolic Information.

Node.js diagnostic report

Runtimes. A structured Node.js report containing selected runtime and process information, such as stacks, heap statistics, versions, and resource state. It can be triggered under configured conditions or requested explicitly. The report is a summary artifact rather than a full object graph or process-memory image. Its trigger, collection time, and version determine how the recorded fields relate to the reported problem.

Ref. R225, Usage; report contents and triggers. See also: Heap snapshot; Node.js Inspector; Thread dump.

Node.js Inspector

Runtimes. Node.js integration with V8's inspector and the Chrome DevTools Protocol for supported debugging and profiling operations. It provides structured access to selected runtime state and execution events. The available view depends on protocol support, connection state, and enabled domains. Inspection or evaluation can affect the target, so observations collected before and after interaction may describe different application states.

Ref. R226, Inspector session; protocol and profiling examples. See also: Heap snapshot; Node.js diagnostic report; Event-loop blocking.

Nonpaged pool

Memory. A Windows kernel memory pool whose allocated contents remain resident in physical memory. It supports data that must be accessible in contexts where servicing a page fault is not permitted. Growth can reduce memory available to other work. We inspect outstanding allocations, tags, and activity over time before deciding whether the growth represents a leak, a backlog, or expected kernel state.

Ref. R101, Paged and nonpaged pool definitions. See also: Paged pool; IRQL; Pool tag.

No Process Dumps

Memory patterns. The absence of an expected process dump despite a reported application failure. The process may have terminated through a different path, remained hung, handled the exception, or encountered a collection problem. We establish the actual process outcome and inspect capture configuration, permissions, and available reporting evidence. No artifact is an observation about the evidence chain, not proof that no failure occurred.

Ref. R379, Opening missing-process-dump discussion and collection-condition examples. See also: Crash dump; Custom Exception Handler; No System Dumps.

No Question

Foundations. Expecting a diagnostic answer without specifying the problem or decision that the analysis should address. Commands and artifacts may accumulate while their relevance remains unclear. We formulate an initial question and refine it as evidence develops. The question need not prescribe a cause; it gives the investigation a purpose and makes the limits of its answer assessable.

Ref. R873, Antipattern discussion. See also: Problem description; Analysis pattern; Diagnostic report.

Normalized Region

Memory patterns. A region's addresses are transformed into a common representation so that comparable memory contents from different executions can be examined together. Relative offsets can remove variation introduced by relocation or ASLR. We document the transformation and distinguish normalized local addresses from other pointers. Normalization should preserve meaningful structure without accidentally making unrelated values appear equivalent.

Ref. R698, Normalized Region pattern and diagnostic examples. See also: ASLR; Memory map; Foreign Stack.

No System Dumps

Memory patterns. The absence of an expected system dump after a system failure or deliberately requested stop. Capture configuration, storage availability, failure severity, and the functioning of the dump-writing path determine whether evidence can be produced. We investigate these prerequisites together with any surviving event or hardware records. An absent dump cannot distinguish all possible failure mechanisms, and collection requirements vary with platform and configuration.

Ref. R380, Opening missing-system-dump discussion and dump-generation prerequisites. See also: Kernel memory dump; Bug check; Truncated Dump.

Not My Thread

Memory patterns. A process contains a thread that the application developer or analyst did not expect their own code to create. Loaded libraries, runtime support, hooks, and other integrations can supply such threads. We identify the thread's origin and role before treating it as suspicious, distinguishing legitimate infrastructure from unexpected participation that could affect the reported behavior.

Ref. R802, Crash Dump Analysis Patterns (Part 239). See also: Runtime Thread (CLR); Message Hooks; Special Stack Trace.

Not My Version

Memory patterns. A component version in the captured environment that differs from the version expected from development, testing, or deployment records. Module identity, paths, and version information can expose an unintended binary or an incomplete update. We verify the actual build and its relationship to the symptom. An older component deserves investigation when unexpected, but age by itself does not demonstrate a defect.

Ref. R389, Opening production-versus-test version comparison and module-path example. See also: Module list; Build ID; Changed Environment.

Not My Version (hardware)

Memory patterns. The failing system has hardware characteristics different from those under which the software was tested or previously worked. Hardware information retained in a kernel or complete dump can reveal processor, chipset, or other configuration differences. We use those differences to refine reproduction and investigation, while avoiding the assumption that a newer or faster machine is itself the cause.

Ref. R617, Not My Version (hardware) pattern and diagnostic examples. See also: Not My Version; Changed Environment; Diagnostic hypothesis.

No Trace Metafile

Trace patterns. Recorded events that cannot be fully interpreted because required decoding metadata is unavailable or mismatched. Some structural observations, including intervals, activity identifiers, and record counts, may remain usable. We separate those observations from guessed message text and obtain the correct metadata where possible.

Ref. R002, p. 222. See also: Recovered Messages; Trace Schema; Message Context.

NULL Pointer (code)

Memory patterns. Execution is transferred through a zero-valued code pointer, such as a missing callback or thread-start routine. The immediate symptom is an attempted instruction fetch at address zero. We inspect the calling instruction, surviving return address, and supplied function argument to identify the transfer. Establishing the null target still leaves the question of how that value reached the call.

Ref. R704, Crash Dump Analysis Patterns (Part 6a). See also: NULL Pointer (data); Function Pointer; Incorrect Stack Trace.

NULL Pointer (data)

Memory patterns. A data access derives its effective address from a null base, often with a small field offset. We reconstruct the instructions and successive dereferences that supplied the base value. This separates the failing field access from the earlier operation that produced or propagated the null pointer, and distinguishes data access from a null control-transfer target.

Ref. R459, Pattern description: NULL Pointer (data). See also: Invalid Pointer (general); NULL Pointer (code); Dereference Nearpoint.

Null pointer dereference

Memory. An attempt to access an object through a pointer representing no object. The immediate evidence may be a read or write near address zero, including an offset to a presumed member. We trace the origin and handling of that pointer before assigning a cause. A failed allocation, missing lookup result, or uninitialized path can produce similar final instructions.

Ref. R049, Description. See also: Access violation; Uninitialized memory; Dangling pointer.

Null Reference

Trace patterns. A reference in message data that cannot be resolved to the expected message or object in the available traces. The target may lie outside the captured interval or artifact. This pattern concerns a failed analytical reference; the recorded value itself need not be a programming-language null pointer.

Ref. R002, pp. 223–224. See also: Linked Messages; Missing Data; Trace Extension.

NUMA Non-uniform memory access
Memory. A memory organization in which access characteristics depend on the processor and memory node involved. Local and remote placement can affect latency and bandwidth. We retain CPU placement, node allocation, and migration context when explaining performance. A sufficient system-wide memory total can hide pressure or imbalance on a particular node, while remote access can increase cost without producing a correctness failure.

Ref. R124, What is NUMA Memory Policy; Memory Policy Concepts. See also: Physical address; Memory pressure; PMU.

O

Object Distribution Anomaly
Memory patterns. An unexpected distribution of objects across types, owners, or processing locations. In the original IRP example, concentration around particular drivers directs attention to where requests accumulate. We compare the observed population with workload, object lifetimes, and a suitable reference distribution. A large count can represent legitimate concurrency or queued work; the anomaly becomes diagnostic when its architectural and temporal context is understood.

Ref. R369, IRP distribution example and author comment dated September 3, 2015. See also: IRP; Memory leak; Inconsistent Dump.

Object Distribution Anomaly (.NET heap)
Memory patterns. An unusual distribution of managed object counts or retained bytes across classes. Large populations associated with one function or component can direct a .NET heap investigation, but temporary queued work can produce a similar distribution. We compare sizes, counts, and relevant observations over time before deciding whether the anomaly represents a leak, a workload peak, or a configuration effect.

Ref. R603, Object Distribution Anomaly (.NET heap) pattern and diagnostic examples. See also: Object Distribution Anomaly; Memory Leak (.NET heap); Memory Fluctuation (process heap).

Object pinning
Runtimes. Preventing a managed object from moving while code requires a stable address, for example during an interaction with native code. Pinning constrains a moving collector and can affect compaction and fragmentation depending on the collector and object placement. A pinned object observed in a dump is therefore a clue about memory layout constraints, not by itself evidence of a leak.

Ref. R194, Determining the number of pinned objects; fragmentation. See also: Managed heap; Garbage collection; Large object heap.

Observability
Telemetry. The ability to investigate a software system through the information it makes available about its behavior and state. We construct a model from traces, logs, metrics, and other artifacts, then test its interpretation. Installing collection tools does not establish that every relevant state can be distinguished. Coverage, context, fidelity, and the questions we can answer matter as much as data volume.

Ref. R249, Observability. See also: Telemetry; Instrumentation; Monitoring.

Observed timestamp

Logs. The time when the collection system observed a log event, represented separately from its source timestamp in OpenTelemetry. The difference between the two can suggest collection delay, buffering, or clock disagreement. It does not isolate any one of these mechanisms without more evidence. Preserving both timestamps helps distinguish late-arriving data from an event that actually occurred later.

Ref. R260, ObservedTimestamp. See also: Event timestamp; Telemetry pipeline; Clock skew.

Observer effect

Performance. A change in the behavior being investigated caused by the act of observing it. Instrumentation can alter timing and execution relationships as well as add measurable cost. In concurrent software, introduced delays can mask synchronization failures. Disappearance under a debugger or profiler is therefore evidence about sensitivity to observation, not proof that the original defect has been removed.

Ref. R402, Abstract. See also: Instrumentation overhead; Instrumentation profiler; Heisenbug.

Off-CPU profile

Performance. An attribution of time during which a thread is not executing on a processor, often grouped by the stack present when it was descheduled. It exposes waiting that CPU sampling misses. Depending on collection and filtering, the interval can include blocking and time awaiting rescheduling. The recorded stack locates a waiting context; it does not alone identify the responsible dependency.

Ref. R393, Introduction; Off-CPU Time; Caveats. See also: CPU profile; Elapsed time; Latency.

OMAP Code Optimization

Memory patterns. Optimized image layout relocates or rearranges code, leaving a difference between symbolic offsets and offsets in the loaded instruction stream. The same apparent function-plus-offset expression can therefore lead to misleading disassembly. We establish the actual address and its mapped code before reconstructing calls or comparing instructions; symbolic proximity alone is insufficient when OMAP transformations are involved.

Ref. R708, Crash Dump Analysis Patterns (Part 5b). See also: Frame Pointer Omission; Incorrect Stack Trace; Reduced Symbolic Information.

One-Thread Process

Memory patterns. A process expected to contain several threads has only one remaining visible thread, sometimes because the others terminated while the survivor became blocked. We inspect that thread's dependencies and the process lifecycle. A single-thread population is meaningful relative to the application's expected behavior and capture time; it is not intrinsically abnormal for every program.

Ref. R789, Crash Dump Analysis Patterns (Part 199). See also: Missing Thread; Hidden Stack; Wait Chain.

OOMKilled
Cloud. A container termination reason reported in Kubernetes when execution was killed for an out-of-memory condition. It is stronger evidence than observing exit code 137 alone, which can result from other forced termination. The reason does not identify why memory demand grew. Examine limits, node conditions, allocation behavior, and the preceding workload; a memory leak is only one possible explanation.

Ref. R326, Exceed a Container's memory limit. See also: Resource limit; Eviction; CrashLoopBackOff.

OOM killer Out-of-memory killer
Memory. A Linux mechanism that selects a process for termination when an out-of-memory condition cannot be resolved through the relevant allocation and reclaim path. Selection is influenced by context and policy, including memory-control-group constraints. We examine the kernel report and the applicable limit before treating the killed process as the source of excessive demand. The victim and the initiating allocation can differ.

Ref. R036, Virtual Memory Primer; Nodes; Page cache; Anonymous Memory; Reclaim; OOM killer. See also: Memory pressure; Memory leak; Kernel panic.

OpenMetrics
Metrics. A specification for exposing metrics with defined names, types, units, samples, and related metadata. It standardizes a representation that monitoring systems can exchange and interpret. Compatibility at the format level does not guarantee that two producers measure the same thing. A diagnosis still requires the metric's operational meaning, collection scope, and aggregation convention, including any exemplars used to connect measurements with executions.

Ref. R333, Data model; metric families; exemplars. See also: Metric; Prometheus; Exemplar.

Operational MCP Tool
AI and ML. A Model Context Protocol tool that provides access to diagnostic artifacts or executes a low-level operation, such as reading memory, listing threads, or extracting a trace interval. It supplies observations on which analysis can operate. We preserve operation parameters and artifact context when using its output to support a diagnostic claim.

Ref. R880, Vocabulary; operational tools. See also: Diagnostic Analysis Pattern Tool; Elementary analysis pattern; Diagnostic artifact.

Opposition Messages
Trace patterns. Messages representing complementary operations, such as allocate and free, enter and leave, or request and return. Pairing them exposes lifetimes and incomplete sequences. An unmatched operation may indicate a problem, but the observation must be assessed against tracing coverage, capture boundaries, and the semantics of the operation.

Ref. R002, p. 225. See also: Incomplete History; Data Reversal; Dialogue.

Optimized Code (function parameter reuse)

Memory patterns. An optimized function reuses storage that previously held one of its parameters. A debugger may consequently display a handle, temporary value, or later result as though it were the original argument. We inspect instructions and value flow around the relevant frame before interpreting those slots, especially when the displayed argument conflicts with the function prototype or observed operation.

Ref. R703, Crash Dump Analysis Patterns (Part 5). See also: False Function Parameters; Frame Pointer Omission; Hidden Parameter.

Optimized-out variable

Debuggers. A source-level variable or argument whose value is not available to the debugger at the selected instruction position because of optimization or insufficient location information. It may have been eliminated, transformed, or retained only in a now-unavailable register. We can sometimes infer constraints from surviving state, but should label that reconstruction as inference. Missing display information does not mean the source operation never occurred.

Ref. R030, Section 10.3. See also: Source mapping; CPU register; Private symbols.

Optimized VM Layout

Memory patterns. An external optimization changes module placement or loading order and exposes behavior that did not occur under the earlier address-space layout. We compare loaded-image addresses and configuration with the expected environment. The relocation is a contextual change to investigate; specific address ranges in the original examples describe those historical Windows systems, not a universal identification rule.

Ref. R491, Pattern description: Optimized VM Layout. See also: Changed Environment; Module Collection; Not My Version.

Orbifold Memory Space

Memory. A proposed view combining virtual and physical memory spaces across multiple computers, extending the Manifold Memory Space model. The earlier Dictionary of Debugging blog entry also calls it cloud memory space. We use the representation to relate separate machine and process views while retaining their identities and mappings. The name is a geometric analogy, not a claim that ordinary distributed memory satisfies a formal orbifold construction.

Ref. R411, Definition and examples. See also: Manifold Memory Space; Memory space; Virtual memory; Physical memory.

Original Message

Trace patterns. The first or otherwise diagnostically distinctive occurrence within repeated messages about an activity. Its vocabulary or context can distinguish the start of an operation from subsequent reports. The pattern requires identifying which occurrence is original for the question being asked, rather than simply selecting the first line in a file.

Ref. R002, p. 226. See also: Message Invariant; Anchor Messages; Gossip.

Origin Module

Memory patterns. A module is a candidate origin of the observed problem, identified from the call path and knowledge of the product. We may follow the trace beyond infrastructure or well-understood functions to find where a problematic operation began. This designation remains a hypothesis. Familiarity with other modules, or their position in the stack, does not establish their correctness.

Ref. R673, Origin Module pattern and diagnostic examples. See also: Top Module; Problem Module; Well-Tested Module.

Ornament

Trace patterns. Formatting or separator material whose meaning comes from the messages it organizes, such as delimiters, prefixes, or region boundaries. An ornament may contain little diagnostic information by itself yet preserve important structure. Removing it during parsing can merge records or conceal the grouping intended by the producer.

Ref. R002, pp. 227–228. See also: Trace Schema; Split Message; Trace Partition.

OTel OpenTelemetry

Telemetry. An open collection of APIs, SDKs, conventions, and components for generating, processing, and exporting telemetry. OpenTelemetry provides a common instrumentation and transport framework across languages and tools. It does not, by itself, provide the storage and investigation capabilities of a complete observability backend. A diagnostic description should identify which SDK, Collector components, and backend actually participated in the data path.

Ref. R249, What is OpenTelemetry?. See also: Instrumentation; Collector; OTLP.

OTLP OpenTelemetry Protocol

Telemetry. The protocol used to exchange OpenTelemetry data between producers, collectors, and backends, with defined encodings and transport bindings. A successful export response describes acceptance at a particular receiving hop. It does not prove that the data is durably stored, queryable, or complete throughout the entire pipeline. Export failures therefore need to be localized to the relevant sender, receiver, and signal.

Ref. R262, Protocol details; delivery guarantees. See also: OTel; Exporter; Telemetry pipeline.

Out-of-bounds read

Memory. A read outside the intended extent of a buffer or object. It may fail immediately, disclose unrelated data, or return bytes that appear usable and corrupt a later computation. We compare the access length and offset with the actual object boundary. Successful address translation and readable page protection establish accessibility; they do not establish that the read belongs to the intended object.

Ref. R044, Description; Common Consequences. See also: Buffer overflow; Memory protection; Memory examination.

Out-of-Module Pointer

Memory patterns. A pointer whose target lies outside the module expected to contain it, as can occur in a redirected dispatch or function table. We compare the target with the table's ownership and documented relationships. The pointer may still resolve to a symbol; symbolic resolution and expected module membership answer different questions about the same value.

Ref. R844, Pattern discussion. See also: Raw Pointer; Patched Code; Hooksware.

Overaged System

Memory patterns. The captured system has a long operational lifetime that makes accumulated software state relevant to the investigation. Uptime can direct attention toward leaks, exhausted counters, or other aging effects, but it does not identify their mechanism. We relate the elapsed lifetime to resource and behavior evidence before recommending a restart as a corrective action.

Ref. R492, Pattern description: Overaged System. See also: Young System; Historical Information; Resource leak.

Overfitting

AI and ML. Learning details of the development data that do not generalize to the intended future population. A diagnostic classifier may recognize report templates, duplicate incidents, or environment identifiers instead of a useful failure relationship. High performance on training examples is therefore insufficient evidence of effectiveness. Evaluation on appropriately separated data and explicit examination of errors help reveal this mismatch.

Ref. R237, Learning model parameters and testing generalization. See also: Cross-validation; Data leakage; Concept drift.

P

Packed Code

Memory patterns. Code transformed into a packed representation that obscures its ordinary image layout or strings until unpacking occurs. Memory acquisition at different execution stages can expose different representations. We examine section structure, readable data, and the reconstructed code in context. Packing is a representation property and a useful analysis lead; its presence alone does not establish malicious behavior.

Ref. R835, Pattern discussion. See also: Pre-Obfuscation Residue; Memory Snapshot; Disassembly.

Packet capture

Networks. Recording network packets visible at a selected interface or observation point, subject to capture filters, buffer capacity, and snapshot length. It preserves concrete protocol evidence from that viewpoint. Absence from a capture does not prove absence from the network: a packet may follow another path, be filtered, or be lost by the capture mechanism. Capture configuration belongs with the artifact.

Ref. R311, Capturing Live Network Data. See also: PCAP; Packet loss; Retransmission.

Packet loss

Networks. Failure of a packet from a defined population to arrive at the required destination within the measurement's waiting threshold. Loss estimates depend on endpoints, packet type, observation interval, and the rule distinguishing loss from very long delay. Missing packets in a local capture are weaker evidence because capture loss and filtering can create similar gaps. Compare endpoint and capture information before locating the failure.

Ref. R314, Metric definitions; methodology. See also: Packet capture; Retransmission; RTT.

Page cache

Memory. Operating-system memory used to cache file contents for file access and memory mapping. Its growth can be useful retained state that is reclaimable under pressure. We distinguish cached file data from anonymous allocations and examine dirty or pinned state before estimating what can be reclaimed. A fall in free memory accompanied by cache growth does not by itself establish a process leak.

Ref. R127, Page-cache definition. See also: Memory-mapped file; Anonymous memory; Memory pressure.

Paged Out Data

Memory patterns. Required virtual-memory contents are absent because the corresponding pages were not resident or were not captured. A missing kernel stack can prevent reconstruction of a relevant thread even though its wait relationship remains visible. We distinguish acquisition coverage from corruption and consider the dump type, paging state, elapsed time, and availability of complementary process captures.

Ref. R500, Pattern description: Paged Out Data. See also: Abridged Dump; Missing Space; Late Crash Dump.

Paged pool

Memory. A Windows kernel memory pool whose contents may be paged out when permitted by their use. Pool allocations belong to kernel components and should be interpreted separately from ordinary user heaps. We correlate pool consumption with allocation tags and driver activity. A high total suggests where to investigate, but does not establish that the component associated with the largest tag has leaked.

Ref. R101, Paged and nonpaged pool definitions. See also: Nonpaged pool; Pool tag; PoolMon.

Page fault

Memory. An exception raised when a memory access requires operating-system intervention under the current page mapping or protection. Many page faults are normal parts of demand paging; others end in an access violation or a signal. The diagnostic question is whether the fault was resolved, and at what cost. A page-fault count alone does not measure disk reads or software defects.

Ref. R013, Working set and hard and soft page faults. See also: Working set [Memory]; Access violation; Memory protection.

PageHeap

Native tools. Windows heap-verification facilities configured through tools such as GFlags. Standard verification checks allocation boundary patterns; full-page verification places inaccessible memory at a boundary to expose certain overruns closer to the offending access. We select the mode before reproducing the problem. Its changed allocation layout and memory overhead can alter behavior, and the protected boundary does not detect every possible lifetime or bounds error.

Ref. R068, Standard and full-page heap verification. See also: Heap corruption; Buffer overflow; Guard page.

Page table

Memory. A structure used by the memory-management hardware and operating system to describe virtual-memory mappings and associated access properties. Its entries help explain whether a virtual address can be translated and how it may be accessed. We interpret an entry according to the processor architecture and operating-system state; a raw entry value has no universal meaning across platforms.

Ref. R036, Virtual Memory Primer; Nodes; Page cache; Anonymous Memory; Reclaim; OOM killer. See also: Virtual address; Page fault; Memory protection.

Pairwise testing

Testing. Combinatorial testing in which every allowed pair of factor values appears in at least one test. It provides a specific interaction-coverage guarantee for the chosen model and constraints. It does not cover every three-way or larger interaction. During diagnosis, a passing pairwise suite cannot rule out a defect that requires several settings to occur together.

Ref. R171, Pairwise coverage and higher-strength interactions. See also: Combinatorial testing; Test oracle; Reproducer.

Palimpsest Messages

Trace patterns. Messages whose contents have been partly erased or overwritten. Surviving fragments, invariants, and surrounding messages may permit partial reconstruction. The result must distinguish bytes actually preserved from content inferred through context, especially when circular storage or corruption has altered the original record.

Ref. R002, pp. 229–230. See also: Recovered Messages; Corrupt Message; Circular Trace.

Parameter Flow

Memory patterns. A parameter value is followed through calls to investigate where a problematic input originated or how it was transformed. We combine stack frames, instructions, and surviving register or memory evidence. Calling conventions and optimization limit what remains recoverable. Matching values can suggest a path, but apparent parameters or stale stack contents require validation before we assert provenance.

Ref. R678, Parameter Flow pattern and diagnostic examples. See also: Hidden Parameter; False Function Parameters; Data Flow.

Paratext

Memory patterns. Logs, crash reports, memory maps, and other accompanying material extend what a memory dump alone can explain. We use this surrounding evidence to recover module paths and versions, memory-region attribution, exception details, or application diagnostics. Its value depends on provenance and correspondence with the dump: a report from another execution can introduce a misleading context.

Ref. R536, Pattern description: Paratext. See also: Embedded Comment; Historical Information; Module Variety.

Partial failure

Distributed systems. A condition in which some components, requests, or communication paths fail while others continue to operate. It makes success observed at one location insufficient to establish system-wide health. A service can respond to one client and remain unreachable to another. Diagnostics must therefore preserve the affected population and viewpoint, and separate unavailable communication from evidence that the remote component itself has stopped.

Ref. R324, Introduction; system assumptions. See also: Network partition; Availability; Timeout [Distributed systems].

Passive System Thread (kernel space)

Memory patterns. A Windows system worker thread is waiting for work in its expected kernel-only role. We compare its stack with the worker or logging activity it implements and with appropriate reference traces. Departure from the expected passive state can direct further investigation, but a wait in a system thread is not sufficient evidence of a hang.

Ref. R479, Pattern description: Passive System Thread (kernel space). See also: Passive Thread; Special Process; Stack Trace Collection.

Passive Thread

Memory patterns. A thread whose observed wait is consistent with its normal role of receiving work, input, or notification. Its inactivity can be useful evidence about which parts of an application are not currently processing requests. We interpret the wait within the application's protocol and resource ownership. A familiar waiting function alone cannot establish passivity, because a thread may wait while holding resources needed elsewhere.

Ref. R365, Opening passive-thread classification and waiting-stack examples. See also: Blocked Thread; Main Thread; Stack Trace Collection.

Pass Through Function

Memory patterns. A function mainly forwards a request to another layer, as with a driver routine relaying an IRP. Its appearance in a stack identifies participation in the path but may explain little about the delay or failure. We inspect the forwarding behavior and the surrounding operation to find where ownership, waiting, or substantive work actually changes.

Ref. R746, Crash Dump Analysis Patterns (Part 83). See also: IRP; Top Module; Directing Module.

Past Process

Memory patterns. Residual structures or mapped-image information suggest a process that existed earlier but is no longer active. We examine retained process objects, image paths, and file-control structures for troubleshooting context. These clues differ from a currently hidden process, and a surviving mapping record alone requires corroboration before we infer the exact process lifetime or activity.

Ref. R572, Pattern description: Past Process. See also: Hidden Process; Historical Information; Execution Residue.

Past Stack Trace

Memory patterns. A previous call path is reconstructed from return-address and other execution residue left in stack memory. We distinguish it from the active frames displayed at capture time and validate candidate calls before inferring earlier activity. The recovered path is historical evidence with uncertain completeness, since later execution may overwrite some regions while leaving older fragments intact.

Ref. R794, Crash Dump Analysis Patterns (Part 214). See also: Execution Residue; Rough Stack Trace (unmanaged space); Function Pointer.

Patched Code

Memory patterns. Executable code modified relative to its expected image, either through an in-place change or through redirection to another location. We compare relevant bytes and reconstruct the resulting control flow, then identify the mechanism and component responsible. The modification can support legitimate instrumentation or other purposes; its diagnostic significance depends on what changed and how execution uses it.

Ref. R842, Pattern discussion. See also: Hooksware; Hooked Functions; Out-of-Module Pointer.

Path condition

Program analysis. A logical constraint describing the input values required to follow a selected execution path. Symbolic execution accumulates conditions at decisions and uses a solver to check feasibility or generate concrete inputs. An unsatisfiable condition excludes that path under the modeled semantics. A satisfying assignment is a candidate test, whose behavior must still be checked against the actual execution environment.

Ref. R148, Symbolic execution and conditional branches. See also: Symbolic execution; Concolic execution; Path explosion.

Path explosion

Program analysis. Rapid growth in the number of execution alternatives an analysis must consider as decisions, loops, and interactions accumulate. Symbolic execution can therefore exhaust time or memory while exploring only part of a program. Search strategies, bounds, and approximations make the task manageable, but change what has been examined. An unfinished search must not be interpreted as an absence of violating executions.

Ref. R148, Symbolic execution trees and path alternatives. See also: Symbolic execution; Bounded model checking; KLEE.

Pattern-based diagnostics

Foundations. The activity of developing and maintaining the pattern knowledge used in diagnostic work. It includes discovery, description, validation, integration, packaging, revision, and feedback from cases. Its focus is the quality and evolution of the catalog, while pattern-driven analysis applies available patterns to the artifacts of a particular investigation.

Ref. R001, p. 160; p. 219. See also: Pattern-driven diagnostics; Pattern-oriented diagnostics; PVC.

Pattern-driven diagnostics

Foundations. The application of an existing pattern catalog to an investigation, with explicit attention to patterns found, not found, or requiring more evidence. Repeated passes through artifacts and patterns refine the diagnosis. The catalog guides observation and collection; familiarity with a named pattern does not remove the need to establish its indicators in the current case.

Ref. R001, p. 219. See also: Pattern-based diagnostics; Pattern-oriented diagnostics; Pattern repertoire.

Pattern language

Foundations. An organized vocabulary of related patterns that can be combined to describe problems, analysis methods, and solutions. Relationships among patterns allow an investigation to move from one observation to another instead of treating each catalog entry in isolation. A useful language preserves context, applicability, and distinctions among analysis, design, implementation, and usage patterns.

Ref. R001, p. 129; pp. 231–233. See also: Pattern repertoire; Analysis pattern; Problem pattern.

Pattern narrative

Narratology. A narrative whose meaningful units are recognized diagnostic patterns rather than the original individual messages or bytes. It compresses the artifact into an account of structure and behavior at the chosen level of analysis. The mapping back to supporting evidence matters, because the narrative may otherwise conceal uncertainty or alternative interpretations behind familiar pattern names.

Ref. R001, pp. 202–205. See also: Analysis narrative; Motivic Trace; Software narrative.

Pattern-oriented diagnostics

Foundations. An approach to identifying and interpreting signs of software structure and behavior through reusable analysis and problem patterns. It combines applying catalogs to cases, maintaining those catalogs, and examining relationships across artifacts and system boundaries. Recognized patterns organize the evidence and guide investigation of the mechanisms that could explain it.

Ref. R001, p. 219; pp. 264–266. See also: Pattern-driven diagnostics; Pattern-based diagnostics; Systemic diagnostics.

Pattern repertoire

Foundations. The patterns an analyst can recognize, interpret, and use effectively. Passive familiarity with names differs from active ability to find indicators and combine patterns in real artifacts. A repertoire includes relationships among patterns, their meanings, and their contextual use, and can be extended when an analyst moves to another platform or diagnostic domain.

Ref. R001, p. 229. See also: Pattern language; Pattern-driven diagnostics; Trace Skill.

Pattern Vector

Trace patterns. A vector whose components describe selected Message Patterns within a fixed interval or other declared analysis unit. Components may count occurrences, including selected Silent Messages, or carry densities or Trace Field values. Repeated vectors expose how the chosen pattern population changes. We retain the component definitions and Time Scale so that different vectors remain comparable.

Ref. R426, Part 231. See also: Message Pattern; Time Scale; Trace Field.

PCAP

Networks. A widely used packet-capture file format associated with libpcap and tcpdump. Packet records preserve capture timestamps, captured and original lengths, and packet bytes under a declared link-layer type. Snapshot limits can truncate the bytes available for interpretation. A PCAP file is a record of what the capture system retained; it is not a complete account of every packet exchanged by the endpoints.

Ref. R313, File header; per-packet headers; snapshot length. See also: Packet capture; Event timestamp; Telemetry loss.

PDB Program Database

Artifacts. A debug-information format widely used with Windows binaries. A PDB can contain symbol, type, source, and compilation information arranged in multiple streams. We match it to the precise binary being examined before interpreting fields or locations. Possessing a PDB with the expected filename does not establish that it describes the loaded build or contains the private information an analysis requires.

Ref. R075, Introduction; File Layout; Streams. See also: Symbol file; Private symbols; Build ID.

pdb

Runtimes. Python's interactive debugger, supporting inspection of execution frames and postmortem examination after an exception. It provides a concrete view of names and control state for a selected context. The view can be changed by evaluation, continued execution, or entering another frame. Debugger interaction therefore belongs in the diagnostic history, especially when reproducing timing-sensitive behavior or inspecting objects with active evaluation behavior.

Ref. R220, Debugger capabilities; postmortem debugging. See also: faulthandler; Reproducer; Assertion.

PE Portable Executable

Artifacts. The executable-image format used by Windows for components such as programs and libraries. Its headers and sections describe architecture, loading, imports, exports, and other information relevant to reconstruction. We use these structures to interpret image boundaries and addresses. A PE timestamp is not always a trustworthy build date, particularly when special values or reproducible-build metadata are involved.

Ref. R074, General Concepts; headers; sections; Debug Directory. See also: Executable image; PDB [Artifacts]; Module.

Percentile

Metrics. A quantile expressed as a percentage. A reported 99th percentile places a boundary near the upper one percent of the specified distribution, subject to the estimation convention and available observations. It is not a maximum and does not describe the magnitude of every slower observation. For histogram-derived percentiles, inspect the bucket layout and population before treating small changes as exact measurements.

Ref. R273, histogram_quantile. See also: Quantile; Tail latency; Histogram.

perf

Native tools. The Linux performance-analysis tool suite built around performance events, with facilities for counting, sampling, recording, and reporting supported events. We select the event, scope, and sampling configuration to answer a specific question about execution. A sample profile estimates where the chosen event occurred; it does not directly measure every kind of waiting or establish the causal source of an application-level delay.

Ref. R135, DESCRIPTION; SEE ALSO command references; R057, DESCRIPTION; counting and sampled events. See also: PMU; Stack trace [Artifacts]; ftrace.

Periodic Error

Trace patterns. An error or status repeatedly observed in a trace. Repetition can reveal retries, recurring environmental conditions, or normal implementation behavior such as buffer-size probing. We examine the interval, context, and corresponding working activity before treating frequency as evidence that the error explains the reported symptom.

Ref. R002, pp. 231–232. See also: False Positive Error; Error Distribution; Time Delta.

Periodic Message Block

Trace patterns. A recurring block of similar messages, extending periodicity beyond one error or one line. Such blocks can show repeated checks, retries, or background cycles. Comparing complete blocks, their intervals, and their internal variations can reveal a transition that a count of individual messages would conceal.

Ref. R002, p. 233. See also: Periodic Error; Sequence Repeat Anomaly; Drone Message.

Pervasive System

Memory patterns. A collection of cooperating components operates throughout the environment, for example through injected modules, hooks, configuration, and interprocess communication. We consider the system formed by these components when individual stacks do not explain the incident. Its broad interactions can influence host applications in unexpected contexts, making component relationships and controlled isolation useful parts of the investigation.

Ref. R756, Crash Dump Analysis Patterns (Part 111). See also: Ubiquitous Component (user space); Coupled Modules; Coupled Processes.

Phantom Activity

Trace patterns. Recorded activity that would normally require an external action or condition, although that trigger is not known to have occurred. It can expose stale or incorrectly updated internal state. We also check the completeness of the trigger evidence before concluding that the activity arose without a corresponding external event.

Ref. R002, p. 234. See also: Message Context; State and Event; Defamiliarizing Effect.

Phase Transition

Trace patterns. A marked qualitative or quantitative change in trace behavior, such as a sudden change in message rate, activity population, or continuing message families. The pattern locates a transition and motivates a search for the relevant parameter or mechanism. The physical terminology is an analogy for the observed change.

Ref. R002, pp. 235–237. See also: Critical Point; Trace Acceleration; Activity Divergence.

Physical address

Memory. An address in the physical address space used to identify memory or a mapped device resource. It is distinct from the virtual address used by a process. When examining a system dump, a successful virtual-to-physical translation identifies backing storage within the captured state; it does not establish the logical type, ownership, or lifetime of the bytes found there.

Ref. R036, Virtual Memory Primer; Nodes; Page cache; Anonymous Memory; Reclaim; OOM killer. See also: Virtual address; Page table; Memory dump.

Physical memory

Memory. Memory considered through its physical implementation and the addresses used to identify its units. The earlier Dictionary of Debugging blog entry presents an idealized linear numbering of those units. We keep that physical view separate from the virtual views presented to processes, and identify which physical regions a captured artifact actually contains before treating its bytes as a complete picture.

Ref. R418, Definition and examples. See also: Physical address; Virtual memory; Complete memory dump; Memory space.

Piecewise Activity

Trace patterns. An activity represented by successive pieces belonging to different threads, processes, or components. A client phase followed by a server phase is one example. The pieces can reflect a dependency chain, but succession alone is insufficient: communication and identity evidence establish how the pieces belong together.

Ref. R002, pp. 238–239. See also: Coupled Activities; Use Case Trail; Discontinuity.

Pivot Message

Trace patterns. A message outside the currently selected activity that provides a reference point for returning to that activity at a useful position. A problem message in the full trace can indicate where to inspect an already filtered adjoint thread. The pivot connects views without requiring the message to belong to both.

Ref. R002, pp. 240–243. See also: Adjoint Thread of Activity; Message Set; Back Trace.

Place Trace

Memory patterns. Memory evidence is organized as a trace over places, such as addresses, pages, modules, or process regions. Values and memory attributes supply the observations, while spatial relationships supply the ordering. We may recover temporal clues from the associated containers, but spatial order is not execution order. This distinction allows trace methods to be applied to memory evidence carefully.

Ref. R680, Place Trace pattern and diagnostic examples. See also: Execution Residue; Value References; Memory map.

Platform-Specific Debugger
Memory patterns. The available debugger or extension does not support the target architecture or execution environment required for a particular analysis. We verify the compatibility requirements and select a suitable debugger configuration before interpreting failed commands as missing evidence. The original examples concern specific historical Windows and CLR combinations; the applicable restriction must be checked for the actual tool and target versions.

Ref. R750, Crash Dump Analysis Patterns (Part 93). See also: Virtualized Process (WOW64); Version-Specific Extension; Managed Stack Trace.

Pleiades
Memory patterns. A cluster of modules or processes serves a similar function, such as several print drivers or equivalent services in different sessions. Any member may be relevant to the reported problem, so we treat the cluster as an investigative population. Comparing members helps organize candidate components without assuming that their shared purpose makes them equally responsible for the failure.

Ref. R769, Crash Dump Analysis Patterns (Part 143). See also: Module Collection (predicate); Module Variety; Coupled Modules.

PLOT Program Lines of Trace
Narratology. A way of relating trace message structures to the program constructs that could have generated them. Repetition, nesting, choices, and ordering in output can suggest corresponding source-level behavior. The reconstruction is constrained by instrumentation and missing information: several different implementations may generate a similar visible trace.

Ref. R001, pp. 118–119. See also: Declarative Trace; Causal model; Software narratology.

PMU Performance monitoring unit
Native tools. Processor facilities for counting or sampling hardware events, such as selected instruction, cycle, or cache-related activity. The available events and their meanings depend on the processor and monitoring interface. We record the event definition and measurement scope with the result. Multiplexing, attribution limitations, and workload changes can affect interpretation, so similarly named counters from different processors are not automatically equivalent evidence.

Ref. R057, DESCRIPTION; counting and sampled events. See also: perf; CPU register; Context switch.

Pod event
Cloud. A Kubernetes event associated with a Pod, reporting observations such as scheduling, image pulling, container startup, probe failure, or restart backoff. Events provide control-plane and node context that application logs may lack. They are a diagnostic stream with retention and aggregation behavior, not a guaranteed complete execution history. Correlate their reasons and times with Pod status and the relevant container instance.

Ref. R294, Examining Pod status; events. See also: Container log; Termination message; Eviction.

Poincaré Trace
Trace patterns. A cross-sectional trace formed by fixing a selected coordinate, such as time, and taking corresponding or latest preceding messages from several activities. It exposes a composite view at that section. The view should retain the ages of preceding observations, which may differ considerably across activities.

Ref. R002, pp. 244–245. See also: Activity Packet; Adjoint Thread of Activity; Trace Dimension.

Pointer Class

Memory patterns. A pointer is classified by the kind of memory region it references, such as stack or heap. We compare that class with the intended ownership and lifetime of the referenced data. A supposed persistent pointer that actually points into a temporary stack variable can expose a lifetime mistake even when the address is readable and the immediate object contents look plausible.

Ref. R812, Crash Dump Analysis Patterns (Part 270). See also: Memory Region; Use-after-free; Hidden Parameter.

Pointer Cone

Memory patterns. The neighborhood of memory around an address obtained by interpreting a value as a pointer. We expand the inspected range on either side to examine adjacent fields, pointers, and data instead of stopping at one dereferenced value. The surrounding structure can clarify the pointer's role, provided the address and the chosen interpretation are valid.

Ref. R565, Pattern description: Pointer Cone. See also: Value References; Wild Pointer; Corrupt Structure.

Polytrace

Trace patterns. A network of trace continuations from shared execution beginnings or restored states. Different input or environmental choices can produce alternative continuations for comparison. The shared prefix makes divergence easier to study, provided the restored state and the effects of external systems are sufficiently controlled.

Ref. R002, p. 246. See also: Bifurcation Point; Master Trace; Serial Trace.

PoolMon

Native tools. A Windows utility for tracking kernel pool allocation statistics by pool tag. Comparing outstanding allocation counts and bytes can reveal categories that grow during a workload. We associate suspicious tags with their possible allocating components and investigate lifetime further. A growing tag provides evidence of accumulation; it does not by itself establish which individual allocation is leaked or which code path failed to release it.

Ref. R093, Use PoolMon; Display driver names in PoolMon. See also: Pool tag; Paged pool; Nonpaged pool.

Pool tag

Memory. A label associated with a Windows kernel pool allocation to support accounting and investigation. Tools can group outstanding allocation counts and bytes by tag, helping us move from a system-wide symptom toward particular allocation paths. The tag is an investigative clue, not a unique causal identity. Shared tags, changing workloads, and pending work can complicate attribution to a driver.

Ref. R093, Use PoolMon; Display driver names in PoolMon. See also: PoolMon; Paged pool; Nonpaged pool.

Postcondition

Program analysis. A condition an operation promises when it completes under the contract's stated assumptions. It may relate the resulting state to the state at entry. A failed postcondition identifies a broken guarantee, but several earlier actions may have contributed to it. Exceptional completion and partial effects need their own specified treatment; ordinary successful-return guarantees should not be silently extended to them.

Ref. R142, Postconditions and supplier guarantees. See also: Contract; Precondition; Invariant.

Postmortem

Reliability. A structured account of an incident, its impact, the response, contributing conditions, and actions for improvement. It reconstructs what happened from available artifacts and participant knowledge, including uncertainty and missing evidence. A useful account distinguishes observations from later interpretation and evaluates decisions using information available at the time. Its corrective actions should address mechanisms and response weaknesses that the evidence supports.

Ref. R282, Postmortem philosophy; blameless postmortems. See also: Incident; Incident commander; Runbook.

pprof

Runtimes. A profiling format and analysis toolchain commonly used with Go runtime profiles. Depending on the profile, it can attribute CPU samples, allocations, in-use memory, blocking, or other events to call contexts. The selected sample type defines the quantity being compared. A large cumulative allocation total, for example, does not imply that the same amount of memory remains live.

Ref. R230, Overview; built-in profiles and sample meanings. See also: Allocation profiling; Goroutine profile; Go execution trace.

Precision

AI and ML. The fraction of predicted-positive cases labeled positive in the evaluation set. For a diagnostic detector, it describes how often selected diagnostic cases match the reference category. Threshold, class prevalence, and label quality affect interpretation; high precision alone says little about positive cases the detector missed.

Ref. R234, Precision, recall, and F-measures. See also: Recall; Confusion matrix; F1 score.

Precondition

Program analysis. A condition a caller must satisfy before an operation can rely on its specified guarantees. It separates an invalid request from an implementation that mishandles a valid request. During diagnosis, inspect the arguments and relevant state at the call boundary. Checking the condition only after the operation may confuse an earlier violation with changes made by the operation itself.

Ref. R142, Preconditions and caller obligations. See also: Contract; Postcondition; Assertion.

Premortem diagnostics

Foundations. Preventive analysis of artifacts collected before a reported failure, particularly periodic memory snapshots. The aim is to identify structural or behavioral indicators that deserve investigation while the system still appears functional. The term concerns proactive artifact analysis here; it is distinct from a planning exercise in which a team imagines a future project failure.

Ref. R001, pp. 171–172. See also: Zero Fault diagnostics; Memory dump analysis; Diagnostics-driven development.

Pre-Obfuscation Residue

Memory patterns. Fragments of original code or data that remain accessible despite subsequent packing or obfuscation. Strings, identifiers, and other remnants can provide leads into a transformed module. We relate each fragment to its location and surrounding representation before drawing conclusions. Residue is selective evidence of earlier content, not a complete reconstruction of the original program or its behavior.

Ref. R836, Pattern discussion. See also: Packed Code; String Hint; Execution Residue.

Priority inheritance

Execution. A scheduling mechanism that temporarily raises a resource holder's effective priority to help a blocked higher-priority participant make progress. It addresses particular priority-inversion paths according to the synchronization implementation. We distinguish configured priority from effective priority when analyzing scheduler evidence. Inheritance does not remove the underlying dependency or make an indefinitely blocked owner able to complete its work.

Ref. R042, Unbounded Priority Inversion; Priority Inheritance. See also: Priority inversion; Mutex; Context switch.

Priority inversion

Execution. A situation in which a higher-priority task is delayed by a lower-priority task whose resource it needs. Intermediate-priority work can prolong the delay by preventing the resource holder from running. We reconstruct ownership and scheduling together. Looking only at the blocked high-priority thread misses the lower-priority activity whose completion would allow the important work to proceed.

Ref. R042, Unbounded Priority Inversion; Priority Inheritance. See also: Priority inheritance; Mutex; Starvation.

Private symbols

Artifacts. Debugging information that includes details beyond a public symbol table, such as local variables and type information, according to the platform and build. On Windows, these details can be necessary for particular object and variable inspections. We describe missing capabilities explicitly when only public data is available. An analysis command that fails for lack of private symbols has not disproved the object's existence.

Ref. R104, Private symbol data and public symbol table. See also: Public symbols; PDB [Artifacts]; Symbol file.

Problem description

Foundations. A narrative of the observed deviation, relevant environment, expected behavior, and circumstances in which the problem occurs. It helps determine which artifacts to collect and how to interpret their time and scope. A useful description separates observation from explanation so that an early assumption does not exclude relevant diagnostic evidence.

Ref. R001, pp. 153–154; p. 264. See also: Elementary diagnostic pattern; Diagnostic report; Extended software trace.

Problem description pattern

Foundations. A recurrent form of information needed to identify a problem and direct its investigation. Such patterns help elicit context, symptoms, affected components, and conditions without prematurely assigning a cause. They connect the reported experience to artifact collection, diagnostic analysis, troubleshooting suggestions, and the choice of further debugging actions.

Ref. R001, p. 154. See also: Problem description; Diagnostic artifact; Problem pattern.

Problem Exception Handler

Memory patterns. The exception handler contributes to the failure through its own defect or an invalid response to the exception being processed. We compare the handler's call path with the exception's permitted processing behavior and look for a secondary fault or characteristic recurrence. This separates the original event from a complication introduced by the attempted recovery or reporting.

Ref. R511, Pattern description: Problem Exception Handler. See also: Custom Exception Handler; Nested Exceptions (unmanaged code); Error Reporting Fault.

Problem Module

Memory patterns. A component whose behavior can interfere with the application's expected operation, such as a module that hooks functions or changes memory management. We identify its presence and role from process, module, and system inventories, then evaluate an appropriate troubleshooting hypothesis. Inclusion on a list of suspect components is contextual evidence, not proof that this particular incident originated there.

Ref. R530, Pattern description: Problem Module. See also: Hooked Functions; Ubiquitous Component (kernel space); Effect Component.

Problem pattern

Foundations. A recurrent configuration of signs or symptoms that identifies a diagnostic problem in context. It describes what can be recognized in the evidence and may have general and concrete forms. Its corresponding analysis patterns explain how to find it, while defect mechanism patterns describe processes that could have produced it.

Ref. R001, pp. 216–219; p. 352. See also: Analysis pattern; Defect mechanism; General problem pattern.

Problem Vocabulary

Memory patterns. A set of words and phrases used to search textual debugger output for potentially relevant activity, such as termination, suspension, exceptions, or particular waits. We build the vocabulary for the target and incident, then inspect each match in context. It accelerates navigation through large dump-derived logs while allowing for false positives and terminology that changes across versions.

Ref. R770, Crash Dump Analysis Patterns (Part 146). See also: Vocabulary Index; Stack Trace Collection; Wait Chain.

ProcDump

Native tools. A Sysinternals utility that captures Windows process dumps manually or when configured conditions occur, such as an exception or a performance threshold. It can preserve state near a symptom that is difficult to observe interactively. We choose triggers and dump content according to the question being investigated. A triggered capture establishes that its collection condition occurred, while the resulting dump still requires independent interpretation.

Ref. R072, Introduction; Using ProcDump. See also: Memory dump; First-chance exception; Full user-mode dump.

Procedure Call Chain

Memory patterns. A logical procedure-call chain is reconstructed by joining stack fragments from communicating threads or processes and removing intervening middleware subtraces. Wait relationships provide the connections across the separate stacks. We preserve the evidence for each join and the omitted infrastructure. The result explains a wider operation; it is not the physical stack of a single thread.

Ref. R690, Procedure Call Chain pattern and diagnostic examples. See also: Wait Chain (LPC/ALPC); Coupled Processes; Glued Stack Trace.

!process

Debuggers. A Windows kernel-debugging extension command that displays information about processes, with flags controlling the amount and kind of detail. Its output connects process objects to execution and address-space context. We use the requested scope deliberately, since broad output can be substantial. The process object's presence establishes recorded operating-system state; it does not by itself establish whether the application was making useful progress.

Ref. R109, Parameters; kernel-mode restriction. See also: Process; Debugger context; !thread.

Process

Execution. An operating-system execution container that provides resources and context for one or more threads. Its address space, handles, modules, and security context help organize diagnostic evidence. We distinguish process identity from an executable name and from a reused process identifier. Two processes running the same image can have different configuration, loaded components, resource limits, and failure histories.

Ref. R015, Processes, threads, thread pools, and fibers; R037, Process-Specific Subdirectories; process memory maps and status. See also: Thread; Address space; Module.

Process Explorer

Native tools. A Sysinternals utility for examining Windows processes and related resources, including loaded components and open handles. It offers an operational view useful for selecting processes and framing further collection. We connect the displayed identities and measurements to the relevant workload and time. A visible resource or busy process is a starting observation; causation usually requires more detailed state, history, or dependency evidence.

Ref. R121, Introduction. See also: Process; Module list; VMMap.

Process Factory

Memory patterns. A parent process repeatedly creates child processes until their population becomes diagnostically significant, potentially consuming substantial resources. We inspect parent identifiers, active threads, and process lifetimes to identify the producer and distinguish ongoing creation from retained terminated objects. The pattern directs attention to the creation mechanism and workload, rather than attributing the resource problem independently to every child.

Ref. R745, Crash Dump Analysis Patterns (Part 80). See also: Zombie Processes; Error Reporting Fault; Insufficient Memory (physical memory).

Process Monitor

Native tools. A Sysinternals tool that records Windows file-system, Registry, process, and thread activity with filtering and stack-related facilities. It supports reconstruction of the operations surrounding a failure or delay. We examine results, paths, identities, and sequence together. An unsuccessful operation may be an expected probe or fallback, so the presence of an error result does not by itself identify the application's problem.

Ref. R071, Introduction; Overview of Process Monitor Capabilities. See also: ETW; Stack trace [Artifacts]; Process.

Processor

Telemetry. A pipeline component that operates on received telemetry before export, for example by batching records, filtering data, or modifying attributes. A processor can improve usefulness while also removing evidence needed for another question. Its position and configuration are part of the diagnostic context. Comparisons between raw application output and stored telemetry should account for every transformation in the path.

Ref. R257, Processors. See also: Telemetry pipeline; Attribute; Telemetry loss.

Profiling

Performance. Collecting measurements of execution and attributing observed activity or cost to code regions, functions, or call paths. A profile describes the workload that actually ran within the collection scope. It can reveal unexpected concentrations of work, but unexercised paths remain unknown. Interpretation requires the measured quantity, collection method, interval, and attribution rules, not only a ranked function list.

Ref. R390, Introduction to Profiling. See also: Sampling profiler; Instrumentation profiler; Hotspot.

Program counter

Execution. The architectural register or equivalent state identifying the current instruction position under the processor's execution rules. Debuggers use it to locate code and construct stack displays. We distinguish a faulting instruction address from an address involved in its data access. A failure reported while executing a library function can arise from arguments or state supplied by earlier application code.

Ref. R028, Section 10.14. See also: CPU register; Disassembly; Access violation.

Program slicing

Program analysis. Reduction of a program to statements relevant to a chosen criterion, usually a variable value at a program location. A backward slice follows influences on the criterion; the analysis may preserve dependencies across many possible executions. Slicing narrows the code requiring interpretation, but relevance is broader than fault: a statement may influence the result without being incorrect.

Ref. R149, Slicing criterion and relevant statements. See also: Dynamic slicing; Data-flow analysis; Fault localization.

Projective debugging

Foundations. Debugging an executable model constructed from an observed software trace. Rules translate messages into modeled entities and actions, allowing inspection of the projected execution and experiments with its behavior. Findings can guide changes to the original software, but the model's fidelity depends on what the trace records and on the translation rules.

Ref. R001, pp. 209–215. See also: Generative debugging; PLOT; Diagnostic artifact.

Projective Space

Trace patterns. A projection from a smaller trace space into a larger one to estimate where corresponding activity is located. It can narrow a search through many split files. Position-based projection is a heuristic whose usefulness depends on comparable capture intervals and message distributions; direct identifiers or timestamps can refine it.

Ref. R002, pp. 247–248. See also: Indexical Trace; Split Trace; Message Space.

Prometheus

Metrics. A monitoring system centered on dimensional time-series data, metric collection, expression evaluation, and alerting. Its model is useful for recognizing changes across instances and operations, while detailed execution context may need other artifacts. A Prometheus graph represents the result of collection and a particular query. Understanding both is necessary before interpreting a visual pattern as a property of the application.

Ref. R278, What is Prometheus?; features; architecture. See also: PromQL; Scrape; Alerting rule.

Prometheus exporter

Metrics. A component that exposes measurements from another system in a form Prometheus can scrape. It commonly adapts an existing interface or data source rather than adding application-level instrumentation. Successful collection from the exporter does not automatically establish that every underlying source was reachable or fresh. Inspect its source-specific errors, collection duration, and metric semantics when the exported view disagrees with application behavior.

Ref. R347, Exporters and integrations. See also: Exporter; Scrape; Prometheus.

PromQL Prometheus Query Language

Metrics. The expression language used to select and transform Prometheus time-series data. Its instant and range evaluations, label matching, functions, and aggregation rules determine the meaning of a result. A syntactically valid query can still answer the wrong question. Diagnostic review should examine the selected population, units, time window, handling of missing data, and whether the metric type matches the operation.

Ref. R272, Querying basics. See also: Prometheus; Rate; Recording rule.

Property-based testing

Testing. Testing executable properties over automatically generated inputs. Instead of checking only named examples, a generator explores values allowed by a declared domain and an oracle checks a general relationship. The useful coverage is constrained by both the generator and the property. A failing example makes the discrepancy concrete; successful runs do not prove the property for every permitted input.

Ref. R160, Properties and generated test data. See also: Shrinking; Test oracle; Metamorphic testing.

PSaC Problem Solving as Code

Foundations. An approach in which problem-solving knowledge, analysis configurations, and procedures are represented in forms that can be versioned and processed as code. Pattern catalogs can guide the generation or provisioning of diagnostic tools and workflows. Making the procedure executable improves repeatability only when its assumptions, inputs, and interpretation remain explicit.

Ref. R001, p. 250. See also: Pattern language; ADC; Analysis narrative.

ptrace

Native tools. A Linux interface through which a tracer can observe and control another process under the applicable permission rules. It supports operations used by debuggers and system-call tracers, including examination of registers and memory. We interpret stops according to their specific event semantics. Tracing is an intervention in target execution, and the availability or behavior of operations depends on architecture, kernel support, and security configuration.

Ref. R056, DESCRIPTION. See also: GDB; strace; Register context.

Public symbols

Artifacts. A restricted set of symbol information suitable for external distribution, typically retaining selected names and addresses while omitting much private debugging detail. The exact contents depend on how symbols are produced. We can often identify code locations and useful stack structure with this information, while recognizing that local variables or complete types may remain unavailable. Public does not mean universally complete.

Ref. R104, Private symbol data and public symbol table. See also: Private symbols; Symbol file; Symbolication.

Punctuated Activity

Trace patterns. A seemingly uniform activity divided by small local gaps into meaningful pieces. These micro-discontinuities can reveal structure before message semantics are fully understood. The gaps are interpreted within the selected activity and its time scale, since the full trace may contain unrelated work between its messages.

Ref. R002, p. 249. See also: Discontinuity; Visitor Trace; Time Scale.

Punctuated Execution

Debuggers. A design-level debugging pattern that introduces observation points into execution. In the unified debugging example, software or hardware breakpoints implement these interruptions so that relevant state can be examined. We choose the points according to the analysis question and account for the intervention when interpreting timing-sensitive behavior.

Ref. R878, Pattern discussion. See also: Breakpoint; Observer effect; Unified Debugging Pattern Language.

Punctuated Memory Leak
Memory patterns. Memory usage increases in distinct steps after particular commands, failures, or other events, with little growth between them. We compare captures taken before and after successive occurrences and identify the allocation regions responsible for each increment. The stepwise shape connects resource retention to an activity, while further analysis must determine whether the retained memory is unintended.

Ref. R533, Pattern description: Punctuated Memory Leak. See also: Memory Leak (process heap); Thread Age; Insufficient Memory (committed memory).

PVC Patterns–View–Controller
Foundations. An architectural interpretation of pattern-oriented diagnostics: catalogs supply the pattern model, domain or platform selections provide views, and tools support the analyst's interaction with artifacts. Feedback can change the catalog as new cases are investigated. The arrangement separates reusable diagnostic knowledge from a particular tool's interface and implementation.

Ref. R001, pp. 178–179. See also: Pattern-based diagnostics; Software diagnostics engineering; Pattern repertoire.

Q

Quantile
Metrics. A value locating a specified cumulative fraction of an observation distribution. A quantile depends on the population, interval, and estimation method; it is not the average of the observations near that fraction. We should distinguish a directly computed sample quantile from one estimated from histogram buckets. Quantiles from separate populations cannot generally be combined without their underlying distribution information.

Ref. R271, Quantiles. See also: Percentile; Histogram; Summary.

Query plan
Data systems. A database's chosen arrangement of operations for evaluating a statement, including scans, joins, sorting, and other work. A plan describes an execution strategy under particular statistics, parameters, settings, and schema. Estimated rows and costs are model outputs, not measured elapsed time. Diagnostic comparison should connect plan changes with actual execution evidence rather than assuming that an expensive-looking node caused the observed delay.

Ref. R298, EXPLAIN Basics. See also: EXPLAIN; EXPLAIN ANALYZE; Wait event.

Queueing delay
Performance. Time that an item spends waiting before its processing begins at a specified stage. It is distinct from the processing duration and may dominate end-to-end latency even when each handler executes quickly. Message age can expose a backlog that producer-side success rates conceal. A diagnosis should identify the queue boundary and distinguish first attempts from retries or intentional deferred delivery.

Ref. R319, Measuring availability and latency. See also: Queue length; Latency; Little's law.

Queue length
Performance. The number of work items waiting within a defined queue at an observation time or over an averaging interval. It describes accumulated demand, but not the amount of processing each item requires. A long queue may contain old work, slow consumers, or an expected burst. Interpret its length with arrival and completion rates, item age, and the queue's admission and retry policies.

Ref. R319, Queue backlogs; measuring availability and latency. See also: Queueing delay; Backpressure; Throughput.

Quiet Dump
Memory patterns. A memory dump whose inspected evidence contains no apparent deviation from the expected state: stacks resemble suitable references, and no suspicious activity or other pattern is visible. We treat this as a statement about the artifact and the analysis performed. It does not establish that the reported problem never occurred or that the capture includes its decisive interval.

Ref. R523, Pattern description: Quiet Dump. See also: False Positive Dump; Early Crash Dump; Late Crash Dump.

Quotient Stack Trace
Memory patterns. A stack trace is reduced to its module-level path by removing function detail and collapsing repeated module entries. The compact form exposes component transitions and can simplify comparison of module wait chains. We state the reduction used and retain access to the original frames. The quotient preserves selected relationships while deliberately discarding detail needed for some causal questions.

Ref. R681, Quotient Stack Trace pattern and diagnostic examples. See also: Module Stack Trace; Stack Trace Signature; Wait Chain (modules).

Quotient Trace
Trace patterns. A trace reduced by replacing message sequences sharing a selected equivalence or attribute with representative messages. Counts or other summaries can preserve information about each collapsed group. The equivalence rule determines what the reduced trace can still explain and which distinctions have been discarded.

Ref. R002, p. 250. See also: Equivalent Messages; Adjoint Message; Motivic Trace.

R

Race condition
Execution. A failure in which the result depends on an unintended ordering or interleaving of activities that share state or resources. We reconstruct the required ordering and the interval in which it can be violated. The final artifact may look like an ordinary invalid pointer or missing value. Reproducing the same outcome with controlled timing can support a mechanism without proving every observed failure shares it.

Ref. R052, Description. See also: Atomic operation; Deadlock; Use-after-free.

RAG Retrieval-augmented generation
AI and ML. A generation approach that supplies retrieved material to a language model so its response can use information outside its parameters. For diagnostic work, retrieved manuals and incident artifacts can ground terminology and candidate explanations. Retrieval quality and source applicability remain separate from fluent generation. A displayed citation is useful only when the referenced material actually supports the associated claim.

Ref. R241, Retrieval-augmented generation architecture. See also: Embedding; LLM-assisted debugging; Hallucination.

Random Data
Trace patterns. Variable message data treated as incidental variation for a particular comparison, such as addresses changed by address-space randomization. The classification is analytical: an unfamiliar value is not thereby statistically random. Stable status codes and meaningful measurements must be distinguished from incidental identity changes before normalizing messages.

Ref. R002, p. 251. See also: Message Invariant; Trace Constants; Data Association.

Random Object
Memory patterns. Execution treats unrelated or no-longer-valid memory as an object whose fields influence subsequent behavior. In the illustrated case, a wait references a freed pool region that resembles a synchronization object. We validate allocation state and object structure before interpreting its apparent state; accidental field values can transform an expected crash into a misleading wait or other milder symptom.

Ref. R504, Pattern description: Random Object. See also: Invalid Pointer (objects); Corrupt Structure; Wait Chain.

Random seed
Testing. A value used to initialize a pseudorandom generator so that its sequence can be repeated under the same relevant implementation and state. Recording seeds helps reproduce generated tests and machine-learning experiments. One seed may not control every random source. Parallel scheduling, nondeterministic operations, library changes, and device differences can still produce different behavior from apparently identical seeded runs.

Ref. R174, Controlling sources of randomness; reproducibility limits. See also: Determinism; Property-based testing; Reproducer.

Rate
Metrics. Change in a counted quantity per unit of time over a specified interval. The interval is part of the result: a long window can smooth a short burst. Prometheus rate estimates a per-second counter rate while accounting for resets and extrapolating to the window boundaries. Apply it before aggregation when separate counters can reset independently, or reset information may be obscured.

Ref. R273, rate. See also: Counter; Aggregation; Throughput.

Raw Pointer
Memory patterns. A pointer displayed without matching symbolic identification. It may fall inside a module at an unresolved offset or outside recognized modules altogether. We interpret it in the structure that contains it, especially where symbolic targets are normally expected. Missing symbols, generated code, and unusual mappings require consideration before treating the raw representation as evidence of tampering.

Ref. R843, Pattern discussion. See also: Out-of-Module Pointer; No Component Symbols; RIP Stack Trace.

RCU Read-copy-update

Execution. A synchronization approach that separates removal of an object from reclamation of its storage, allowing readers to continue under defined read-side rules. A grace period establishes when earlier readers can no longer require the removed object. We distinguish an object no longer reachable through the main structure from one already safe to free. Confusing these stages can produce lifetime failures.

Ref. R083, RCU overview; removal and reclamation. See also: Reader-writer lock; Use-after-free; Memory leak.

Reader-writer lock

Execution. A synchronization mechanism that permits compatible readers to share access while a writer requires exclusive access. Scheduling and fairness rules vary by implementation. We distinguish read ownership from write ownership when reconstructing waits. A large population of readers, a pending writer, or an attempted lock upgrade can produce different progress problems even when all stacks mention the same lock.

Ref. R081, Lesson 1, Spin locks; Lesson 2, Reader-writer spinlocks. See also: Mutex; Starvation; RCU.

Readiness probe

Cloud. A Kubernetes check indicating whether a container is ready to serve the traffic governed by readiness. A failing probe can remove a Pod from the ready endpoints of matching Services while the container continues running. It does not, by itself, request a restart. Compare readiness transitions with rollout state, dependency behavior, and actual endpoint membership when diagnosing missing or uneven traffic.

Ref. R293, Readiness probe. See also: Liveness probe; Health check; Pod event.

Recall

AI and ML. The fraction of reference-positive cases identified as positive. It measures captured positives within the specified evaluation population. It cannot count failures absent from that population or mislabeled in the reference. Raising recall may increase false positives, so report the threshold and precision alongside the result.

Ref. R234, Precision, recall, and F-measures. See also: Precision; Confusion matrix; F1 score.

Receiver

Telemetry. A collection component that accepts telemetry through a protocol or obtains it from a source. In an OpenTelemetry Collector pipeline, receivers form the input boundary and must support the intended signal. Receipt at this boundary is distinct from successful export. When an expected record is absent, receiver configuration and accepted-data counters help distinguish an ingestion problem from a later transformation or delivery problem.

Ref. R257, Receivers. See also: Collector; Telemetry pipeline; Exporter.

Recording rule

Metrics. A scheduled expression whose evaluated result is stored as another time series. Recording rules can make expensive or frequently used calculations easier to query and can standardize derived measurements. The result is already a transformation of the source data. Its rule text, evaluation interval, labels, and revision therefore belong to the diagnostic context, especially when comparing periods across a rule change.

Ref. R274, Recording rules; rule groups. See also: PromQL; Aggregation; Alerting rule.

Recovered Messages
Trace patterns. Messages reconstructed or partially identified from surviving content, context, corresponding code, or another trace when ordinary decoding fails. Recovery can restore useful diagnostic structure, but the reconstruction should record which parts are observed and which are inferred. Matching one context does not guarantee a unique original message.

Ref. R002, p. 252. See also: No Trace Metafile; Palimpsest Messages; Message Context.

RED method Rate, Errors, Duration
Performance. A service-oriented monitoring method that examines request rate, error rate, and duration. These quantities describe how much work a service receives, how often it fails, and how long operations take. They help locate a change in externally visible behavior. The definition of a request, an error, and the measured interval must remain consistent, especially when retries or asynchronous operations are involved.

Ref. R302, The RED Method. See also: Golden signals; Span metrics; USE method.

Reduced Symbolic Information
Memory patterns. Available symbols reveal only part of a module's structure, ranging from public names to exports without detailed type or local-variable information. We supplement them with function prototypes, known layouts, and meaningful arguments or strings. This supports interpretation while keeping uncertainty explicit, particularly when a displayed function name and large offset do not identify the actual internal routine.

Ref. R788, Crash Dump Analysis Patterns (Part 196). See also: No Component Symbols; Module Stack Trace; String Parameter.

Reference cycle
Runtimes. A group of objects whose references form a cycle. Reference counting alone cannot reclaim an isolated cycle while its members keep one another's counts nonzero; runtimes such as CPython supplement it with cycle collection. A cycle is not automatically a leak. Inspection settings can themselves retain objects, so the collector's debugging configuration matters when interpreting a population of apparently unreclaimed objects.

Ref. R222, Cycle collector; debugging and DEBUG_SAVEALL. See also: Garbage collection; GC root; tracemalloc.

Reference Leak
Memory patterns. Internal object references remain after the corresponding releases should have occurred, retaining objects even when their handle counts are zero. Large populations of terminated processes can reveal this pattern. We compare pointer counts, object state, and pool usage, then use reference histories where available; enabling that history collection can itself consume significant resources.

Ref. R549, Pattern description: Reference Leak. See also: Handle Leak; Instrumentation Side Effect; Object Distribution Anomaly.

Region Boundary
Memory patterns. The inaccessible surroundings that delimit a closed memory region in the structural pattern model. Unmapped pages around a stack provide an example. We establish the range and the meaning of unreadability before following nearby data. A readable region beyond the boundary may belong to another object or thread and require a different interpretation.

Ref. R830, Pattern discussion. See also: Memory Region; Memory map; Anchor Region.

Region Clusters

Memory patterns. Values from a memory region are grouped according to a chosen ordering or semantic interpretation so that repeated forms become easier to inspect. Sorting raw stack values, for example, can expose error-like values, module ranges, or regular data. We retain occurrence counts and symbolic context while checking whether each apparent cluster has diagnostic meaning.

Ref. R563, Pattern description: Region Clusters. See also: Region Spectrum; Value References; Rough Stack Trace (unmanaged space).

Region Profile

Memory patterns. A statistical description of values, addresses, and symbolic references within a selected memory region. We use distributions and correlations to guide inspection of extensive raw data, including stack residue that is difficult to review manually. The method is independent of a particular scripting library, and its results remain descriptive until connected with the region's layout and execution meaning.

Ref. R809, Crash Dump Analysis Patterns (Part 260). See also: Execution Residue; Memory Region; Structure Field Collection.

Region Spectrum

Memory patterns. Values in a memory region are classified into categories such as zero, code, heap, stack, errors, or addresses outside the expected space. Encoding or visualizing the sequence can reveal distributions and transitions that raw numbers obscure. We interpret the spectrum using the correct process mappings and value width, because the same bit pattern can admit several meanings.

Ref. R574, Pattern description: Region Spectrum. See also: Region Clusters; Rough Stack Trace (unmanaged space); Wild Pointer.

Region Strata

Memory patterns. Corresponding memory regions viewed as successive layers across snapshots. Their comparison exposes changes in contents, structure, or occupancy that one capture cannot show. We preserve region identity, mapping context, and acquisition order when aligning the layers. The archaeological analogy organizes observations over time; it does not imply that every intervening change survives in the available snapshots.

Ref. R833, Pattern discussion. See also: Snapshot Collection; Memory Region; Historical Information.

Region Summary

Memory patterns. A memory region receives an explanatory summary of its contents, symbolic references, and apparent themes. This can help orient analysis of a large raw listing, including summaries generated by a language model. We retain the underlying addresses and validate claims against the bytes and context. Coincidental symbol matches and plausible narrative should not be promoted into execution history without evidence.

Ref. R697, Region Summary pattern and diagnostic examples. See also: Rough Stack Trace (unmanaged space); Coincidental Symbolic Information; LLM-assisted debugging.

Register context

Debuggers. A selected or saved set of processor-register values used to interpret execution. A debugger may expose the currently stopped thread, a saved exception context, or values reconstructed for a caller frame. We record which context is active before reading instructions or arguments. Changing context can change the apparent stack and variables without changing any bytes in the original dump.

Ref. R087, Process Context; Register Context; Local Context. See also: Debugger context; CPU register; Exception record.

Regression bisection

Testing. Searching a sequence or graph of revisions to locate a transition between an accepted result and a failing result. Git bisect automates candidate selection from revisions classified as good, bad, or untestable. The result identifies a change associated with the chosen predicate. Build differences, unstable tests, and environmental changes can produce a misleading boundary unless controlled.

Ref. R155, Basic bisect commands; skip and bisect run. See also: Regression test; Flaky test; Reproducer.

Regression test

Testing. A repeatable test used to detect an unwanted behavioral change after software or its environment changes. A test derived from a diagnosed problem can preserve the conditions and expected outcome that matter. Its value depends on reproducing the relevant mechanism and checking the requirement. Merely observing that the formerly failing input no longer crashes may leave an incorrect result undetected.

Ref. R882, Section 2.2.3 (Confirmation Testing and Regression Testing), pp. 30–31; R165, Introduction: test case; Basic example: expected-result assertions. See also: Regression bisection; Test oracle; Reproducer.

Regular Data

Memory patterns. Memory exhibits a recognizable data regularity, such as character encoding, image data, or a domain-specific numeric layout. We use that continuity to interpret otherwise untyped bytes and to detect possible overwrites across allocation boundaries. A repeated shape supplies structural evidence, but its meaning depends on the expected format and the region in which it occurs.

Ref. R778, Crash Dump Analysis Patterns (Part 167). See also: Latent Structure; Small Value; Dynamic Memory Corruption (process heap).

Relative Density

Trace patterns. The ratio of the occurrences or densities of two semantically related message types within comparable regions. Arrival and display messages, for example, can expose an imbalance. Interpretation requires the expected relationship, matching selection windows, and treatment of missing or duplicated records.

Ref. R002, p. 253. See also: Density Distribution; Statement Density and Current; Opposition Messages.

Relative Memory Leak

Memory patterns. Growth in one allocation domain is compared with growth in another across successive captures, such as a native heap and a managed heap. Normalized sizes and successive growth ratios make differently sized populations easier to compare. We use their relationship to investigate shared retention mechanisms, while recognizing that parallel growth alone does not establish object ownership or dependence.

Ref. R557, Pattern description: Relative Memory Leak. See also: Memory Leak (process heap); Punctuated Memory Leak; Data Correlation (CPU times).

Remote debugging

Debuggers. Debugging in which the controlling debugger communicates with a target through a remote interface, often a separate target agent or stub. This permits analysis where the target cannot conveniently host the full debugger. We identify which side evaluates expressions, manages symbols, and implements stopping. Target capabilities, transport behavior, and permissions constrain the operations available; a disconnected control channel is not necessarily a target crash.

Ref. R115, Chapter 20. See also: Debugger; Breakpoint; Debugger context.

Renormalization
Trace patterns. A transformation replacing a selected message together with its context by one derived message. It supports analysis at a coarser scale while retaining a connection to the detailed neighborhood. The name borrows a physical analogy; the representation must state what is preserved and what the reduction suppresses.

Ref. R002, p. 254. See also: Message Context; Quotient Trace; Trace Frames.

Replication lag
Data systems. A measure of how far a replica's progress trails its source, expressed through time, log positions, or another explicitly defined unit. Different stages can lag: receipt, durable writing, or application of changes. PostgreSQL's write, flush, and replay lag fields describe particular acknowledgement intervals; they are not forecasts of catch-up time. Interpret lag with write activity, replication state, and the relevant stage.

Ref. R299, pg_stat_replication; lag fields and their interpretation. See also: Partial failure; Transaction isolation; Queue length.

REPLS REPL Streaming
Foundations. A diagnostic architecture that streams debugger command results into further processing and returns enriched results to the investigation. Pattern analysis, transformed output, or annotations can augment the ordinary command-and-response interaction. Here the debugger cycle is Read–Execute–Print–Loop, and the streaming extension connects it with additional analysis components and artifacts.

Ref. R001, p. 354. See also: CoTrace; ADC; Analysis narrative.

Reproducer
Testing. A package of inputs, steps, configuration, and relevant environment information that makes an observed problem occur again. It connects a report with an executable experiment and permits competing explanations to be tested against the same behavior. A reproducer should distinguish the expected result from the observed one. Missing versions, commands, or state can prevent a second investigator from observing the same problem.

Ref. R166, Information required to reproduce crashes and miscompilations. See also: Minimal reproducer; Hermetic test; Random seed.

Reproducible build
Testing. A build whose specified artifacts can be recreated byte for byte from the same source, build environment, and instructions. This allows an investigator to compare binaries without confusing unexplained build variation with a source change. The guarantee applies to the declared artifacts and inputs. It does not imply that executions of those artifacts are deterministic or that the source is correct.

Ref. R173, Reproducible-build definition. See also: Reproducer; Determinism; Regression bisection.

Reserved memory
Memory. A virtual-address range set aside for later use. In the Windows allocation model, reservation prevents competing allocations from using the range, but does not by itself commit backing storage. A large reservation therefore need not imply large physical-memory consumption. When investigating allocation failure, we distinguish a shortage of contiguous address space from a shortage of commitment or resident memory.

Ref. R091, Free, Reserved, and Committed. See also: Committed memory; Address space; Memory fragmentation.

Resident set size

Memory. A measure of the physical memory resident for a process, commonly reported as RSS on Unix-like systems. The precise accounting depends on the interface, and shared pages may be included in multiple processes. We use RSS together with mappings and private or proportional measurements. Adding process RSS values does not necessarily give the physical memory uniquely consumed by those processes.

Ref. R037, Process-Specific Subdirectories; process memory maps and status. See also: Working set [Memory]; Memory map; Memory pressure.

Resource attributes

Telemetry. Attributes describing the entity associated with telemetry, such as a service, process, container, or deployment. They place an observation within an execution environment and help separate instances that share the same operation names. Resource identity must be sufficiently precise for the investigation. A service name alone cannot distinguish two replicas, and an instance identifier alone does not explain its deployment context.

Ref. R253, Resources; resource attributes. See also: Attribute; Instrumentation scope; Log correlation.

Resource exhaustion

Execution. A condition in which demand for a resource cannot be satisfied within the available capacity or applicable limit. Leaks, excessive concurrency, unbounded requests, and unexpectedly low quotas can all produce this outcome. We identify the exhausted resource and reconstruct allocation, release, and workload rates. Exhaustion is a failure condition; it does not establish which participant or mechanism created the imbalance.

Ref. R408, Description; Common Consequences; Modes of Introduction. See also: Resource leak; Memory pressure; OOM killer; Hang.

Resource leak

Execution. A resource retained beyond the period in which the program needs it, without the release required to make it available again. Memory, file descriptors, handles, and connections are common examples. We trace acquisition, ownership transfer, and cleanup across normal and exceptional paths. Growing counts support a leak hypothesis only when legitimate workload, pooling, and delayed release cannot explain the retained population.

Ref. R407, Description; Common Consequences; Demonstrative Examples. See also: Memory leak; File descriptor; Resource exhaustion.

Resource limit

Cloud. A configured bound on a container's use of a resource, distinct from a resource request used for scheduling and reservation decisions. Kubernetes CPU limits can constrain execution through throttling, while memory-limit enforcement can lead to OOM termination. Different resources and runtimes have different enforcement behavior. A diagnostic analysis should connect the configured limit to actual usage, waiting, and termination evidence.

Ref. R325, Requests and limits; resource enforcement. See also: Saturation; OOMKilled; Eviction.

Response time

Performance. Elapsed time from a defined request or stimulus to a defined response, measured at a specified observation boundary. It can include queueing, execution, dependency calls, and transport. A server's measured handling time may therefore differ from the client's experienced delay. Decomposing response time helps locate contributing intervals, while overlapping work must not be counted repeatedly as though every duration were sequential.

Ref. R341, Time Division Method. See also: Latency; Critical path; Queueing delay.

Resume Activity

Trace patterns. Messages from an activity as it continues after a discontinuity. Their content can reveal the state in which work resumed and whether its expected continuation occurred. This differs from Break-in Activity, which concerns a possible triggering event that may belong to another activity.

Ref. R002, pp. 255–256. See also: Break-in Activity; Discontinuity; Message Change.

Retained size

Runtimes. The amount of heap storage associated with an object and the objects that would become unreachable if the relevant retaining object or set were removed, under the analyzer's reachability model. It helps identify structures responsible for keeping large object groups alive. Retained size is neither allocation rate nor a guaranteed reduction in process memory after deleting one source-level reference.

Ref. R205, Retained heap and retained sets. See also: Shallow size; Dominator tree; GC root.

Retransmission

Networks. Sending data again after the sender has not received the acknowledgement or progress expected by its protocol. TCP analysis tools recognize possible retransmissions from sequence and timing evidence, but their labels depend on what the capture contains. A retransmission can follow packet loss, acknowledgement loss, or delayed delivery. It does not alone locate the fault or prove that the original data packet was lost.

Ref. R312, TCP Retransmission; analysis flags; R307, Retransmission timeout. See also: Packet loss; Packet capture; RTT.

Retry

Distributed systems. Another attempt to perform an operation after an apparent failure or an uncertain result. A retry can recover from transient conditions while also increasing load or repeating a side effect that already occurred. Log each attempt in relation to the logical operation. A final successful response does not erase the diagnostic significance of earlier failures, additional work, and increased elapsed time.

Ref. R284, Failures Happen; Retries and backoff. See also: Idempotency; Backoff; Retry storm.

Retry storm

Distributed systems. A surge of repeated attempts that amplifies load on a struggling dependency. Retries at several layers can multiply downstream operations and sustain failure after the initial disturbance. We separate original demand from retry traffic and examine attempt limits, timing, deadline propagation, and coordination between retry policies.

Ref. R337, Retries. See also: Retry; Jitter; Cascading failure.

Reverse debugging

Debuggers. Debugging that allows movement backward through execution, using a supported recording, replay, or target mechanism. It helps locate the earlier change that produced an observed state. We distinguish the replay's available history from an ordinary memory snapshot, which does not contain execution history automatically. Support for reverse operations and the treatment of external effects depend on the debugger and its target.

Ref. R026, Chapter 6. See also: Time Travel Debugging; Memory dump; Single stepping.

Riemann RCA Pattern Language

Narratology. A proposed framework that connects diagnostic pattern languages with transformations of narrative artifacts into shorter analysis narratives and problem-solving configurations. Its optional programming-language component was conceived as a complement to the pattern language. The name refers to this diagnostic proposal and is separate from the independently developed Riemann event-monitoring system.

Ref. R001, pp. 247–249. See also: Pattern narrative; PSaC; Narralog.

RIP Stack Trace

Memory patterns. A stack trace containing raw instruction or return addresses outside the address ranges of recognized loaded modules. Such frames can direct attention to injected or otherwise unrecognized code. We first establish memory mappings and runtime context, including generated code such as .NET JIT output. A raw address on a stack is an investigative clue, not a malware verdict.

Ref. R838, Pattern discussion. See also: Raw Pointer; JIT Code (.NET); Stack Trace [Memory patterns].

Root cause analysis

Foundations. Investigation of the mechanisms and conditions that explain a problem and identify effective corrective actions. Diagnostic indicators constrain the explanation, but recognizing a symptom pattern does not complete the causal argument. We may need several interacting causes, further artifacts, and controlled experiments, and should state the boundary within which an explanation is considered sufficient.

Ref. R001, pp. 239–241; pp. 253–254. See also: APM; Defect mechanism; Causal model.

Rough Stack Trace Collection (unmanaged space)
Memory patterns. Symbolic references are collected from the raw stack regions of multiple threads, extending a rough stack scan across the thread population. The collection can reveal repeated components or past execution that current stack traces omit. We retain the source region for each reference and validate possible call relationships. Collected residue remains broader and noisier than a set of active stacks.

Ref. R693, Rough Stack Trace Collection (unmanaged space) pattern and diagnostic examples. See also: Rough Stack Trace (unmanaged space); Execution Residue; Past Stack Trace.

Rough Stack Trace (managed space)
Memory patterns. A managed stack region contains values that resolve to runtime method information, forming a rough account of possible execution residue beyond the current managed stack trace. We scan the region and interpret candidate references with runtime metadata. The output is deliberately broader than a reconstructed call chain: stale values, unrelated data, and repeated references require filtering and validation.

Ref. R671, Rough Stack Trace (managed space) pattern and diagnostic examples. See also: Managed Stack Trace; Execution Residue (managed space); Rough Stack Trace (unmanaged space).

Rough Stack Trace (unmanaged space)
Memory patterns. A collection of symbolic addresses from a thread's raw stack region, without the frame relationships required for a validated call stack. It can expose fragments of execution residue when ordinary unwinding fails. We inspect candidate calls and surrounding memory before assigning meaning, because unrelated values and stale addresses may also resolve to plausible function names.

Ref. R547, Pattern description: Rough Stack Trace (unmanaged space). See also: Execution Residue; Caller-n-Callee; Coincidental Symbolic Information.

RSDP Rapid Software Diagnostics Process
Foundations. A proposed diagnostic process with an initial tier for rapid investigation of readily recognizable cases and a subsequent tier for deeper analysis. The first tier can use familiar patterns and collection rules; unresolved cases move to more detailed work. Escalation should preserve the artifacts, observations, and reasoning already obtained.

Ref. R001, p. 183. See also: Pattern-driven diagnostics; Diagnostic report; Analysis narrative.

RTT Round-trip time
Networks. The elapsed time for a defined exchange to travel from an observation point to a remote endpoint and back. It combines outbound delay, return delay, and any included endpoint turnaround. RTT does not identify one-way delay without additional assumptions, and the two paths can differ. Compare measurements only when the exchange, endpoints, sampling method, and included processing have the same meaning.

Ref. R315, Singleton definition; methodology. See also: Latency; Packet loss; Retransmission.

Runbook

Reliability. A maintained procedure for achieving a defined operational outcome, including prerequisites, steps, expected results, and handling of exceptions. A diagnostic runbook can guide evidence collection and initial investigation without making every incident identical. Its assumptions and version matter. When observed results diverge from the procedure, record the divergence and reconsider the model instead of treating a completed checklist as proof of recovery.

Ref. R290, Implementation guidance. See also: Incident; Alert; Toil.

Runtime Thread (CLR)

Memory patterns. A thread's native stack contains CLR runtime activity, suggesting that a managed stack view may reveal the application operation behind the runtime frames. We inspect that additional view even when the native trace shows no obvious exception. Runtime participation identifies an analysis opportunity; it does not imply that the thread is a runtime service thread or that the CLR is defective.

Ref. R751, Crash Dump Analysis Patterns (Part 95). See also: Managed Stack Trace; Managed Code Exception; Version-Specific Extension.

Runtime Thread (Python, Linux)

Memory patterns. A native Linux stack exposes execution inside the Python runtime. Recognizing its interpreter and support frames identifies the execution environment even when application-level information is limited. We obtain matching runtime symbols where possible and relate the native thread to its interpreted activity; a runtime frame alone does not identify the Python operation responsible for a failure.

Ref. R506, Pattern description: Runtime Thread (Python, Linux). See also: Managed Stack Trace (Python); Reduced Symbolic Information; Runtime Thread (CLR).

Runtime verification

Program analysis. Checking observed execution events against a specified behavioral model or property while the system runs. A monitor can report a prohibited transition or a missing required event. This makes the model-to-observation relationship explicit. The conclusion is limited by what the event stream reveals, the monitor's semantics, and whether the property can be decided from the finite observation available.

Ref. R150, Runtime verification overview; monitors and reference models. See also: Model checking; Assertion; Dynamic analysis.

Ruptured Trace

Trace patterns. A trace that stops unexpectedly within an activity instead of reaching its usual closing or background sequence. It can direct investigation toward a crash, hang, or interrupted recording. Comparing the ending with expected capture behavior and obtaining contemporaneous execution artifacts helps determine what was interrupted.

Ref. R002, pp. 257–258. See also: Last Activity; Truncated Trace; Adjoint Space.

S

Same Vendor

Memory patterns. Similar failures occur in different products whose modules share a vendor or organizational origin. Version and copyright information can connect incidents that otherwise have different executable names. We use that relationship to investigate possible shared code, components, or construction practices. Common ownership suggests a comparison route but does not establish that the failures have the same defect.

Ref. R635, Same Vendor pattern and diagnostic examples. See also: Module Collection; Crash Signature Invariant; Template Module.

Sampling

Telemetry. Selecting a subset of observations for collection, retention, or analysis. Sampling reduces cost and overhead but changes what the surviving data can establish. A sample selected to preserve errors is useful for investigating failures while being unsuitable for an unadjusted estimate of their population frequency. The selection rule, probability where applicable, and decision location belong with the resulting diagnostic artifact.

Ref. R258, Sampling. See also: Head sampling; Tail sampling; Telemetry loss.

Sampling profiler

Performance. A profiler that observes execution at selected instants instead of recording every operation. Timed samples can estimate where CPU activity accumulates with relatively low overhead. Sample count is not call count, and short or rare operations may be missed. The sampling schedule, collection duration, stack quality, and selection mechanism determine how representative the resulting profile is.

Ref. R391, Sampling; Sampling versus instrumentation. See also: Profiling; CPU profile; Instrumentation profiler.

Saturation

Performance. Demand that a resource cannot immediately serve, often visible as queued work, waiting, or rejected operations. It differs from the fraction of time a resource is busy: a fully busy resource may or may not have excess demand. Saturation directs attention toward constrained capacity, but the queue's location does not necessarily identify the originating cause. Dependencies and resource limits can move the waiting elsewhere.

Ref. R301, Terminology definitions. See also: USE method; Queue length; Resource limit.

Saved Exception Context (.NET)

Memory patterns. The .NET runtime's saved exception information can preserve a usable record and machine context when the dump's default exception information is missing, invalid, or mismatched. We inspect the runtime-specific storage and verify that the recovered pair describes the event of interest. Its layout and accessibility depend on the runtime version and available symbols, so the technique requires matching support.

Ref. R814, Crash Dump Analysis Patterns (Part 276). See also: Managed Code Exception; Invalid Exception Information; Stored Exception.

Scrape

Metrics. A collection operation in which a monitoring system requests exposed metrics from a target. A scrape records the data available at that boundary and time; it does not continuously observe all state changes between requests. Failed collection must be distinguished from a successful response reporting zero activity. The target, endpoint, authentication, timeout, and parsing result are part of the collection context.

Ref. R334, scrape_config. See also: Scrape interval; Prometheus; Telemetry loss.

Scrape interval

Metrics. The configured spacing between successive metric collection attempts. It affects resolution, load, and how much short-duration behavior a sampled gauge can miss. It is distinct from a query's evaluation step and from the maximum time allowed for a scrape. A smooth chart can result from collection and query settings even when the underlying workload contains sharp bursts.

Ref. R334, global; scrape_interval; scrape_timeout. See also: Scrape; Gauge; Rate.

Screwbolt Wait Chain

Memory patterns. A client request creates server-side work that calls back into the client, which creates another thread and makes another server request, continuing the chain. We follow the alternating dependencies using stacks, thread ages, and wait times. Growing thread or handle counts may accompany the recursion; the pattern can occur beyond the original LPC and ALPC examples.

Ref. R545, Pattern description: Screwbolt Wait Chain. See also: Wait Chain; Thread Age; Handle Leak.

Script Messages

Trace patterns. Messages containing scripting content, potentially introduced through logged external input. They can indicate an injection attempt when a later viewer or processing channel interprets the content. The diagnostic question includes both how the text entered the log and whether any downstream component treated it as executable material.

Ref. R002, p. 259. See also: Malnarrative; Corrupt Message; Text Trace.

Searching Memory

Debuggers. Searching selected memory ranges for values, strings, symbols, or other representations relevant to an analysis question. We specify the range, encoding, and match criteria before interpreting hits. A matching byte sequence supplies a candidate relationship; surrounding data, type information, and execution context determine whether the match has the proposed meaning.

Ref. R001, pp. 268–270. See also: Dumping Memory; String Hint; Coincidental Symbolic Information.

Second-chance exception

Execution. A Windows debugger notification after the system has not found an application handler that resolves an exception. Microsoft also calls this the last-chance notification. If the debugger does not handle it, the process terminates. We distinguish this stage from the first notification and inspect the preserved context, while remembering that earlier corruption or failed handling can explain the final state.

Ref. R096, First-chance and last-chance notifications. See also: First-chance exception; Crash dump; Exception record.

Segmentation fault

Execution. A memory-access failure commonly reported through SIGSEGV on Unix-like systems. It identifies a failed access under the current mapping or protection, rather than a unique programming error. We inspect the fault address and instruction, then reconstruct how the address was obtained. Null references, expired pointers, invalid indexing, and exhausted stack space can converge on a similar final signal.

Ref. R055, Signal dispositions; Standard signals; R058, DESCRIPTION; MAP_SHARED; MAP_PRIVATE; MAP_ANONYMOUS. See also: Access violation; Core dump; Null pointer dereference.

Self-Diagnosis (kernel mode)

Memory patterns. The kernel deliberately stops after detecting an inconsistent condition, failed assertion, or damaged structure, before a later hardware fault would necessarily occur. The bug-check path therefore includes the detector's own activity. We identify the detection condition and examine the relevant state, separating the self-diagnostic stop from the mechanism that created the inconsistency.

Ref. R619, Self-Diagnosis (kernel mode) pattern and diagnostic examples. See also: Bug check; Assertion; Lateral Damage.

Self-Diagnosis (registry)

Memory patterns. The registry subsystem detects and reports an internal problem, and its captured diagnostic context may identify the affected hive. We correlate the bugcheck information with hive structures and paths to locate the relevant system or user-profile data. This identifies the failing configuration area without automatically deciding between corrupt contents, storage problems, and other causes of the reported registry failure.

Ref. R741, Crash Dump Analysis Patterns (Part 69c). See also: Self-Diagnosis (kernel mode); Hardware Error; System Object.

Self-Diagnosis (user mode)

Memory patterns. A runtime or component detects a problem internally and exposes its diagnosis through messages or characteristic stack activity. We inspect the retained message, title, and reporting path to determine what condition the component recognized. Such evidence can explain why execution stopped or a dump was produced, while its reported condition still requires correlation with the underlying artifact.

Ref. R495, Pattern description: Self-Diagnosis (user mode). See also: Self-Dump; Message Box; Self-Diagnosis (kernel mode).

Self-Dump

Memory patterns. A dump produced by the same process whose state it records, usually through an internal reporting or diagnostic path. The selected stack can therefore describe the mechanics of writing the dump rather than the event that motivated it. We identify the trigger and any separately preserved exception context before reconstructing the failure. Self-capture can follow an internal condition without an unhandled exception.

Ref. R385, Opening self-capture discussion and dump-writing stack example. See also: Manual Dump; Exception record; Special Stack Trace.

Self Extractor

Debuggers. A diagnostic package that carries and extracts the supporting components needed for execution, potentially choosing components appropriate to the host platform. The pattern simplifies distribution of a tool and its dependencies. We identify the extracted component and its version when interpreting results, especially where a common package can deploy different architecture-specific implementations.

Ref. R861, Pattern discussion. See also: Supporting Module; Debugware; Tool Box.

Semantic conventions

Telemetry. Shared definitions for telemetry names, attributes, units, and operation meanings. They allow data from different libraries and services to be interpreted together with fewer local assumptions. Their value is semantic consistency, not merely matching field spelling. Conventions evolve, and some remain unstable; an analysis should retain the convention or schema version when a changed name or meaning could alter the conclusion.

Ref. R255, Semantic Conventions. See also: Attribute; Instrumentation scope; Metric.

Semantic Field

Trace patterns. A set of messages associated with a subject or semantic category, such as authentication or database activity. It can be constructed through knowledge of implementation discourse or assisted classification. The category is the selected collection; a trace field is instead a mapping that assigns values to messages.

Ref. R002, pp. 260–261. See also: Implementation Discourse; Trace Field; Feature of Activity.

Semantic Mapping

Trace patterns. A transformation that replaces opaque trace identifiers or schema labels with meaningful diagnostic names. It may rename particular values, such as a thread identifier, or the fields themselves. We preserve the correspondence to original evidence so the rewritten trace remains interpretable. Unlike a Trace Field or Semantic Field, this pattern changes the presentation rather than merely assigning values or grouping meanings.

Ref. R454, Part 260. See also: Trace Schema; Trace Field; Semantic Field.

Semantic Split

Memory patterns. An apparently uniform population of anomalous observations separates into classes with different meanings or weak relationships. Blocked threads, for example, may lead to distinct display, storage, or communication dependencies. We partition the evidence using stack semantics and component roles before forming a single explanation. Similar visible symptoms do not establish that every class shares one mechanism.

Ref. R624, Semantic Split pattern and diagnostic examples. See also: Stack Trace Collection; Wait Chain; Clustering.

Semantic Structure (PID.TID)

Memory patterns. A process-identifier and thread-identifier pair forms a meaningful structure within otherwise untyped memory. In raw-stack execution residue, such a pair can help identify a communication endpoint whose message details are missing. We locate the candidate process and thread, then check their stacks and communication objects. Two adjacent integers become useful evidence only after this relationship is validated.

Ref. R642, Semantic Structure (PID.TID) pattern and diagnostic examples. See also: Execution Residue; Wait Chain (LPC/ALPC); Technology-Specific Subtrace (COM client call).

Semaphore

Execution. A synchronization mechanism that controls access through a count of available permits. It can represent a limited resource pool or coordinate producers and consumers without identifying one exclusive owner. We interpret its count and waiters according to the application protocol. Applying mutex-style ownership reasoning to a semaphore can invent an owner that the mechanism never records.

Ref. R017, Synchronization object types. See also: Mutex; Condition variable; Thread pool.

Sequence Repeat Anomaly

Trace patterns. A repeated message sequence containing unexpected additions or omissions in one or more occurrences. Comparing the anomalous block with the recurring form identifies a focused context for investigation. Before interpreting the difference, we check whether the expected repetition allows legitimate branching or changes in recording.

Ref. R002, p. 262. See also: Periodic Message Block; Missing Message; Message Pattern.

Serial Trace

Trace patterns. A trace formed by joining successive captures that use compatible schemas, such as captures across restarts. It permits analysis of the combined sequence while preserving capture boundaries. The join does not recover activity between captures, and identifiers may acquire new meanings or lifetimes across the boundary.

Ref. R002, pp. 263–264. See also: Split Trace; Trace Mask; Glued Activity.

Service map

Tracing. A graph summarizing observed relationships between services over a selected population and time interval. It is useful for locating dependencies near a failure or latency change. A graph assembled from separate one-hop calls does not establish that every displayed chain occurred in one request. Sampling, missing instrumentation, naming choices, and the graph's construction rule determine which edges are visible.

Ref. R336, Topology Graphs. See also: Distributed tracing; Span kind; Sampling.

Setting Processes

Debuggers. Selecting the process context in which subsequent inspection takes place. Address interpretation, available mappings, and other operations can depend on that selection. We state the chosen process before comparing results obtained in different contexts. Selecting a process changes the debugger's interpretation context; it does not necessarily resume or alter the process's execution.

Ref. R001, pp. 268–270. See also: Process; Listing Processes; Address space.

Setting Symbolic Information

Debuggers. Establishing the symbolic information used to interpret addresses, types, and execution state. In an elementary analysis sequence, this action prepares later operations rather than explaining the problem by itself. We check the relationship between the symbols and the captured components, then distinguish resolved information from missing or unreliable metadata.

Ref. R001, pp. 268–270. See also: Symbol file; No Component Symbols; Elementary analysis pattern.

Setting Threads

Debuggers. Selecting the thread context for subsequent register, stack, and state inspection. We identify both the process and thread because values interpreted in the wrong context can suggest a misleading execution path. The selection is an analysis operation; whether execution is also controlled depends on the debugger and the kind of target.

Ref. R001, pp. 268–270. See also: Thread; Listing Threads; CPU register.

Shadow Stack Trace

Memory patterns. A return-address view recovered from a supported hardware shadow stack provides an additional comparison for the ordinary call stack. We use it when stack-buffer damage may have altered conventional return addresses. Availability depends on target support and captured state, and discrepancies require interpretation; the shadow view is evidence about protected returns rather than a replacement for all frame data.

Ref. R818, Crash Dump Analysis Patterns (Part 288). See also: Local Buffer Overflow; Incorrect Stack Trace; Stack Trace Signature.

Shallow size

Runtimes. The storage occupied by an object itself, excluding objects reached through its references. It describes representation cost at the object's own allocation, subject to runtime layout and the heap tool's accounting. A small shallow object can retain a much larger graph. Comparing shallow and retained sizes helps distinguish a bulky object from a small owner of substantial retained state.

Ref. R205, Shallow heap. See also: Retained size; Heap histogram; Dominator tree.

SHAP SHapley Additive exPlanations

AI and ML. A framework for assigning additive feature contributions to a prediction relative to a specified baseline or expectation. It helps explain which represented inputs moved a model output and by how much under the selected formulation. Background data and dependencies between features affect interpretation. A contribution describes model behavior; it does not demonstrate the causal contribution of that feature to a software incident.

Ref. R244, SHAP framework and additive feature attribution. See also: Explainable AI; LIME; Feature vector.

Shared Buffer Overwrite

Memory patterns. Data in a shared buffer is overwritten, damaging embedded pointers or objects without overwriting allocator metadata or stack-frame control structures. Later dereferences can expose character fragments or other unrelated values as addresses. We examine the buffer's use and lifetime to distinguish overwriting from reuse of freed storage. The visible crash may follow the damaging write by a substantial interval.

Ref. R634, Shared Buffer Overwrite pattern and diagnostic examples. See also: Wild Pointer; Use-after-free; Local Buffer Overflow.

Shared Debug Event State

Debuggers. A debugging design that preserves state from one event for use when a later event is examined. The published example retains an address across breakpoint events to relate observations before and after a call. We preserve the identity and lifetime of the referenced object; carrying an address forward does not guarantee that it still denotes the same data.

Ref. R879, Pattern discussion. See also: Shared Variable; Breakpoint; Focus of Tracing.

Shared Page

Memory patterns. Different virtual addresses, possibly in different processes, map to the same underlying physical page. Comparing page-frame identities in a suitable system dump can connect otherwise separate corruption symptoms. We establish the mappings and access context before investigating a shared overwrite, since sharing is often intentional and does not itself identify which participant modified the data.

Ref. R570, Pattern description: Shared Page. See also: Shared Structure; Wild Pointer; Lateral Damage.

Shared Point

Trace patterns. An event, object, value, or activity recognizable in different traces and used to establish their correspondence. It is broader than a shared field alone and may come from the problem description. The shared point anchors correlation only within its verified identity and contextual scope.

Ref. R002, p. 265. See also: Linked Messages; Inter-Correlation; Basic Facts.

Shared Structure

Memory patterns. The same object address or handle appears in the execution context of several threads. We reconstruct and validate the relevant arguments, accounting for optimized code and reused stack slots, before connecting the observations. If one thread faults while accessing that structure, the others provide leads for ownership, lifetime, and synchronization analysis without by themselves proving a race.

Ref. R544, Pattern description: Shared Structure. See also: False Function Parameters; Wild Pointer; High Contention.

Shared Thread

Memory patterns. A thread owned by one process temporarily operates in another process's address space through a kernel attachment. Consequently, the current execution context and the thread's owning process need not agree. We examine both ownership and attachment information before interpreting memory or process lists; the discrepancy can be a legitimate part of operations such as process creation.

Ref. R568, Pattern description: Shared Thread. See also: Hidden Process; Dual Stack Trace; Coupled Processes.

Shared Variable

Debuggers. An implementation pattern for carrying selected diagnostic state between debug events. In the unified debugging example, a debugger pseudo-register stores an address for later inspection. We distinguish this debugger-side variable from application shared memory and record what event assigned its value, so later observations can be related to the correct earlier context.

Ref. R879, Pattern discussion. See also: Shared Debug Event State; Breakpoint; CPU register.

Sheaf of Activities

Trace patterns. Several selected adjoint activities examined separately and then related back to their containing traces. The construction helps compare relevant message families when background activity differs greatly. Its sheaf-inspired organization supports coordinated local views; formal sheaf properties would require an additional mathematical specification.

Ref. R002, pp. 266–267. See also: Adjoint Thread of Activity; Bifurcation Point; Message Cover.

Shrinking

Testing. The search for a simpler failing example after a generated test violates a property. Smaller values or structures can remove irrelevant detail and make the failure easier to explain. The result depends on the generator, reduction strategy, and stable preservation of the property violation. A reported minimal example should be understood within that search procedure, not as a universal minimum.

Ref. R175, Failing test cases; shrink phase and replay. See also: Property-based testing; Test-case reduction; Flaky test.

Signal

Trace patterns. A collection of related message measurements, typically sharing an invariant while carrying changing values. Selecting the corresponding activity separates the signal from unrelated background output. This pattern connects trace interpretation with measurement-oriented analysis while preserving how each value was recorded and which population it represents.

Ref. R002, pp. 268–269. See also: Counter Value; Message Invariant; Measurement.

Signal

Telemetry. A particular form of telemetry used to describe software activity, such as a metric, log record, or trace. Different signals expose different projections of the same execution: an aggregate may show an increase while an individual record supplies context. A signal becomes useful evidence only when its producer, measurement semantics, and relationship to the problem are understood.

Ref. R250, Signals. See also: Telemetry; Metric; Log record; Distributed tracing.

Signal

Execution. An operating-system notification delivered under the signal model used by Unix-like systems. Depending on its disposition, a signal can be handled, ignored, stop execution, terminate a process, or request a core dump. We retain the signal number, origin, and handling state. A signal associated with a memory fault has a different diagnostic meaning from a deliberate termination request.

Ref. R055, Signal dispositions; Standard signals. See also: Segmentation fault; Core dump; ptrace.

Significant Event

Trace patterns. An event selected for its diagnostic relevance, such as an exception, identity fact, activity boundary, or unexpected operation. Significance follows the problem and the analysis context rather than the producer's severity label alone. Trace boundaries can also be significant when assessing capture timing and completeness.

Ref. R002, p. 270. See also: Basic Facts; Error Message; Anchor Messages.

Significant Interval

Trace patterns. The recording interval needed to preserve the events and state observations required for an investigation. Capturing the final symptom may be insufficient if the preceding state or transition lies outside the artifact. The pattern connects the diagnostic question with capture timing, retention, and state-logging frequency.

Ref. R002, p. 271. See also: State Dump; Data Interval; Truncated Trace.

Silence

Reliability. A time-bounded rule that suppresses notifications for alerts matching specified criteria. In Alertmanager, silences use label matchers and an active time range. They are useful when a condition is already understood or work is planned. A silence does not mean the underlying condition is healthy; its scope, reason, owner, and expiry are essential context when investigating why no notification reached responders.

Ref. R276, Silences. See also: Alert; Alert inhibition; Incident.

Silent Messages

Trace patterns. Analytical placeholders for unoccupied positions at a chosen time resolution between recorded messages. They allow absence and spacing to be studied using trace patterns. A silent message is not an emitted record or proof of inactivity; its number and distribution depend on the selected resolution and interval.

Ref. R002, pp. 272–273. See also: Time Scale; Discontinuity; Statement Density and Current.

Single stepping

Debuggers. Controlled advancement of execution by a source-level step or a machine instruction. Step-into and step-over operations apply different treatment to calls, and source steps depend on available line information. We compare the observed instruction sequence with the intended source operation. Optimized code can make source-level movement appear irregular, while stopping between operations can change interleavings and hide or expose a concurrency failure.

Ref. R022, Section 5.2. See also: Disassembly; Breakpoint; Source mapping.

Singleton Event

Trace patterns. An event expected to occur once within the relevant lifetime or scenario, or to have occurred before recording began. An additional occurrence can reveal restart, repeated initialization, or a violated expectation. The lifetime and configuration must be specified before counting an event as an unexpected duplicate.

Ref. R002, p. 274. See also: Glued Activity; Original Message; Identification Messages.

Singleton Trace

Trace patterns. A trace containing a single message, such as a status display, error dialog, or abort report. Although its chronology is minimal, its content and associated context can still be analyzed. The message may include a stack trace or references to other artifacts that extend the available evidence.

Ref. R002, p. 275. See also: Trace Extension; Fiber Bundle; Error Message.

SLA Service level agreement

Reliability. An agreement that defines service commitments and consequences associated with meeting or missing them. Its indicators, exclusions, evaluation intervals, and remedies may differ from internal operational objectives. Diagnostic evidence used to evaluate an agreement must follow its stated measurement convention. A technically observed failure and a contractual breach are therefore separate conclusions, even when they concern the same service interruption.

Ref. R280, Agreements. See also: SLI; SLO; Availability.

SLI Service level indicator

Reliability. A quantitative measure of a service characteristic relevant to users, such as requests completed successfully within a latency threshold. Define its population, measurement point, interval, and treatment of missing data. An internal counter may only approximate the intended outcome. The distinction between the desired indicator and its implementation belongs in its definition.

Ref. R280, Indicators; collecting indicators. See also: SLO; SLA; Metric.

SLO Service level objective

Reliability. A target or acceptable range for a service level indicator over a defined evaluation interval. It makes a reliability expectation explicit and provides a basis for detecting unacceptable service behavior. The target is meaningful only with the indicator's population and measurement rules. Meeting an aggregate objective can still conceal a poorly served region, operation, or user group that was combined with healthier populations.

Ref. R280, Objectives. See also: SLI; Error budget; SLA.

Slow query log

Data systems. A selected record of database statements meeting configured duration and other criteria. It narrows investigation to a population defined by the logging policy, not every operation that contributed to an application delay. In MySQL 8.4, initial lock-acquisition time is excluded from the stated execution-time measure, and entries are written after completion and lock release. Log order can therefore differ from execution order.

Ref. R329, The Slow Query Log. See also: Query plan; Database blocking; Response time.

Small DA+TA Dump Artifact + Trace Artifact

Trace patterns. Small supporting artifacts, such as configuration files or registry details, analyzed alongside larger dump and trace artifacts. One setting can explain behavior that extensive execution logging leaves ambiguous. The pattern directs attention to the relevant configuration and its effective time, not merely to the largest available file.

Ref. R002, pp. 276–277. See also: Cartesian Trace; Hidden Facts; Master Trace.

Small memory dump

Artifacts. A Windows system-crash artifact preserving a limited set of stop information, execution context, stack data, and module information, with additional captured data depending on the platform. It can support initial classification when larger dumps are unavailable. We avoid treating its reduced memory coverage as a full view of the system, particularly when the suspected corruption occurred outside the preserved thread state.

Ref. R117, Included state and analysis limitations. See also: Bug check; Kernel memory dump; Minidump.

Small Value

Memory patterns. A small numeric value in registers or raw memory may represent a character, identifier, handle, count, or part of a larger structure. We interpret it through neighboring values and the operation that uses it, rather than treating every word as an address. Collections of such values can reveal meaningful aggregates or the objects involved in a wait.

Ref. R790, Crash Dump Analysis Patterns (Part 202). See also: Regular Data; Latent Structure; Wait Chain.

Snapshot Collection

Memory patterns. A collection of separate snapshot artifacts, commonly acquired at different times or under different conditions. Comparing them can reveal growth, disappearance, or a change between working and failing states. We preserve each artifact's identity and acquisition context. This collection differs from an Aggregate Snapshot, which combines constituent observations within one artifact.

Ref. R828, Pattern discussion. See also: Memory Snapshot; Aggregate Snapshot; Region Strata.

Soft lockup

Execution. A Linux watchdog condition indicating that a CPU has remained in kernel execution without the expected watchdog-thread progress for the configured interval. We examine the reported stack and workload to investigate the scheduling stall. The observation alone does not distinguish an unintended loop from an exceptionally long operation, and detector configuration affects what is reported.

Ref. R084, Configuration; Implementation. See also: Hard lockup; Kernel panic; Stack trace [Artifacts].

Software antinarrative

Narratology. A narrative containing an event sequence that is considered impossible under the current model of software behavior. The contradiction prompts investigation of the model, instrumentation, data integrity, and interpretation. It is a useful diagnostic discrepancy rather than immediate proof that the executing system violated its own rules.

Ref. R001, p. 141. See also: Impossible Trace; Causal model; Malnarrative.

Software codiagnostics

Foundations. Artifact transformations that cooperate with diagnostic analysis by exposing additional indicators or making patterns easier to recognize. Filtering, rearranging, projecting, and otherwise transforming a log can produce complementary views of the same evidence. We preserve the original artifact and document transformation rules so that newly visible patterns can be checked against their source.

Ref. R001, pp. 275–276. See also: Software diagnostics; General trace; Diagnostic operad.

Software diagnostics

Foundations. The study and interpretation of signs of software structure and behavior in diagnostic artifacts. It uses analysis methods to identify normal organization, deviations, and problem patterns, then connects the findings with further investigation. Its scope includes memory, traces, logs, and related data across system boundaries and throughout software construction and operation.

Ref. R001, p. 161; pp. 244–246; p. 285. See also: Pattern-oriented diagnostics; Software pathology; Debugging.

Software diagnostics engineering

Foundations. The engineering of diagnostic capabilities together with the methods needed to use them. It connects instrumentation, collection, tools, and architectures with post-construction artifact analysis and problem solving. The discipline spans devices, applications, servers, and distributed systems, so diagnostic requirements should address how evidence will support investigations across those boundaries.

Ref. R001, pp. 285–286. See also: Debugware; Diagnostics-driven development; Systemic diagnostics.

Software diagnostic space

Foundations. A model of the artifacts, states, and relationships available to an investigation. Problem descriptions, reproductions, traces, and snapshots can be connected into a graph through which analysis searches for relevant patterns. The space may be incomplete or disconnected; an absent relationship can identify a collection need rather than a gap that should be filled by assumption.

Ref. R001, pp. 258–262. See also: Diagnostic artifact; APM; Software codiagnostics.

Software Exception

Memory patterns. Software explicitly raises an exception to report a detected condition or transfer control through exception handling. A raising function and its callers can reveal this path in a stack or execution residue. We examine the exception record and the reason for raising it. The mechanism alone does not establish whether the condition was expected, handled, or fatal.

Ref. R674, Software Exception pattern and diagnostic examples. See also: C++ Exception; Translated Exception; Exception record.

Software forensics

Foundations. Investigation of retained software artifacts to reconstruct relevant structure, behavior, and earlier activity. Memory snapshots and traces contain surviving evidence whose meaning depends on context, capture, and subsequent changes. The reconstruction is bounded by what remains observable; overwritten data and incomplete acquisition can prevent a unique account of the past.

Ref. R001, pp. 54–56; p. 263. See also: Memory dump analysis; Diagnostic artifact; Execution Residue.

Software narrative

Narratology. A representation of software events, states, and their relationships that can be read as an account of computation. Logs and traces are common examples, but problem descriptions and construction narratives also qualify. The narrative has a viewpoint and selection of material; its recorded sequence is not automatically the entire history of the system.

Ref. R001, pp. 144–149. See also: Software narratology; Extended software trace; Pattern narrative.

Software narratology

Narratology. The application of narrative concepts to representations of computation and software work. It distinguishes events, their execution history, the recorded narrative, and its presentation, helping us examine how an artifact selects and organizes information. These distinctions support trace analysis while preserving the difference between software behavior and the story available to an analyst.

Ref. R001, p. 58; pp. 121–122; pp. 145–149. See also: Software narrative; Software plot; Trace Viewpoints.

Software pathology

Foundations. The study of mechanisms that produce abnormal software structure and behavior. It connects diagnostic indicators with their causes and effects through traces, logs, memory snapshots, and other narratives. The pathology analogy directs attention to mechanisms; it does not make a symptom pattern a unique diagnosis or replace investigation of the underlying implementation.

Ref. R001, pp. 305–306. See also: Defect mechanism; Software diagnostics; DMP.

Software plot

Narratology. A selected sequence of recorded software events used to present a relevant story of computation. Filtering a larger trace can create a plot centered on an activity or problem. The selection improves readability but also changes what the reader can infer, so the relationship to the original artifact and omitted material should remain recoverable.

Ref. R001, pp. 121–122; pp. 349–350. See also: Software narrative; Minimal Trace; Trace Viewpoints.

Software prognostics

Foundations. Investigation aimed at anticipating problematic behavior from currently observable patterns, conditions, and changes. Preventive memory analysis and monitoring can identify indicators that warrant action before a visible failure. The prediction should state its evidence and limits; recognizing a concerning pattern does not by itself determine whether or when a future failure will occur.

Ref. R001, pp. 171–172; pp. 273–274. See also: Premortem diagnostics; Zero Fault diagnostics; Diagnostic hypothesis.

Sorted Trace

Trace patterns. A trace rearranged according to selected attribute values, such as activity identifiers, message types, or data fields. Sorting exposes group structure and can precede further reduction. The resulting order is an analytical presentation; original sequence and timestamps should remain available for questions that depend on execution order.

Ref. R002, p. 278. See also: Adjoint Thread of Activity; Quotient Trace; Event Sequence Order.

SOS

Runtimes. A debugging extension for inspecting managed runtime structures, including threads, stacks, exceptions, objects, and garbage-collection roots. It interprets runtime data in a live target or suitable dump through a supported host. The output depends on artifact completeness and matching runtime support. Object and stack reports supply evidence for analysis; their presence does not establish that the listed structure caused the reported problem.

Ref. R199, SOS debugging extension; command reference. See also: dotnet-dump; ClrMD; GC root.

Source mapping

Debuggers. The association between instruction addresses and source files, lines, or constructs. Debug information supplies this association, which the debugger combines with loaded image addresses. We verify source and symbol agreement with the captured build. Optimization can map a statement to several instruction ranges or combine statements, limiting what a displayed source line explains.

Ref. R029, Section 9.6; R030, Section 10.3. See also: Disassembly; Build ID; Optimized-out variable.

Source Stack Trace

Memory patterns. A stack trace carries source-level information for its frames, such as file names and line references obtained from symbols. This adds another representation in which recurring subtraces and relationships can be examined. We check that symbols and source correspond to the captured build. Source annotations improve navigation but do not remove uncertainty caused by optimization or incomplete unwinding.

Ref. R687, Source Stack Trace pattern and diagnostic examples. See also: Stack trace [Artifacts]; Symbol file; Source mapping.

Span

Tracing. A recorded operation with a start, an end, identity, and associated context. Spans can describe requests, internal work, or asynchronous processing and can be related through parentage or links. Their boundaries follow instrumentation choices. A long span establishes an elapsed interval for that operation; it does not establish how much of the interval was CPU execution, queueing, network delay, or waiting.

Ref. R266, Spans. See also: Span context; Span link; Latency.

Span context

Tracing. Identification and propagation information associated with a span: its trace identifier, span identifier, trace flags, and trace state. Another operation can join or reference the execution through this context. A valid context establishes association under the tracing model; it neither authenticates the caller nor explains internal behavior.

Ref. R259, SpanContext. See also: Context propagation; Trace identifier; Span identifier.

Span Differentiator

Debuggers. A diagnostic utility whose selected options determine the functional role it serves, such as client, server, or middleware investigation. Several roles may coexist on one physical machine. Here span concerns the utility's deployment and functional scope. We retain the chosen role when interpreting output; it is separate from a distributed-tracing span.

Ref. R860, Pattern discussion. See also: Debugware; Configuration Wrapper; Span.

Span event

Tracing. A timestamped annotation attached to a span to record something that occurred during the described operation. An event can carry attributes and mark a decision, exception, or other relevant occurrence without creating another duration-bearing span. Its timestamp and context are evidence of the recorded occurrence. Events omitted by instrumentation or retention cannot be inferred merely from a quiet section of the span.

Ref. R266, Span Events. See also: Span; Attribute; Log record.

Span identifier

Tracing. An identifier for one span within a trace. Together with the trace identifier, it distinguishes an operation and supports parent or link relationships. It is neither a process nor a thread identifier. Several spans may share a thread; logical operations can cross execution contexts.

Ref. R259, SpanContext: SpanId. See also: Trace identifier; Span context; Span link.

Span kind

Tracing. An OpenTelemetry classification of a span's role: Internal, Server, Client, Producer, or Consumer. It describes the operation's relationship to incoming, outgoing, or deferred work. Kind helps reconstruct boundaries and dependencies rather than implementation details. Incorrect classification can distort service graphs and metrics derived from spans.

Ref. R259, SpanKind. See also: Span; Service map; Span metrics.

Span link

Tracing. An explicit association from a span to another span's context, including one in a different trace. Links can represent relationships that do not fit a single parent, such as batch consumption or deferred work. They help reconstruct a wider execution story. Their presence records an instrumentation-declared relationship; their absence does not prove that two operations were independent.

Ref. R266, Span Links. See also: Span; Causal order; Context propagation.

Span metrics

Tracing. Metrics derived by aggregating span data, commonly request rates, error rates, and duration distributions for services or operations. They connect a broad performance view with individual traces that can be inspected. Their population is the set of spans available to the aggregation stage. Sampling, dropped spans, duplicated exports, and semantic changes can therefore alter the metrics independently of the application workload.

Ref. R342, Service Performance Monitoring. See also: RED method; Span status; Sampling.

Span status

Tracing. An OpenTelemetry outcome classification: Unset, Ok, or Error. The default Unset does not supply the positive assertion represented by Ok. Status reflects instrumentation rules rather than a universal business definition of success. Interpretation therefore requires the operation's semantic conventions and its recorded error context.

Ref. R259, Set Status. See also: Span; Semantic conventions; HTTP status code.

Sparse Trace

Trace patterns. Insufficient logging coverage in the code paths relevant to the investigation. Expected execution details are absent because no suitable statements record them, even though collection may function correctly. The pattern distinguishes a need for additional instrumentation from a component omitted by configuration or an intrinsic visibility limit.

Ref. R002, p. 279. See also: Declarative Trace; Missing Component; Visibility Limit.

Spatial Form

Trace patterns. A Sorted Trace arranged primarily by spatial, topological, or diagnostic distance from a chosen origin. Messages in each distance layer may retain their local time order. We declare the origin and distance rule to distinguish nearby symptoms from effects in more remote components. Borrowed from literary analysis, the pattern supplies a relational reading strategy without discarding temporal evidence.

Ref. R453, Part 258. See also: Sorted Trace; Layered Periodization; Trace Network.

Special Process

Memory patterns. A process whose presence or role provides an unusual clue about the system's current circumstances. Debuggers, error-reporting processes, and environment-specific helpers can indicate an exceptional workflow or an additional participant in the problem. We inspect identity, command line, relationships, and timing rather than relying on the executable name alone. The process may explain how evidence was collected without explaining why the original failure occurred.

Ref. R382, Opening process-role discussion and debugger/reporting-process examples. See also: Process; Coupled Processes; Manual Dump.

Special Stack Trace

Memory patterns. A stack trace representing an unusual or diagnostically significant state within the application's expected lifecycle. Termination, module loading, and exception-related paths can reveal that collection occurred during a transition rather than ordinary processing. We connect the trace with thread responsibilities and the collection method. The unusual state may be a consequence of the problem or of capture itself, rather than its initiating mechanism.

Ref. R374, Opening uncommon-state discussion and lifecycle stack examples. See also: Stack trace [Artifacts]; Main Thread; Manual Dump.

Special Thread (.NET CLR)

Memory patterns. A CLR thread has a recognizable runtime role, such as finalization, debugging support, or thread-pool maintenance. Recognizing that role helps distinguish expected infrastructure activity from application behavior. We interpret its stack and state against the runtime version and surrounding dependencies. A familiar role explains why the thread exists; its particular wait or execution path may still require investigation.

Ref. R651, Special Thread (.NET CLR) pattern and diagnostic examples. See also: Runtime Thread (CLR); Finalization; Thread pool.

Special trace

Narratology. A trace interpreted with explicit or implicit temporal structure, including the ordering of execution events and their relationships. Conventional timestamped logs are common examples. Temporal information enables useful analysis, but clock order, recording order, and causal order can differ, so the chosen interpretation must respect the trace’s collection and synchronization properties.

Ref. R001, pp. 206–208. See also: General trace; Generalized software trace; Event Sequence Order.

Spectrum-based fault localization

Testing. Ranking program elements from execution summaries, commonly whether each statement was covered by passing and failing tests. A suspiciousness formula gives higher priority to elements associated with failure under its assumptions. The ranking is sensitive to test selection, correlated coverage, and multiple defects. It supplies candidates for examination and experiment; a high score is not a causal demonstration.

Ref. R156, Passing/failing coverage and suspiciousness visualization. See also: Fault localization; Code coverage; Statistical debugging.

Spike Interval

Memory patterns. A time interval during which a thread's CPU consumption rises, inferred by comparing cumulative CPU times across successive dumps. We correlate those increments with the stacks captured during each interval to localize active work. The sampling schedule limits temporal precision, and expected bursts such as application startup should be considered before interpreting every increase as abnormal.

Ref. R781, Crash Dump Analysis Patterns (Part 176). See also: Spiking Thread; Stack Trace Surface; Waiting Thread Time (user dumps).

Spiking Interrupts

Memory patterns. Interrupt and DPC activity consumes enough processing time to contribute to response delays or apparent freezes. We inspect per-processor counts and time, then compare an appropriate baseline or interval. Large cumulative totals alone are insufficient because they depend on system age and workload. In the source's Windows 11 example, kernel DPC delegate threads can appear as RUNNING even when swapped out; we check their context-switch counts and kernel time before interpreting that display as sustained DPC activity.

Ref. R692, Spiking Interrupts pattern and diagnostic examples. See also: DPC; CPU utilization; Distributed Spike; Spiking Interrupts (Linux).

Spiking Interrupts (Linux)

Memory patterns. Disproportionate hard-IRQ or softirq activity coincides with response lag, apparent freezes, or kernel CPU pressure. We correlate per-CPU counter changes, interrupt affinity, relevant kernel stacks, and watchdog or RCU-stall messages. Cumulative counts require an uptime-aware comparison, and a busy workqueue thread is supporting evidence only after its work is connected to interrupt processing.

Ref. R578, Linux mapping, diagnostic guidance, and summary. See also: Spiking Interrupts; Interrupt Stack Collection; Paratext.

Spiking Thread

Memory patterns. A thread associated with unexpectedly high processor consumption during the problem interval. Its stack and instructions suggest where execution is spending time, while thread identifiers connect the dump to external measurements. We distinguish accumulated lifetime CPU time from current activity and compare repeated observations when possible. A running thread in one snapshot need not be the thread responsible for sustained processor demand.

Ref. R355, CPU-consuming thread discussion and thread-time examples. See also: Thread; Busy System; Dispatch Level Spin.

SPIN

Program analysis. A model checker for concurrent systems described in Promela. It explores interactions and checks properties such as assertions and temporal requirements, producing trails for violating behavior. A trail explains a problem in the modeled system. Its relevance to an implementation depends on the mapping between modeled states and real operations, including details intentionally omitted to keep exploration tractable.

Ref. R147, Basic Spin Manual; models and verification. See also: Model checking; Runtime verification; Bounded model checking.

Spinlock

Execution. A lock whose acquisition path can wait by repeatedly checking until ownership becomes available. It trades sleeping and rescheduling for active waiting under suitable conditions. We examine both the spinning participants and the owner, together with interrupt and preemption context. A hot acquisition loop can reveal contention, but it does not explain why the critical section remains occupied.

Ref. R081, Lesson 1, Spin locks; Lesson 2, Reader-writer spinlocks. See also: Critical section; Deadlock; IRQL.

Split Message

Trace patterns. One logical message emitted in several pieces, potentially separated by records from other activities. Recognizing the producer's delimiters and activity identity allows the pieces to be reassembled. This differs from intentionally separate messages for complementary operations, such as a call record and a return record.

Ref. R002, pp. 280–281. See also: Ornament; Collapsed Message; Opposition Messages.

Split Trace

Trace patterns. One recording distributed across an ordered collection of files or fragments. Analysis must preserve sequence, boundaries, and any missing parts. File ordering, capture metadata, and time ranges help determine which fragments cover the reported occurrence and whether joining them retains a continuous history.

Ref. R002, p. 282. See also: Serial Trace; Blackout; Indexical Trace.

SRE Site reliability engineering

Reliability. An approach that applies software engineering to operating services and meeting reliability objectives. It includes measurement, automation, capacity and change management, and incident response. For diagnostics, the important connection is between user-visible outcomes and the engineering work needed to explain or improve them. Reliability targets and error budgets provide context for prioritizing that work rather than treating every deviation as equally important.

Ref. R338, Google's Approach to Service Management: Site Reliability Engineering; Tenets of SRE. See also: SLO; Error budget; Toil.

Stack

Memory. Storage associated with active execution, commonly used for call frames, saved state, and local data. Its organization depends on the architecture, compiler, and runtime. The stack is also a source of residual values left by earlier activity. We distinguish the currently active call structure from such residual bytes, because a recognizable return address outside an active frame does not prove a current call.

Ref. R027, Section 8.1. See also: Stack frame; Stack trace [Artifacts]; Stack unwinding.

Stack frame

Memory. The execution state associated with a particular function invocation, as represented by a debugger or the calling convention. It can involve stack storage, registers, and unwind information rather than one simple block. We identify the selected frame before interpreting arguments and local variables. Optimizations, inlining, and damaged state can make the displayed frame differ from a literal source-level call.

Ref. R027, Section 8.1. See also: Stack; Calling convention; Optimized-out variable.

Stack overflow

Memory. Exhaustion of the storage available for a thread stack, often associated with excessive recursion, large local allocations, or insufficient stack capacity. It differs from overwriting a particular stack buffer. We examine repeated frames, allocation size, and stack limits to reconstruct the mechanism. The final exception may occur in ordinary code that merely requested the next increment of stack space.

Ref. R132, Reserved and committed stack memory; stack-growth limits. See also: Stack; Buffer overflow; Guard page.

Stack Overflow (insufficient memory)

Memory patterns. A thread cannot grow its committed stack because the necessary additional memory is unavailable. The resulting stack-overflow exception can appear with a short call chain and substantial reserved address space. We inspect stack-region commitment, the guard-page situation, and other low-memory evidence before interpreting the failure as excessive recursion or unusually deep execution.

Ref. R468, Pattern description: Stack Overflow (insufficient memory). See also: Insufficient Memory (committed memory); Stack Overflow (user mode); Hidden Exception.

Stack Overflow (kernel mode)
Memory patterns. A kernel thread exhausts its available stack, possibly while already handling another exception. The original x86 example reaches a double fault and requires recovery of the earlier processor context. We inspect stack boundaries, frame consumption, and raw stack contents; a short or damaged reconstructed trace does not exclude deeper call activity that consumed the stack.

Ref. R467, Pattern description: Stack Overflow (kernel mode). See also: Stack Overflow (user mode); Exception Stack Trace [Memory patterns]; Rough Stack Trace (unmanaged space).

Stack Overflow (managed space)
Memory patterns. Stack exhaustion caused by managed execution, visible through repeated JIT-generated calls and the underlying user-mode stack-overflow exception. Managed stack display can become slow or incomplete when there are many small frames. We compare native and managed views and map relevant instruction addresses back to managed methods before identifying the recursion's origin.

Ref. R592, Stack Overflow (managed space) pattern and diagnostic examples. See also: Stack Overflow (user mode); JIT Code (.NET); Managed Stack Trace.

Stack Overflow (software implementation)
Memory patterns. A software-maintained stack structure exhausts or exceeds its allocated entries. The published example concerns the stack-location array in a Windows I/O request packet, where the request requires more locations than were provided. We inspect the structure's capacity and current position. This pattern applies to the logical data structure and should be distinguished from exhaustion of a thread's call stack.

Ref. R713, Crash Dump Analysis Patterns (Part 16c). See also: IRP; Stack overflow; Double IRP Completion.

Stack Overflow (user mode)
Memory patterns. Exhaustion of a user-mode thread's stack, often accompanied by a characteristic exception and repeated call frames. A very long trace may obscure the initial caller or exceed ordinary display limits. We inspect the repetition and the raw stack near its beginning to reconstruct the originating call, distinguishing the repeated failure mechanism from the code that initiated it.

Ref. R591, Stack Overflow (user mode) pattern and diagnostic examples. See also: Stack overflow; Near Exception; Stack trace [Artifacts].

Stack pointer
Memory. A processor register that identifies the current stack position according to the architecture and calling convention. Its value helps delimit active stack storage and supports stack unwinding. We compare it with the thread stack limits and instruction context. An unusual value can indicate corruption or stack switching, but its numerical appearance alone is insufficient to distinguish those explanations.

Ref. R028, Section 10.14. See also: Stack frame; CPU register; Stack overflow.

Stack Trace

Memory patterns. A sequence of frames serves as a diagnostic signature and a route into the failing activity. We examine the relevant exception context, symbol quality, and progressively broader subtraces when comparing with known cases. A common framework frame has less identifying value than a specific call sequence, and a matching signature remains a lead requiring contextual verification.

Ref. R473, Pattern description: Stack Trace. See also: Crash Signature; Exception Stack Trace [Memory patterns]; Incorrect Stack Trace.

Stack trace

Artifacts. An ordered presentation of reconstructed call frames for a thread or execution context. It commonly includes function names, addresses, and sometimes source locations or arguments. We interpret it as a model built from registers, memory, and unwind information. Missing symbols, optimized calls, damaged stacks, or absent dump pages can limit this model; adjacent displayed frames should not automatically be treated as a complete historical narrative.

Ref. R021, Section 8.2. See also: Stack frame; Stack unwinding; Symbol file.

Stack Trace Change

Memory patterns. A difference between corresponding stack traces in captures taken at different stages of an incident. We establish thread identity and capture order, then compare the call sequences with the expected state and reported behavior. The changed region can locate where execution moved when a hang or CPU spike appeared, while unchanged frames supply context for that transition.

Ref. R535, Pattern description: Stack Trace Change. See also: Stack Trace [Memory patterns]; Variable Subtrace; Early Crash Dump.

Stack Trace Collection

Memory patterns. A set of stack traces examined together to reconstruct relationships and recurring states across threads or processes. Shared waiting paths, exceptional traces, and repeated work become visible at the collection level. We preserve thread identity, capture scope, and acquisition time when grouping similar stacks. Identical-looking traces can belong to different roles, and a collection from one moment does not establish persistence.

Ref. R361, Opening collection discussion and multi-thread stack examples. See also: Stack trace [Artifacts]; Coupled Processes; Blocked Thread.

Stack Trace Collection (CPUs)

Memory patterns. Stack traces from the threads currently executing on the captured CPUs, including idle threads where available. This collection can reveal that an automatic analysis displayed the selected processor instead of the processor that raised the bug check. We compare processor identities, running threads, and their stacks before interpreting the reported fault location or investigating CPU-intensive activity.

Ref. R600, Stack Trace Collection (CPUs) pattern and diagnostic examples. See also: Stack Trace Collection; Bug check; Spiking Thread.

Stack Trace Collection (I/O requests)
Memory patterns. A collection of I/O-request stack locations exposes how requests pass through device-driver layers. These records describe request handling and parameters rather than ordinary thread call frames. We inspect active and completed IRPs together, preserving each request's current location, completion state, and associated devices, to find repeated routes or a common point of stalled progress.

Ref. R476, Pattern description: Stack Trace Collection (I/O requests). See also: Stack Trace (I/O request); Object Distribution Anomaly; Wait Chain.

Stack Trace Collection (managed space)
Memory patterns. Managed stack traces collected across the threads of a captured process. The collection reveals repeated managed execution paths and relationships that a single selected thread can conceal. We use a runtime-compatible inspection extension and retain thread identities when relating managed frames to native stacks. An unavailable or incompatible extension limits the interpretation rather than proving that managed execution was absent.

Ref. R599, Stack Trace Collection (managed space) pattern and diagnostic examples. See also: Stack Trace Collection; Managed Stack Trace; SOS.

Stack Trace Collection (predicate)
Memory patterns. A stack-trace collection selected by an explicit condition, such as CPU consumption, suspension state, or a particular wait object. We use the predicate to isolate a relevant population before comparing its frames and dependencies. Recording the condition and capture scope preserves the meaning of the result: threads excluded by selection should not be mistaken for missing threads.

Ref. R721, Crash Dump Analysis Patterns (Part 27c). See also: Stack Trace Collection; Module Collection (predicate); Thread Cluster.

Stack Trace (database)
Memory patterns. Recorded call stacks are retained in a dedicated memory database for actions such as allocations or handle creation. We interpret its return-address records using the database's format, which differs from an ordinary debugger frame listing. These historical stacks connect present resources to earlier activity when the corresponding recording facility was enabled and its data survives.

Ref. R474, Pattern description: Stack Trace (database). See also: Historical Information; Memory Leak (process heap); Instrumentation Information.

Stack Trace (file system filters)
Memory patterns. A file-system filter operation has a callback sequence that may not be fully visible in the ordinary thread stack or IRP stack locations. We inspect filter-manager context to recover the participating filters and pending callbacks. This adds another view of the I/O path, especially when asynchronous processing separates the current thread from the operation that remains outstanding.

Ref. R719, Crash Dump Analysis Patterns (Part 25c). See also: IRP; Wait Chain; Hidden IRP.

Stack Trace (I/O devices)

Memory patterns. A device hierarchy examined in relation to an I/O request's current stack location. The device-stack view runs through the participating devices, while the request-stack representation exposes the request's progress through its locations. We relate both views before assigning responsibility to a driver. A visible device in the hierarchy establishes participation, not the cause of a stalled request.

Ref. R598, Stack Trace (I/O devices) pattern and diagnostic examples. See also: IRP; Stack Trace (I/O request); Wait Chain.

Stack Trace (I/O request)

Memory patterns. The sequence of I/O stack locations associated with an I/O request packet. These locations expose driver and device participation and the request's current position, rather than an ordinary thread call stack. We inspect the current and retained locations together with device and file information. Those relationships can redirect an investigation toward the request or a blocking file.

Ref. R597, Stack Trace (I/O request) pattern and diagnostic examples. See also: IRP; Stack Trace (I/O devices); Blocking File.

Stack Trace Motif

Memory patterns. A recognizable sequence of stack frames conveys an operation's purpose, such as printing, querying a database, or processing a document. We compare these functional motifs across threads and processes to identify related activity and dependencies. One trace may contain several motifs; similarity in purpose guides correlation even when exact frame addresses or implementation details differ.

Ref. R560, Pattern description: Stack Trace Motif. See also: Stack Trace [Memory patterns]; Coupled Processes (semantics); Technology-Specific Subtrace (JIT .NET code).

Stack Trace Race

Memory patterns. Related threads or processors are captured in similar operations, with one showing an exception or other failure, suggesting a possible race over shared state. We compare their common subtraces, objects, and execution context to formulate that hypothesis. Simultaneous stack similarity supports investigation of concurrency; the actual conflicting accesses and missing ordering still need to be established.

Ref. R807, Crash Dump Analysis Patterns (Part 254). See also: Stack Trace Collection; Critical Region; Data race.

Stack Trace Set

Memory patterns. A collection of stack traces is reduced to distinct traces under a chosen comparison rule. The rule may compare complete module-and-function sequences or another stated property. We use the resulting set to expose different execution paths without repeated listings, while retaining thread identities and multiplicities separately when they matter. Different comparison rules can produce different sets.

Ref. R648, Stack Trace Set pattern and diagnostic examples. See also: Stack Trace Collection; Stack Trace Signature; Unified Stack Trace.

Stack Trace Signature

Memory patterns. A compact representation of a stack's shape, for example module names and frame counts grouped into lower setup, central work, and upper completion or system-call portions. We use that representation to compare stacks when detailed symbols are unavailable or unnecessarily specific. Matching signatures can identify related execution shapes, while preserving the distinction between structural resemblance and a shared defect.

Ref. R803, Crash Dump Analysis Patterns (Part 242). See also: Crash Signature; Module Stack Trace; Quotient Stack Trace.

Stack Trace Surface

Memory patterns. A sequence of stack-trace collections forms a view of how execution and dependencies change across memory snapshots. We compare threads, wait chains, and process identities over that surface, looking for changing endpoints or restarts. The term expresses a temporal arrangement of dump evidence; continuity still depends on capture consistency and correct identity matching between snapshots.

Ref. R799, Crash Dump Analysis Patterns (Part 229). See also: Stack Trace Collection; Spike Interval; Mirror Dump Set.

Stack unwinding

Debuggers. The reconstruction of preceding call frames from a current execution context using the platform’s calling rules and available unwind information. It allows a debugger to present callers even when a simple frame-pointer chain is absent. We check whether the required registers and stack memory are present. An interrupted prologue, incompatible metadata, or corrupted return state can make the reconstructed chain incomplete or misleading.

Ref. R131, Unwind data for exception handling and debugger support; struct UNWIND_INFO. See also: Stack trace [Artifacts]; Calling convention; Register context.

Staleness

Metrics. The condition in which previously reported metric data no longer represents a current series. Prometheus applies lookback and stale-series rules when evaluating queries, so a disappeared target need not produce a zero-valued sample. The disappearance itself can be evidence. Filling every gap with zero changes the question and can conceal collection failure, label changes, or the removal of a service instance.

Ref. R272, Gotchas: Staleness. See also: Time series; Scrape; Telemetry loss.

Startup probe

Cloud. A Kubernetes check that gives an application a defined opportunity to complete initialization before liveness and readiness probing proceeds. Once startup succeeds, the ordinary probes take over; sustained startup failure can cause restart. It is useful when initialization duration differs from normal operation. Interpret repeated failures against the startup budget and initialization evidence rather than assuming the running application later became unhealthy.

Ref. R293, Startup probe. See also: Liveness probe; Readiness probe; CrashLoopBackOff.

Starvation

Execution. Failure to obtain the resources or scheduling opportunities needed for progress while other work continues. It can involve CPU time, workers, locks, or other limited resources. We examine who receives service and for how long, rather than relying only on a snapshot. A thread that appears ready or repeatedly waiting may be eligible for work without receiving a useful opportunity to complete it.

Ref. R102, Starvation; Livelock. See also: Priority inversion; Thread pool; Deadlock.

State and Event

Trace patterns. A distinction between records describing system state and records describing events that act on or change that state. The same event can have different consequences in different states. Reading the two together supports reconstruction of behavior in event-driven systems and helps avoid treating a status report as a transition.

Ref. R002, pp. 283–284. See also: State Dump; Message Change; Phantom Activity.

State Dump

Trace patterns. Selected state variables written into a logging stream at particular moments or intervals. This provides explicit state evidence when a memory dump is unavailable or insufficient. The recorded selection and timing constrain the reconstruction, especially when values are read separately while other threads continue to modify them.

Ref. R002, p. 285. See also: State and Event; Counter Value; Significant Interval.

Statement Density and Current

Trace patterns. Statement density is the fraction of recorded messages belonging to a selected message type or set; statement current is the number of selected or all messages per unit time. The measures describe different properties. Comparisons require consistent selection and intervals, because logging scope can change either quantity without a matching change in useful work.

Ref. R002, pp. 286–289. See also: Relative Density; Activity Region; Trace Acceleration.

Static analysis

Program analysis. Analysis of source code or another program representation without executing the target program on concrete test inputs. It can examine paths that tests have not reached and identify properties or suspicious operations. The meaning of a report depends on the analysis model and supported features. Neither every warning nor every clean result has the same evidential strength across different analyzers.

Ref. R151, Analyzer overview and analysis approach. See also: Abstract interpretation; Clang Static Analyzer; Dynamic analysis.

Statistical debugging

Testing. Analysis that relates observations collected across executions to success or failure. Instrumented predicates, value relationships, or other features can reveal conditions strongly associated with a problem. Population composition and sampling affect the result, and several defects may produce different failure groups. A useful predictor guides the next diagnostic experiment; it does not establish that changing the predictor will remove the failure.

Ref. R157, Statistical predicates and failure predictors. See also: Fault localization; Spectrum-based fault localization; Dynamic analysis.

Step Dumps

Memory patterns. A sequence of dump-file sizes forms plateaus separated by steps. Size changes can help prioritize captures when examining a long sequence, especially when inspecting every dump is expensive. We compare artifacts with compatible capture settings and then examine state around the changes. Equal file sizes do not establish that memory contents or application behavior remained unchanged.

Ref. R664, Step Dumps pattern and diagnostic examples. See also: Memory dump; File Size; Memory Fluctuation (process heap).

Stop-the-world pause

Runtimes. An interval during which a runtime suspends application execution for work requiring a coordinated view of state, commonly a phase of garbage collection. A concurrent collector can still contain such intervals. Their significance depends on duration and placement relative to the affected operation. Total collector CPU time, aggregate collection duration, and application suspension time are different measurements.

Ref. R228, Collector phases and latency. See also: Garbage collection; JFR; Go execution trace.

Stored Exception

Memory patterns. An exception record remains available in the dump even when its original thread or register context cannot be restored. The record can still identify an exception code, address, and associated parameters. We preserve this evidence while stating what is missing. Recovering the record alone does not recover the faulting stack, register values, or all surrounding execution state.

Ref. R658, Stored Exception pattern and diagnostic examples. See also: Exception record; No Current Thread; Exception Stack Trace [Memory patterns].

strace

Native tools. A tool for observing Linux system calls and signals, including selected arguments, return values, and timing information. It reveals interactions across the process–kernel boundary that may explain missing files, failed operations, or waiting. We distinguish observed system-call duration from a complete explanation of its causes. Tracing can perturb timing, and operations performed entirely in user space are outside this particular view.

Ref. R063, DESCRIPTION; OPTIONS. See also: System call [Execution]; errno; ptrace.

Strand of Activity

Trace patterns. A combination of threads or adjoint activities of the same identifier type, such as several thread-based views considered together. It provides an intermediate level between individual activities and combinations of different analytical dimensions. The grouping retains the distinct identities of its constituent activities.

Ref. R002, p. 290. See also: Thread of Activity; Cord of Activity; Fiber of Activity.

String Hint

Memory patterns. A readable string whose content appears unexpected for the containing module or process and therefore supplies an investigation lead. Examples include unfamiliar service names, forms, credentials-related labels, or network locations. We validate the string's origin and surrounding data before inferring purpose. Presence in memory does not establish that the corresponding operation was performed.

Ref. R840, Pattern discussion. See also: Namespace; Module Hint; Execution Residue.

String Parameter

Memory patterns. A function parameter exposes useful textual context directly or through a structure containing string fields. File names, operation names, or other strings can connect a stack frame to the activity being investigated. We inspect the pointed-to storage using the correct character interpretation and validate its lifetime. Readable text is a clue whose relevance still depends on the call and object context.

Ref. R637, String Parameter pattern and diagnostic examples. See also: Parameter Flow; Stack trace [Artifacts]; Module Hint.

Structural pattern

Foundations. A recurrent arrangement of software entities and relationships, such as processes, threads, modules, memory regions, or object links. Structural patterns can describe normal organization and provide the context needed to recognize abnormality. A structure observed in one snapshot may constrain behavioral explanations without uniquely identifying the sequence of operations that created it.

Ref. R001, pp. 104–105. See also: Behavioral pattern; Memory dump analysis; Problem pattern.

Structured logging

Logs. Logging that represents selected information as explicit fields with defined names and values instead of requiring every distinction to be recovered from message text. Structure supports reliable filtering and correlation, but field presence does not guarantee consistent meaning or accurate values. A diagnostic schema should retain the operation, outcome, identity, and time needed to interpret an event within its execution context.

Ref. R332, Event attributes; data to exclude; event collection. See also: Log record; Correlation identifier; Semantic conventions.

Structure Field Collection

Memory patterns. A collection of selected field values from many objects of the same type, such as thread names or context-switch counts. We retain each object's identity so an unusual value can lead back to detailed inspection. This turns a repeated structural question into a comparable population and generalizes techniques such as collecting the last-error value from every thread.

Ref. R815, Crash Dump Analysis Patterns (Part 279). See also: Last Error Collection; Region Profile; Object Distribution Anomaly.

Structure Sheaf

Memory patterns. Debugger commands are applied across related memory areas or dumps, and their outputs are compared as a structured collection. The name is a mathematical analogy for organizing these views, extending stack-change comparison beyond one command or one temporal sequence. We retain the command, artifact, and region associated with each result so that differences can be interpreted in context.

Ref. R691, Structure Sheaf pattern and diagnostic examples. See also: Stack Trace Change; Step Dumps; Evental Dumps.

Subsystem Modules

Memory patterns. Several loaded modules belong to one plugin, framework, or other architectural subsystem. Their paths may suggest the grouping, while product design supplies a stronger basis for it. We use that grouping to compare stacks and historical evidence across component boundaries. Sharing a directory is a clue to subsystem membership, not a complete architectural description or causal explanation.

Ref. R686, Subsystem Modules pattern and diagnostic examples. See also: Module Collection; Module Variety; Historical Information.

Summary

Metrics. In Prometheus instrumentation, a metric type that records a count and sum and can calculate selected quantiles within the client over a configured time window. Those quantiles answer the client's configured question at collection time. They cannot generally be averaged or summed into a correct cross-instance quantile. The count and sum have different aggregation properties and should be interpreted separately.

Ref. R271, Summary; quantiles. See also: Histogram; Quantile; Aggregation.

Supporting Module

Debuggers. A reusable helper component shared by diagnostic tools or Troubleshooting Units of Work. It supplies recurring operations such as environment or configuration queries. We distinguish the helper's returned facts from the conclusions made by its callers, and keep its interface consistent so that several checks can reuse the same operation without silently changing its meaning.

Ref. R859, Pattern discussion. See also: Troubleshooting Unit of Work; Checklist; Debugware.

Surveyor

Trace patterns. Messages indicating that another tracing or logging mechanism was active. Such evidence can lead to additional artifacts that were not included in the initial collection. The discovered observer may have recorded a different scope or interval, so its relationship to the current trace must be established.

Ref. R002, p. 291. See also: Trace Extension; Inter-Correlation; News Value.

Suspended Thread

Memory patterns. A thread prevented from running by suspension rather than merely waiting for an application condition. Available suspend counts and thread context can expose an otherwise unexplained loss of progress. We identify who requested suspension and whether it belongs to debugging, capture, or application behavior. A suspended thread may hold resources needed elsewhere, but suspension observed during collection is not automatically evidence of the original problem.

Ref. R381, Suspend-state discussion and thread suspension examples. See also: Thread; Wait Chain; Manual Dump.

Swarm of Shared Locks

Memory patterns. A large population of shared locks clusters around particular structures, subsystems, or participating modules. We group the locks and inspect their owning-thread stacks, then compare processor and ready-thread state to understand whether work is progressing. Shared ownership is normally valid; the unusual concentration and surrounding evidence make the swarm diagnostically useful rather than inherently erroneous.

Ref. R623, Swarm of Shared Locks pattern and diagnostic examples. See also: High Contention; Wait Chain; Semantic Split.

Symbol file

Artifacts. An artifact that supplies names, types, source relationships, or other debugging information for a compiled component. Its exact contents depend on format and build settings. We match symbols to the captured binary before treating their interpretation as reliable. Symbol availability can improve a stack or object display, but cannot restore memory that the dump never preserved.

Ref. R025, Section 18.3; R104, Private symbol data and public symbol table. See also: PDB [Artifacts]; DWARF; Symbol server.

Symbolication

Debuggers. The conversion of recorded code addresses into symbolic descriptions such as module names, functions, and source locations. Reliable conversion requires appropriate binary identity, matching debug information, and correct load-address interpretation. We preserve these inputs with the original address evidence. A plausible function name produced from mismatched symbols can make a report easier to read while making its interpretation less reliable.

Ref. R034, Defining Load Addresses for Sections; Use Builtin Python Module to Symbolicate. See also: Symbol file; Build ID; dSYM.

Symbolic execution

Program analysis. Execution over symbolic input values, with program operations represented as expressions and decisions accumulated as constraints. It examines families of concrete executions and can produce inputs reaching selected conditions. The expressions are a model of computation, so external behavior and unsupported operations require explicit treatment. Path growth and solver difficulty limit how much of the program can be explored.

Ref. R148, Symbolic inputs, execution trees, and testing. See also: Concolic execution; Path condition; Path explosion.

Symbolless or Imageless Analysis
Foundations. Continuing an investigation without obtaining matching symbols or executable images when their absence prevents reliable interpretation. Raw offsets and debugger warnings can then be treated with unjustified confidence. We establish what the available metadata supports, obtain the missing material where possible, and state the remaining limits. Necessary analysis under incomplete evidence differs from overlooking a resolvable deficiency.

Ref. R871, Antipattern discussion. See also: No Component Symbols; Symbol file; Incorrect Stack Trace.

Symbol path
Debuggers. The configured search locations a debugger uses to find matching symbol information, including local directories, caches, and symbol-server specifications. In Windows debugging, this path controls an important part of artifact interpretation. We inspect both the path and the actual symbol-loading result. Changing the search path cannot compensate for a symbol file that was never retained or for a binary whose identity was not preserved.

Ref. R111, Symbol path syntax; Control the symbol path. See also: Symbol server; Symbol file; WinDbg.

Symbol server
Artifacts. A service or structured store from which a debugger retrieves symbols associated with particular binary builds. It helps maintain the relationship between deployed components and their debugging information. We record the configured source and verify the selected symbols. A successful download does not guarantee the presence of private types or source files, while a cache miss may reflect access or retention rather than an invalid binary.

Ref. R111, Symbol path syntax; Control the symbol path. See also: Symbol path; Symbol file; Build ID.

Syslog
Logs. A standardized logging protocol with fields for priority, time, origin, application, and structured data. RFC 5424 separates the message representation from the transport used to deliver it. A syslog record can therefore be well formed while delivery remains lossy or delayed. Diagnostic interpretation must account for the sender’s clock, severity convention, forwarding path, and the actual transport configuration.

Ref. R310, Sections 5–6. See also: Log record; Log level; Telemetry loss.

Systematic diagnostics
Foundations. The organization of diagnostic knowledge into a coherent body of principles, methods, and pattern catalogs. It connects individual observations with reusable explanations and analysis practices across software construction and operation. Systematicity concerns how the knowledge is ordered and developed, while systemic diagnostics concerns relationships across the boundaries of artifacts and components.

Ref. R001, pp. 352–353. See also: Systemic diagnostics; Pattern-based diagnostics; Pattern-oriented diagnostics.

System Call

Memory patterns. A user-space stack ends at a system-call transition while the thread may be performing work or waiting inside the kernel. Its user-space appearance can therefore resemble an ordinary blocked thread. We inspect the transition and seek kernel context where available. The process dump's visibility limit must remain explicit when deciding whether the thread is idle, blocked, or active.

Ref. R684, System Call pattern and diagnostic examples. See also: Dual Stack Trace; User Space Evidence; Blocked Thread.

System call

Execution. An entry into operating-system services from a less-privileged execution context through the platform's defined interface. Its arguments, result, and duration can reveal resource access and blocking. We distinguish a system-call boundary from the complete application operation. One user request can involve many calls, while work inside a runtime or a user-space cache may require none.

Ref. R883, DESCRIPTION: system calls, C library wrappers, and the user/kernel transition; R063, DESCRIPTION; OPTIONS. See also: strace; Kernel mode; errno.

System Description Snapshot

Artifacts. A captured description of system properties, such as hardware, files, configuration, and registry information. Comparing descriptions across time or machines can reveal environmental changes relevant to an investigation. We record the collection scope and time. This descriptive snapshot complements execution and memory artifacts; it need not contain the actual memory state of the investigated process.

Ref. R862, Pattern discussion. See also: Changed Environment; Snapshot Collection; Configuration Wrapper.

Systemic diagnostics

Foundations. Analysis that applies the same general diagnostic patterns across artifacts, products, and platforms and follows their relationships across system boundaries. A process dump, service trace, network capture, and problem description may expose complementary parts of one situation. The investigation must preserve the timing and identity relationships that make cross-artifact comparisons meaningful.

Ref. R001, p. 219; pp. 264–266. See also: Systematic diagnostics; Trace Quilt; APM.

System Object

Memory patterns. An operating-system object visible in a named object directory provides evidence about system or product activity. Its state or name may identify a condition, reporting operation, or process of interest. We correlate that evidence with memory statistics and execution context, since an object name or signaled event acquires diagnostic meaning only through the mechanism that creates and uses it.

Ref. R775, Crash Dump Analysis Patterns (Part 158). See also: Structure Field Collection; Error Reporting Fault; Dump Context.

T

Tail latency

Performance. The slower end of a latency distribution, commonly examined through high percentiles and unusually long individual operations. It can dominate user experience when a request waits for several dependencies. An average can hide this behavior, while a single percentile can hide the size of the remaining tail. Investigating tail latency requires preserving the affected population and examining concrete executions within it.

Ref. R303, Abstract; tail effects in large-scale services. See also: Percentile; Histogram; Critical path.

Tail sampling

Telemetry. A tracing decision made after enough spans have arrived to evaluate a trace against a policy, such as retaining failures or slow requests. The decision requires buffering and a rule for when a trace is sufficiently complete. Late or missing spans can change the apparent outcome. Tail sampling cannot recover data discarded earlier and can bias the retained population toward selected behavior.

Ref. R258, Tail sampling. See also: Sampling; Head sampling; Telemetry loss.

Taint analysis

Program analysis. Tracking information from designated sources through propagation rules to designated sinks. It can identify a path by which untrusted input reaches an operation requiring validation. The labels describe an analysis policy, not an intrinsic property of a value. Missing sources, inappropriate sanitizers, or incomplete propagation rules can hide a relevant flow or report one that the implementation safely handles.

Ref. R152, Sources, propagations, filters, and sinks. See also: Data-flow analysis; DataFlowSanitizer; Static analysis.

Tampered Dump

Memory patterns. A memory dump has been deliberately modified so that its structures or apparent execution history support a misleading interpretation. Altered contexts, fabricated residue, or suppressed module evidence can resemble ordinary corruption. We test consistency across independent structures and preserved exception information, while keeping a distinction between evidence of inconsistency and evidence that someone intentionally changed the artifact.

Ref. R546, Pattern description: Tampered Dump. See also: Inconsistent Dump; Corrupt Dump; Execution Residue.

TCP zero window

Networks. A TCP receiver advertisement indicating that no additional receive-window space is currently available. The sender uses zero-window probing to discover when space becomes available again. This points toward receiver-side flow control rather than directly proving network congestion. A slow reader, scheduling delay, or another receiver condition may contribute, so correlate packet evidence with the receiving application and its resource state.

Ref. R307, Section 3.8.6.1: Zero-Window Probing. See also: Packet capture; Backpressure; Queue length.

Technology-Specific Subtrace (COM client call)

Memory patterns. A stack fragment identifies a COM client call and can guide reconstruction of its destination server and thread. The visible path may involve ALPC or a modal loop, requiring different evidence to recover the relationship. We correlate communication identities and raw-stack clues before extending a wait chain. The client stack alone need not reveal the final blocking operation.

Ref. R641, Technology-Specific Subtrace (COM client call) pattern and diagnostic examples. See also: Technology-Specific Subtrace (COM interface invocation); Wait Chain (LPC/ALPC); Wait Chain.

Technology-Specific Subtrace (COM interface invocation)

Memory patterns. A recognizable stack fragment reflects COM interface invocation within a larger execution path. The fragment may span several modules and overlap other technology-specific sequences, such as RPC, window messaging, or managed-runtime activity. We identify the operation represented by each subtrace before interpreting the whole stack. Recognizing a technology establishes execution context, not responsibility for a failure.

Ref. R640, Technology-Specific Subtrace (COM interface invocation) pattern and diagnostic examples. See also: Stack trace [Artifacts]; Implementation Discourse; Technology-Specific Subtrace (COM client call).

Technology-Specific Subtrace (dynamic memory)

Memory patterns. A recognizable sequence of stack frames indicates dynamic-memory activity such as allocation or release. We use that subtrace to interpret the surrounding operation and decide which memory structures require inspection. Corruption can cause a fault inside this path before an allocator explicitly reports damaged bookkeeping, so the subtrace is useful even without a dedicated heap or pool diagnostic.

Ref. R762, Crash Dump Analysis Patterns (Part 127b). See also: Dynamic Memory Corruption (process heap); Dynamic Memory Corruption (kernel pool); Constant Subtrace.

Technology-Specific Subtrace (JIT .NET code)

Memory patterns. A recognizable fragment of runtime-generated .NET calls within an otherwise native stack trace. With the matching CLR diagnostic extension, we resolve candidate JIT code addresses to managed methods and compare the resulting sequence with the managed stack. This connects the two representations while preserving the distinction between an active frame and an unverified address on a raw stack.

Ref. R517, Pattern description: Technology-Specific Subtrace (JIT .NET code). See also: JIT Code (.NET); Managed Stack Trace; Execution Residue.

Telemetry

Telemetry. Data emitted or collected to describe the operation of a system and made available for analysis. Telemetry can preserve individual observations or aggregate many observations into a smaller representation. Its diagnostic meaning depends on where it originated, how it was transformed, and what was omitted. The collection system is also software and can introduce gaps, delay, duplication, and misleading context.

Ref. R250, Signals. See also: Signal [Telemetry]; Telemetry pipeline; Telemetry loss.

Telemetry loss

Telemetry. The failure to preserve observations that were expected to reach an analysis destination. Loss can occur when queues fill, retries expire, a process exits, or a component rejects data. It differs from deliberate selection by a sampling policy, although both create gaps. To interpret an absent event, inspect pipeline health and retention boundaries as well as the software that should have produced it.

Ref. R263, Sending queues; retry mechanisms; persistent storage. See also: Sampling; Backpressure; Log retention.

Telemetry pipeline

Telemetry. The ordered path by which observations travel from their origin to an analysis destination. A pipeline can receive, enrich, batch, filter, sample, and export data, with each stage changing its availability or representation. We can reconstruct this path when evidence is missing or inconsistent. Processing order matters: removing an attribute before sampling can change which records survive.

Ref. R256, Pipelines; processing order. See also: Collector; Processor; Telemetry loss.

Template Module

Memory patterns. A module retains identifying material copied from a sample or template, making its apparent vendor or product information misleading. We compare image resources with timestamps, symbol paths, and related modules to establish more credible provenance. Template-derived metadata is a clue about how the binary was produced; it should not automatically assign responsibility to the author named in the copied material.

Ref. R755, Crash Dump Analysis Patterns (Part 108). See also: Deviant Module; Module Variety; Reduced Symbolic Information.

Tensor Trace

Trace patterns. An indexed arrangement of traces within a larger trace collection, such as messages indexed within runs and runs indexed within machines. The tensor analogy makes these levels explicit. It does not, by itself, establish the multilinear transformation properties of a mathematical tensor.

Ref. R002, pp. 292–293. See also: Serial Trace; Trace Dimension; Singleton Trace.

Termination message

Cloud. A short explanation made available with a Kubernetes container's terminated state, normally read from a configured file written by the container. A policy can use recent logs as a fallback in defined circumstances. The message is application-provided evidence rather than a guaranteed explanation of every termination. A forced kill can prevent the application from writing a useful final account.

Ref. R295, Writing and reading a termination message. See also: Container log; OOMKilled; Pod event.

Test-case reduction

Testing. Transformation of a failing test into a smaller or simpler case while preserving a specified interesting behavior. Removing unrelated code or input can expose the structure needed for interpretation and make repeated experiments cheaper. The preservation predicate must distinguish the original problem from an unrelated crash, timeout, or harness failure. Reduction preserves an observation, not automatically its original mechanism.

Ref. R154, Failure-preserving simplification. See also: Delta debugging; C-Reduce; Shrinking.

Test double

Testing. A substitute for a real collaborator used to control or observe an execution under test. It may provide canned results, maintain simplified behavior, or record calls; a mock is one implementation. The substitution creates a useful experimental boundary. Interpretation requires identifying which properties of the real collaborator were preserved and which timing, failure, or state interactions were omitted.

Ref. R172, Replacing objects during testing. See also: Mock; Hermetic test; Test harness.

Test fixture

Testing. The preparation and cleanup required to execute a test in a defined state. It may establish data, objects, resources, or configuration. A fixture is part of the experiment: unintended sharing, incomplete cleanup, or setup failure can explain an observed test outcome. Capturing its state helps distinguish a defect in the tested operation from a problem in how the test was arranged.

Ref. R165, Test fixtures; setUp and tearDown. See also: Unit test; Flaky test; Hermetic test.

Test harness

Testing. The code and execution arrangement that supply inputs to a target, control setup, and expose outcomes to a test engine. In fuzzing, the harness determines which interface and state transitions are reachable. Its own defects or omitted initialization can produce misleading failures. A useful diagnostic harness preserves the relevant behavior while making the failure predicate explicit and repeatable.

Ref. R162, Fuzz target interface and harness requirements. See also: Fuzzing; Test oracle; Reproducer.

Test oracle

Testing. A mechanism or source of judgment that distinguishes acceptable behavior from a failure for a test. It may be an assertion, specification, reference implementation, relation between runs, or informed human assessment. The oracle determines what the test can detect. A test that executes the relevant code but checks the wrong outcome can repeatedly pass while preserving the defect.

Ref. R164, Oracle problem and sources of expected behavior. See also: Assertion; Differential testing; Metamorphic testing.

Text Trace

Trace patterns. Text represented as a trace by assigning structural or semantic attributes to its units, and the reverse conversion of structured traces into text. Sentences, paragraphs, or selected vocabulary can supply analytical units. The chosen mapping determines which trace patterns are meaningful and which textual distinctions are preserved.

Ref. R002, pp. 294–295. See also: Denormalized Message; Trace Schema; Implementation Discourse.

Thinking-based diagnostics

Foundations. A diagnostic approach that makes reasoning methods explicit alongside tools and pattern recognition. Critical, systemic, and semiotic thinking can be combined with deduction, induction, and abduction. The practical purpose is to examine assumptions, distinguish evidence from interpretation, and choose analysis actions that can change what is known about the problem.

Ref. R001, p. 175. See also: Abduction; Diagnostic hypothesis; Causal model.

!thread

Debuggers. A Windows kernel-debugging extension command that displays a thread's operating-system state and related information, potentially including its stack. We connect the reported wait or execution state to the selected process and captured time. A thread waiting in a common kernel routine may be behaving normally; diagnosing a blocked dependency requires the relevant object, owner, and wider set of participating threads.

Ref. R110, Parameters; kernel-mode restriction. See also: Thread; Deadlock; Stack trace [Artifacts].

Thread

Execution. A schedulable execution context with its own registers and stack, usually sharing process resources with other threads. The earlier Dictionary of Debugging blog entry also models a thread by its instruction-pointer positions indexed by time. We relate the current stack to that execution history and to other threads; a waiting thread can participate in a healthy synchronization protocol.

Ref. R015, Processes, threads, thread pools, and fibers; R023, Section 4.10; R415, Definition. See also: Process; Stack trace [Artifacts]; Context switch.

Thread Age

Memory patterns. The elapsed interval between a thread's creation and the dump's capture, when the artifact preserves the necessary timing information. We compare that age with process uptime, incident chronology, and capture times from coupled processes. A recently created blocked thread may postdate another dump, which limits conclusions drawn by treating the two artifacts as simultaneous observations.

Ref. R522, Pattern description: Thread Age. See also: Waiting Thread Time; Coupled Processes; Early Crash Dump.

Thread Cluster

Memory patterns. Threads form groups according to selected diagnostic properties, such as waiting time and modules present in their stacks. The groups can expose long waits, common dependencies, or related activity that is hard to see in a full listing. We state the grouping dimensions and inspect representative members. Membership expresses similarity under those dimensions, not necessarily a causal relationship.

Ref. R668, Thread Cluster pattern and diagnostic examples. See also: Stack Trace Collection; Waiting Thread Time; Clustering.

Thread dump

Runtimes. A runtime-oriented snapshot of threads and their reported stacks, states, and sometimes lock information. It helps recognize repeated waiting, active computation, or related call contexts across threads. The snapshot is evidence of a collection interval, not a complete history. Repeated comparable dumps can distinguish a stable state from progress, while the runtime's state labels require interpretation in their own terms.

Ref. R214, Description; stack traces and lock information. See also: jstack; dotnet-stack; Go execution trace.

Thread-local storage

Execution. Storage that associates a separate value with each thread under a common access mechanism. It is often used for per-thread state, but does not automatically follow asynchronous work that moves between threads. We select the correct thread before inspecting it. An error value or cached pointer read from another thread’s storage can give a convincing but unrelated account of the problem.

Ref. R097, Thread-specific data and TLS slots. See also: Thread; Debugger context; errno.

Thread of Activity

Trace patterns. The ordered messages associated with one execution thread within its identity and lifetime. Filtering this activity removes interleaved messages from other threads and exposes its local sequence. A complete interpretation may still require other activities, especially when work crosses asynchronous or interprocess boundaries.

Ref. R002, pp. 296–297. See also: Adjoint Thread of Activity; Error Thread; Glued Activity.

Thread pool

Execution. A group of worker threads used to execute submitted work without creating a dedicated thread for every operation. The pool separates logical work items from the threads that run them. We examine queued work, worker availability, and blocking behavior together. A large queue may reflect slow dependencies or blocked workers rather than insufficient CPU capacity or an inherently small pool.

Ref. R134, Thread Pool Architecture; Best Practices. See also: Thread; Starvation; Deadlock.

Thread Poset

Memory patterns. An ordering of selected threads supports comparison between stack collections, potentially using creation times or list position. By analogy with a partially ordered set, the pattern preserves relationships that an unordered selection would lose. We compare thread roles and relative arrangement across runs. Similar ordering is an initial correspondence; individual stacks can still contain important behavioral differences.

Ref. R676, Thread Poset pattern and diagnostic examples. See also: Stack Trace Collection (predicate); Thread Age; Stack Trace Change.

ThreadSanitizer

Program analysis. A compiler-and-runtime instrument for detecting data races in executed code. It observes memory operations and recognized synchronization to identify accesses lacking the required ordering. The report connects participating operations and threads for investigation. It does not establish freedom from every concurrency defect, and missing instrumentation or unsupported synchronization can change its interpretation. Execution overhead can also alter which schedules occur.

Ref. R177, Introduction; usage and limitations. See also: Go race detector; Helgrind; Dynamic analysis.

Thread Starvation (normal priority)

Memory patterns. Runnable normal-priority work dominates available processor time while lower-priority ready threads make inadequate progress. We compare scheduling priorities, CPU usage, affinity, and the stacks of competing threads to explain the imbalance. The pattern concerns relative service and readiness; a thread blocked on an event is waiting for a different reason, even if the system is also busy.

Ref. R738, Crash Dump Analysis Patterns (Part 102). See also: High Contention (processors); Affine Thread; Busy System.

Thread Starvation (realtime priority)

Memory patterns. CPU-intensive threads at unusually high or realtime priority prevent lower-priority work from receiving sufficient processor service. We compare priorities, running threads, processor availability, and execution time rather than attributing every busy system to starvation. The pattern connects scheduling preference to missing progress; it does not require speculation about who changed the priority or why.

Ref. R616, Thread Starvation (realtime priority) pattern and diagnostic examples. See also: Starvation; Spiking Thread; Thread Starvation (normal priority).

Throughput

Performance. The amount of defined work completed per unit of time, such as successful requests or processed records. The counted unit and success condition must be explicit: admitted, attempted, and completed work are different quantities. High throughput does not establish acceptable latency, and a plateau may indicate capacity, a downstream limit, or reduced demand. Compare throughput with offered load and unfinished work.

Ref. R291, Handling Overload. See also: Rate; Queue length; Saturation.

Time Delta

Trace patterns. The measured interval between selected trace messages or events. It can identify delays and compare phases of an operation. The interpretation depends on the timestamps' clock, resolution, and event meaning; elapsed time between messages does not necessarily equal processor time spent by the activity.

Ref. R002, p. 298. See also: Delay Dynamics; Time Scale; Unsynchronized Traces.

Timeout

Trace patterns. Repeated gaps or delay intervals whose similar duration suggests expiration of a waiting limit. Associated error messages and backward tracing can identify the relevant operation. The pattern is an observed timing form; establishing a configured timeout requires supporting status, configuration, code, or other execution evidence.

Ref. R002, pp. 300–302. See also: Discontinuity; Time Delta; Periodic Error.

Timeout

Distributed systems. A limit on how long an operation or wait is permitted to continue before the observer reports expiry or takes another action. Expiry establishes that the required completion was not observed within that boundary. It does not establish whether remote work failed, completed, or is still running. Connect, read, request, and overall timeouts cover different intervals and must not be silently substituted.

Ref. R284, Timeouts. See also: Deadline; Retry; Partial failure.

Time Scale

Trace patterns. The temporal resolution or grouping interval used to view and analyze a trace. Coarser scales combine more messages and may expose large activity structure, while finer scales reveal local gaps and ordering. A conclusion about density or silence should identify the scale on which it was obtained.

Ref. R002, p. 299. See also: Silent Messages; Activity Packet; Renormalization.

Time series

Metrics. A sequence of observations associated with time and a stable identity. In Prometheus, the metric name and complete label set identify a series; changing a label value creates a different series. Apparent discontinuities can therefore arise from identity changes as well as behavior. Analysis should preserve units, sample timing, and the distinction between an absent sample and a measured zero.

Ref. R269, Metric names and labels; samples. See also: Label; Staleness; Metric.

Time Travel Debugging

Debuggers. A WinDbg facility for recording supported user-mode execution and replaying it for investigation. It allows movement through recorded execution and queries about events such as memory access. We use the recording to connect a later symptom with earlier state changes. Recording scope, supported behavior, and collection overhead still matter; a replay can explain the recorded run without establishing that every production run follows the same path.

Ref. R088, What is Time Travel Debugging. See also: Reverse debugging; Watchpoint; WinDbg.

TLS alert

Networks. A protocol message used to communicate closure or an error during a Transport Layer Security exchange. The alert description identifies a protocol-level condition, whose meaning depends on the negotiated version and handshake stage. A closure alert is not automatically a failure. Diagnose the peer that sent the alert and its local context; the alert alone may not expose the configuration or certificate condition that prompted it.

Ref. R309, Section 6: Alert Protocol. See also: Packet capture; Timeout [Distributed systems]; HTTP status code.

Toil

Reliability. Operational work that is repetitive, largely automatable, reactive, and does not produce lasting improvement, often growing with service size. The term does not cover every manual activity. A new investigation can create knowledge even when it requires substantial hands-on work. Repeatedly performing the same diagnosis or workaround suggests an opportunity to improve instrumentation, repair the mechanism, or automate a stable procedure.

Ref. R289, Toil Defined. See also: SRE; Runbook; Postmortem.

Tool-augmented language model

AI and ML. A language-model system that can request external operations and use their returned results. In diagnostics, tools can provide real source text, test outcomes, or runtime observations that the model could not establish from generation alone. The result must be associated with the actual tool call and its scope. A suggested command or narrated action is not evidence that an operation occurred.

Ref. R246, API selection, calls, and returned results. See also: Agentic debugging; LLM-assisted debugging; RAG.

Tool Box

Debuggers. A collection of diagnostic tools presented through a common launcher or access point. The tools can be selected independently rather than executed in a prescribed sequence. We choose the tool according to the current question and available artifacts. A convenient collection supplies access to capabilities; it does not itself define the investigation method.

Ref. R855, Pattern discussion. See also: Tool Chain; Tool Façade; Analysis pattern.

Tool Chain

Debuggers. A sequence of related tools arranged to achieve a diagnostic goal, such as enabling logging, displaying activity, and capturing a dump. The relationship need not be a simple data pipeline. We identify the order, dependencies, and artifacts passed between steps, because a missing or mistimed operation can change the evidence available later.

Ref. R854, Pattern discussion. See also: Tool Box; Debugware; Diagnostic artifact.

Tool Façade

Debuggers. An interface placed over an existing diagnostic tool to make its capabilities easier to use in a particular workflow. A graphical front end for a command-line engine illustrates the pattern. We retain the underlying operation and its parameters in the diagnostic account, so a simplified interface does not obscure what evidence the tool actually obtained.

Ref. R851, Pattern discussion. See also: Debugware; Dual Interface; Tool Chain.

Top Module

Memory patterns. A module selected by its position nearest the top of a displayed stack, often the first component of interest after operating-system frames. We use that position to prioritize module information and nearby code for inspection. The designation also applies to an incorrect stack; any claim about actual execution or responsibility still requires validation.

Ref. R761, Crash Dump Analysis Patterns (Part 125). See also: Blocking Module; Exception Module; Incorrect Stack Trace.

Trace Acceleration

Trace patterns. An increase or decrease in message rate across successive regions of a trace or selected activity. The physical analogy highlights change in statement current, rather than the current itself. Region boundaries and measurement windows influence the result, so a visible change should be checked against capture and instrumentation changes.

Ref. R002, p. 303. See also: Statement Density and Current; Activity Region; Phase Transition.

Trace acquisition pattern

Telemetry. A reusable approach to collecting trace or log evidence for a particular diagnostic context. The catalog addresses placement, timing, use-case coverage, complementary captures, and selection of full, tuned, first-occurrence, or differential strategies. We connect these choices with the question under investigation, because acquisition determines which activities and relationships can later be observed.

Ref. R001, p. 177. See also: Instrumentation; Software narrative; Negative evidence.

Trace Amplitude

Trace patterns. A family of trace paths between fixed activity regions whose intervening regions may differ. Several Trace Strings can belong to the same amplitude because both the route and its internal sequence can vary. Borrowed from the path-integral analogy, the pattern organizes alternative execution narratives without assigning them quantum probabilities or asserting that every imagined route is feasible.

Ref. R431, Part 236. See also: Trace String; Activity Region; Feynman Trace.

Trace Benchmark

Trace patterns. An assessment of a trace after analysis against declared diagnostic criteria. Possible dimensions include relevance to the incident, clarity, collection overhead, sufficiency, sensitive content, and contribution to resolution. The original proposal leaves the criteria open. We record what each score measures and how it was assigned; a benchmark is an evaluation of usefulness, while a Trace Summary supplies preliminary information.

Ref. R429, Part 234. See also: Trace Summary; Canonical Trace; News Value.

Trace Bias

Trace patterns. An assessment of how faithfully a trace, considered as a model, represents the computation that produced it. Instrumentation choices and omitted activity affect what the model can express. The term borrows the idea of model bias from statistics and machine learning. We examine the relevant representational omissions rather than assuming that more messages necessarily provide a complete account.

Ref. R437, Part 242. See also: Declarative Trace; Moduli Trace; Trace Variance.

Trace Braidoids

Trace patterns. A braid-inspired representation that permits activities to begin or end inside the recorded trace boundaries. It makes partial activity lifetimes visible instead of assuming that every thread extends beyond both ends. Different choices of execution or adjoint identifiers produce different braidoid views.

Ref. R002, pp. 304–305. See also: Braid Group; Trace Fabric; Thread of Activity.

Trace Capacitance

Trace patterns. Accumulation of internal conditions before a conspicuous change appears in the trace. Pending work, resource pressure, retry state, or other quantities may grow while ordinary messages continue, followed by a burst of errors or a phase transition. The analysis searches backward for the accumulating condition and the threshold or mechanism behind its visible manifestation.

Ref. R008. See also: Trace Reactance; Trace Inductance; Phase Transition.

Trace Class

Trace patterns. A coarse classification obtained by replacing large trace structures with a small alphabet of conceptual units, such as ordinary activity, bad activity, discontinuity, and crash evidence. Consecutive units of the same kind may be collapsed. We use the resulting sequence to compare broad behavior while retaining the classification rules; identical class strings can conceal different detailed mechanisms.

Ref. R432, Part 237. See also: Activity Region; Discontinuity; Quotient Trace.

Trace Constants

Trace patterns. Values treated as constants within an implementation convention, a trace, or a diagnostic scenario. Error codes can have a shared namespace, while a user or host value may be constant only within one capture. Distinguishing these scopes improves normalization, comparison, and search for a message's diagnostic essence.

Ref. R002, p. 306. See also: Message Essence; Message Invariant; Random Data.

Trace Context

Trace patterns. The case information accompanying a trace: the reported issue, collection procedure, system circumstances, related artifacts, earlier traces, and previous analysis. A checklist can make missing context visible before interpretation begins. This pattern concerns situational knowledge around the artifact. We distinguish it from the identifiers and propagation fields used as distributed-tracing context within individual requests and spans.

Ref. R449, Part 254. See also: Trace Summary; Adjoint Space; Diagnostic report; Span context.

Trace Contour

Trace patterns. A structural outline obtained from a combed trace by retaining occupied activity positions and suppressing other message content. The contour exposes the arrangement of activity across selected dimensions. It supports shape comparison while requiring a route back to the messages for semantic interpretation.

Ref. R002, p. 307. See also: Combed Trace; Trace Shape; Trace Fabric.

Trace D'Enfant

Trace patterns. A graph representation relating messages in a causal history to the activity regions through which they pass, using two kinds of vertices. The name borrows the dessin-d'enfant analogy from mathematics. Algebraic interpretations require their own structural definitions beyond the diagnostic diagram.

Ref. R002, p. 308. See also: Causal History; Activity Region; Message Complex.

Trace Dimension

Trace patterns. The organizational dimensions across which recorded activity is distributed, such as processes, sessions, machines, or environments. Additional boundaries often introduce infrastructure messages. Reproducing a problem with fewer dimensions can simplify the evidence, provided the essential behavior remains reproducible.

Ref. R002, pp. 309–310. See also: Tensor Trace; False Positive Error; Use Case Trail.

Trace Divergence

Trace patterns. A structural count of activity identities corresponding to a selected adjoint thread ID (ATID), interpreted as incoming or outgoing relationships under a declared construction rule. Examples include threads associated with a process or sources targeting one destination. We specify direction, scope, and identity handling. This pattern measures the constructed relationship structure; Activity Divergence instead examines changing activity over time.

Ref. R451, Part 256. See also: Adjoint Thread of Activity; Trace Network; Activity Divergence; Message Flow.

Trace Expert

Debuggers. A lightweight rule-based facility that examines formatted trace messages and returns diagnostic guidance. Searches or regular expressions can recognize known textual conditions and suggest further work. We retain the matched evidence and the rule's scope when evaluating the result. A recognized message form supports a hypothesis only within the context for which the rule applies.

Ref. R856, Pattern discussion. See also: Software narrative; Analysis pattern; Diagnostic hypothesis.

Trace Extension

Trace patterns. A message's reference to another trace or to additional tracing that can provide missing detail. Following the extension broadens the evidence available for the same activity. The referenced artifact may not exist yet, may require collection, or may cover a different interval, so the reference itself is only a lead.

Ref. R002, p. 311. See also: Surveyor; Adjoint Space; Fiber Bundle.

Trace Fabric

Trace patterns. The structural fabric formed by activity strands in a combed trace after other message content is removed. It makes interleaving, gaps, and activity lifetimes visible and can be represented in a stave-like arrangement. The fabric is a projection whose interpretation depends on the chosen activity identifiers.

Ref. R002, pp. 312–313. See also: Combed Trace; Trace Contour; Trace Foliation.

Trace Field

Trace patterns. A mapping from trace messages to analytical values, which may be numeric, categorical, or structured. The assigned values support another view of the same message domain. A categorical treatment needs specified objects, morphisms, and preserved structure; a useful mapping alone does not automatically constitute a functor.

Ref. R002, p. 314. See also: Trace Flux; Semantic Field; Measurement.

Trace Flux

Trace patterns. External or global quantities associated with trace positions or intervals, such as system thread counts or context-switch measurements. These quantities add a wider system view to local messages. Their sampling times, scope, and units must remain explicit when they are correlated with individual activities.

Ref. R002, p. 315. See also: Trace Field; Counter Value; Inter-Correlation.

Trace Foliation

Trace patterns. A separation of a trace into compatible subtraces that retain its structure and scale, for example by message category. The leaves can be analyzed separately and recombined through a suitable mask. Their selection and alignment determine whether the original trace can be reconstructed without loss.

Ref. R002, pp. 316–317. See also: Trace Mask; Trace Fabric; Message Set.

Trace Frames

Trace patterns. A framing of trace regions according to their openings, endings, and embedded intervals. Missing introductory or terminal portions can expose incomplete capture or alter interpretation. The pattern organizes the surrounding narrative structure and differs from an operating system stack frame or a fixed-size transport frame.

Ref. R002, pp. 318–319. See also: Trace Partition; Truncated Trace; Message Directory.

Trace Graph

Trace patterns. A plotted representation constructed from trace data, for example by converting Trace Field values or selected activities into a series. Its interpretation depends on the horizontal coordinate, value mapping, and treatment of irregular time intervals. We preserve those choices when comparing plots. This pattern concerns graphing evidence; a Trace Network instead constructs explicit relationships among diagnostic entities.

Ref. R430, Part 235. See also: Trace Field; Time Scale; Trace Network.

Trace Homotopy

Trace patterns. Comparison of execution paths or code changes that retain selected starting and ending states. The homotopy analogy directs attention to variations between fixed endpoints. Equal endpoints do not imply identical intermediate behavior, timing, or safety properties, which remain subjects of the analysis.

Ref. R002, p. 320. See also: Master Trace; Meta Trace; Trace Path.

Trace identifier

Tracing. An identifier used to associate spans with the same distributed trace. It provides a practical join key across services and, when recorded, across logs and exemplars. Its diagnostic value depends on consistent generation and propagation. Reused, malformed, or untrusted identifiers can create misleading associations. A shared trace identifier alone does not establish the order or causal dependency of every recorded operation.

Ref. R264, trace-id. See also: Span identifier; traceparent; Log correlation.

Trace Inductance

Trace patterns. Persistence of earlier behavior after a new input or condition, delaying the appearance of a changed trace pattern. Queues, retries, caches, pools, or background work can retain effects of the previous state. We look for the intervening mechanism and correlated evidence of the initiating change, which may be recorded in another artifact.

Ref. R008. See also: Trace Reactance; Trace Capacitance; Causal History.

Trace Join

Trace patterns. A derived trace combining selected records from traces that satisfy a stated matching condition, such as a shared activity or feature identifier. The source schemas need not be identical. A join needs explicit identity and multiplicity rules, and without shared ordering information it need not produce a global chronological sequence.

Ref. R002, p. 321. See also: Linked Messages; Trace Mask; Trace Schema.

Trace Lattice

Trace patterns. A selection of trace messages separated by a fixed distance in record order or another declared metric. The lattice exposes regularly spaced positions for comparison or further analysis. We retain the distance rule and starting position because they determine which evidence is selected. It differs from Time Scale, which uses time intervals and may require additional transformations.

Ref. R436, Part 241. See also: Time Scale; Message Set; Trace Schema.

tracemalloc

Runtimes. A Python facility that records selected memory allocations with traceback information and supports snapshots and comparisons. It can locate code contributing to growth between observations or report an allocation's origin. Tracing begins only when enabled and does not necessarily cover memory allocated outside its tracked domains. Snapshot differences therefore describe the observed allocation population, rather than all memory held by the process.

Ref. R219, Tracing allocations; snapshots and comparisons. See also: Allocation profiling; Heap snapshot; Reference cycle.

Trace Mask

Trace patterns. A superposition of traces into a combined representation using compatible alignment information. One trace can impose useful structure on another with a different message rate or scale. This produces a new analytical artifact, unlike correlation that may only compare selected records without merging them.

Ref. R002, pp. 322–323. See also: Inter-Correlation; Trace Join; Time Scale.

Traceme

Trace patterns. A smallest meaningful unit of trace interpretation within a message, such as an object address together with its semantic role or a reference-count value. One message can contain several tracemes. The unit is semantic rather than a fixed number of characters or bytes.

Ref. R002, p. 347. See also: Multidimensional Message; Message Invariant; Trace Molecule.

Trace Model

Trace patterns. A trace or log whose messages satisfy the pattern structure represented by a Trace Sketch. Several different artifacts can realize the same sketch and therefore be regarded as equivalent under that chosen abstraction. Borrowed from model-theoretic language, this relationship supplies one notion of Trace Similarity. Matching overall Trace Shapes alone does not establish equivalence under the sketch.

Ref. R442, Part 247. See also: Trace Sketch; Trace Similarity; Trace Shape.

Trace Molecule

Trace patterns. A graph-like arrangement of tracemes associated with message complexes, using connections between selected message elements. The molecular analogy exposes how locally meaningful units form a larger diagnostic structure. It differs from treating whole analysis patterns as molecules in a separate artificial-chemistry model.

Ref. R002, pp. 324–325. See also: Traceme; Message Complex; Message Bond.

Trace Multiphysics

Trace patterns. Analysis of several coupled Trace Fields defined over the same trace domain. Field values can come from the primary messages or correlated artifacts, and their interactions may be immediate, delayed, constraining, or reinforcing. We need evidence for the coupling: simultaneously high field values do not establish that one field explains or influences another.

Ref. R007. See also: Trace Field; Trace Reactance; Trace World.

Trace Nerve

Trace patterns. A thread or adjoint activity that passes through all the selected activity regions of a trace. It supplies a continuous analytical route connecting those regions. There can be several such nerves, and their existence depends on how regions and activity identifiers have been chosen.

Ref. R002, p. 326. See also: Activity Region; Adjoint Thread of Activity; Trace Path.

Trace Network

Trace patterns. An attributed network constructed from trace evidence using diagnostic identities as vertices and an explicit relationship rule for edges. Relationships may represent requests, spawning, shared resources, or causality. Scope, aggregation, direction, and weighting determine the resulting network. We can inspect hubs and connected regions, but a correlation edge does not become a causal edge merely because it appears in the diagram.

Ref. R452, Part 257. See also: Trace Divergence; Trace Graph; Causal History; Message Complex.

Trace Ontology

Trace patterns. A structured interpretation of the entities, classes, relationships, and constraints expressed or implied by trace messages. Finding expected identifiers is only a beginning: we also establish what they identify and how those entities relate. The resulting model supports diagnostic inference while making the assumptions behind that inference available for examination.

Ref. R004. See also: Basic Facts; Trace World; Semantic Field.

Trace Parameters and Hyperparameters

Trace patterns. A machine-learning analogy that treats trace messages and their assigned values as model parameters, while tracing configuration and Trace Schemas act as hyperparameters. Values may be numerical or linguistic, and a CoTrace can represent a further analysis layer. We state the intended mapping explicitly; the analogy does not imply that ordinary logging performs neural-network training or backpropagation.

Ref. R445, Part 250. See also: Trace Field; Trace Schema; CoTrace; Trace Bias.

traceparent

Tracing. The W3C Trace Context header that carries a format version, trace identifier, parent identifier, and trace flags between services. It gives the receiver information for continuing a trace. The parent identifier denotes the sender's operation in this propagation context, rather than a permanent application hierarchy. Diagnostic tools should validate the header and preserve the distinction between incoming context and the span created locally.

Ref. R264, traceparent header. See also: Trace identifier; Span identifier; tracestate.

Trace Partition

Trace patterns. An analytical division into head, prologue, core, epilogue, and tail around the activity being investigated. Markers and known reproduction steps help establish the boundaries. Comparing partitions focuses attention while preserving uncertainty about whether an absent opening or ending reflects truncation or a legitimate scenario difference.

Ref. R002, pp. 327–328. See also: Anchor Messages; Trace Frames; Truncated Trace.

Trace Path

Trace patterns. A path through messages of interest formed by analytical relationships and navigation. It can support an explanation or connect selected observations in forward order, complementing backward tracing. The path should retain the relation used for each step so that analytical movement is not mistaken for execution flow.

Ref. R002, p. 329. See also: Back Trace; CoTrace; Explanation Trace.

Trace Plan

Trace patterns. The requirements for what tracing and logging should reveal about execution. We compare the resulting artifact with that plan to examine both software behavior and the adequacy of instrumentation. Discrepancies can guide changes to tracing code or collection. The plan belongs to trace engineering: it states the intended evidence before the generated trace is inspected.

Ref. R450, Part 255. See also: Declarative Trace; Trace Schema; Instrumentation; Trace Bias.

Trace Polynomial

Trace patterns. An ordered symbolic representation that groups a message sequence into abstract monomials and combines those groups into a trace expression. Message concatenation and concatenation of trace fragments are represented at different levels. Repeated fragments may receive coefficients. We preserve sequence order and the chosen grouping rule; familiar commutative or distributive polynomial laws are not automatically valid for this proposed notation.

Ref. R447, Part 252. See also: Canonical Trace; Message Pattern; Quotient Trace.

Trace Precision and Recall

Trace patterns. Proposed measures of diagnostic selection. Trace Precision compares messages used in successful diagnostics with all selected messages; Trace Recall compares them with all relevant messages, including relevant activity omitted from the trace. We declare the analysis task, relevance judgment, and denominator. Recall is difficult to establish when the relevant missing population is unknown, even when the selected evidence proved useful.

Ref. R446, Part 251. See also: CoTrace; Trace Bias; Precision; Recall.

Trace Presheaf

Trace patterns. Selected memory fragments associated with trace messages, for example the structures referenced by logged pointers. The association supplies local state without requiring a full dump for every event. Presheaf is the organizing analogy; a formal construction would additionally specify the domains and consistent restriction maps.

Ref. R002, p. 330. See also: Adjoint Space; State Dump; Fiber Bundle.

Trace Pressure

Trace patterns. A thermodynamic analogy for comparing trace regions using Trace Temperature, Trace Volume, and the number of non-Silent Messages. The source invokes an ideal-gas metaphor and illustrates regions with approximately equal pressure. We make each quantity's interpretation and scale explicit before comparing regions. The analogy does not establish a physical law for software behavior.

Ref. R435, Part 240. See also: Trace Temperature; Trace Volume; Silent Messages.

Trace Quilt

Trace patterns. A combined diagnostic view in which traces, memory evidence, configuration, and other artifacts are connected through activity relationships. The quilt metaphor emphasizes both the distinct patches and their connections. Each patch retains its provenance and capture interval so that the combined interpretation does not erase incompatible viewpoints.

Ref. R002, p. 331. See also: Trace Fabric; Adjoint Space; Small DA+TA.

Trace Reactance

Trace patterns. An umbrella pattern for the ways system structure delays, smooths, accumulates, or releases behavior before it becomes visible in trace evidence. Its specializations distinguish delayed response from accumulated conditions that later become apparent. The electrical analogy directs us to the intervening state and mechanisms rather than equating the first visible error with the beginning of the problem.

Ref. R008. See also: Trace Inductance; Trace Capacitance; Delay Dynamics.

Trace Retract

Trace patterns. A reduced representation using silent positions and standardized or grouped messages while retaining selected non-silent message positions. It supports structural comparison after message content has been simplified. The reduction must state which positions and relationships remain fixed; the topological name does not substitute for those definitions.

Ref. R002, p. 332. See also: Trace Skeleton; Silent Messages; Quotient Trace.

Trace Schema

Trace patterns. The arrangement and meaning of fields used to represent trace messages, including activity identifiers, feature identifiers, and message content. A schema makes selection, comparison, and joining explicit. Similar column names do not guarantee compatible meanings, units, or identifier scopes across different producers.

Ref. R002, pp. 333–334. See also: Definition Trace; Message Metadata; Trace Join.

Trace Shape

Trace patterns. The structural form of a trace or selected activity after some content differences have been abstracted away. Unrelated systems may exhibit similar shapes. Such similarity can suggest a reusable analysis approach, while the underlying messages and system mechanisms determine whether the same diagnosis is applicable.

Ref. R002, p. 335. See also: Trace Contour; Trace Skeleton; Trace Similarity.

Trace Sharding

Trace patterns. Division of a large trace into parts for parallel or otherwise distributed analysis. Some patterns cross shard boundaries, so independent local processing can miss relationships or double-count boundary records. The partitioning strategy must specify overlap, ordering, and how cross-boundary evidence will be reunited.

Ref. R002, p. 336. See also: Split Trace; Trace Join; Causal History.

Trace Similarity

Trace patterns. Closeness between traces under a stated representation and comparison measure. Message-based, activity-based, and error-focused comparisons can produce different results for the same artifacts. We choose the measure for the diagnostic question and retain the distinction between structural similarity and equivalence of behavior or cause.

Ref. R002, pp. 337–338. See also: Equivalent Messages; Trace Shape; Master Trace.

Trace Skeleton

Trace patterns. A trace representation that suppresses message content while preserving the lengths of activity blocks and their occupied positions. Unlike collapsing blocks to one representative, the skeleton retains their extent. It supports comparison of rhythm and structure while requiring the original trace for semantic interpretation.

Ref. R002, pp. 339–340. See also: Quotient Trace; Trace Retract; Trace Shape.

Trace Sketch

Trace patterns. A schematic representation that retains selected trace structures while omitting message detail. Dia|gram illustrations provide examples: a debugger log may be drawn as a sequence of Activity Regions, or reduced to a Trace Skeleton. We state which properties the sketch preserves because the drawing can guide recognition without reproducing the full evidence from which it was constructed.

Ref. R441, Part 246. See also: Dia|gram; Activity Region; Trace Skeleton; Trace Model.

Trace Skill

Trace patterns. The specialist competence required to interpret a trace's vocabulary, abstractions, timing, and failure semantics. Relevant evidence may be present yet remain unrecognized when the analyst lacks the appropriate domain knowledge. A viewpoint can be selected immediately; the ability to use it may require learning or collaboration across several technical disciplines.

Ref. R006. See also: Pattern repertoire; Trace Viewpoints; Trace World.

Traces of Individuality

Trace patterns. Naming, wording, and logging conventions characteristic of an individual developer or an organization. These features differ from conventions imposed by the technology itself. Recognizing them can help interpret unusual messages, but style alone is insufficient to establish authorship or identify a responsible component.

Ref. R002, p. 348. See also: Implementation Discourse; Message Invariant; Equivalent Messages.

tracestate

Tracing. The W3C Trace Context header that carries ordered vendor-specific tracing information alongside traceparent. It allows tracing implementations to retain information not represented by the common trace identifier and flags. It is not a general business-data container. Missing or changed entries can explain differences in tracing behavior, but their interpretation depends on the responsible vendor's defined semantics.

Ref. R264, tracestate header. See also: traceparent; Baggage; Context propagation.

Trace String

Trace patterns. A view of changing message or activity density along the ordered positions of a trace, and comparison of that variation across runs or regions. The string analogy concerns longitudinal structure rather than elapsed time. This distinguishes it from frequency changes measured against a time axis.

Ref. R002, pp. 341–342. See also: Density Distribution; Fourier Activity; Inter-Correlation.

Trace Summary

Trace patterns. A compact account of a trace containing selected timing, identity, and statistical information. Summaries from several captures can help locate relevant artifacts or serve as an index. A summary should preserve access to its source because aggregate descriptions may conceal the sequence needed for diagnosis.

Ref. R002, p. 343. See also: Indexical Trace; Trace Frames; Measurement.

Trace Temperature

Trace patterns. An externally assigned measure of incident importance attached to a trace or one of its regions. Values may be numerical or qualitative, such as hot or cold. A sparse or empty trace can therefore have high temperature. We distinguish this contextual priority from News Value, which derives its importance from the content of the trace itself.

Ref. R434, Part 239. See also: Trace Field; News Value; Sparse Trace.

Trace Tessellation

Trace patterns. An arrangement of trace segments along a line according to an explicit selection or mapping rule. Segment lengths, colors, and repetition can represent activity structures or adjoint thread ID (ATID) values. The resulting one-dimensional tessellation makes periodicity and interruptions visible. We distinguish the diagnostic arrangement from the decorative choices used to draw it, and retain the rule connecting segments to evidence.

Ref. R427, Part 232. See also: Trace Sketch; Activity Region; Adjoint Thread of Activity.

Trace Variance

Trace patterns. Variation among traces produced in different environments or at different times. The pattern borrows the model-variance analogy, treating the changing execution environment as input. Bifurcation Point, Trace String, and Polytrace provide patterns for investigating the variation. The term does not prescribe one statistical estimator; the analysis must specify which trace property varies and which conditions were compared.

Ref. R438, Part 243. See also: Trace Bias; Bifurcation Point; Trace String; Polytrace.

Trace Viewpoints

Trace patterns. The analytical perspectives from which a trace is read, including errors, use cases, architecture, implementation, performance, and foreground activity. Structural selections such as threads provide additional viewpoints. Naming the viewpoint makes its relevance and omissions visible and permits complementary readings of the same evidence.

Ref. R002, pp. 344–345. See also: Use Case Trail; Implementation Discourse; Background and Foreground Components.

Trace Volume

Trace patterns. The size of a trace representation when both recorded messages and inserted Silent Messages contribute to the count. The same idea applies to an Activity Region, an activity thread, or a braid. We state the counting unit and padding resolution before comparison. This volume describes the chosen representation and need not coincide with file size or the number of emitted records.

Ref. R433, Part 238. See also: Silent Messages; Activity Region; File Size.

Trace Window

Trace patterns. A selected portion of a trace along its message or time axis, its attribute dimensions, or both. The window defines which evidence is currently available to analysis. Its boundaries and retained fields matter when interpreting incomplete sequences, correlations, and measurements.

Ref. R002, p. 346. See also: Data Interval; Time Scale; Message Set.

Trace World

Trace patterns. A shared diagnostic model enriched by several traces, logs, and telemetry streams. Their local vocabularies and implementation discourses are aligned so that entity identity, relationships, and narrative continuity survive changes of viewpoint. The pattern requires meaningful integration; merely collecting several artifacts together does not establish that their entities or explanations agree.

Ref. R005. See also: Trace Ontology; Implementation Discourse; Trace Viewpoints.

Training-serving skew

AI and ML. A difference between the data or feature processing used to develop a model and what the deployed model receives. Missing fields, changed units, or inconsistent transformations can alter predictions without a change to model parameters. Diagnosis should compare the actual representations at both boundaries. Similar aggregate distributions do not guarantee that individual examples were processed consistently.

Ref. R238, Checking training-serving skew. See also: Data drift; Feature extraction; Data leakage.

Transaction isolation

Data systems. The rules governing which effects of concurrent transactions an operation may observe. Isolation determines whether apparent inconsistencies can occur within the database's declared contract. Product implementations can provide stronger guarantees than the minimum associated with a standard level, and stronger isolation can introduce retries or blocking. Record the actual level, statement boundaries, and transaction lifetime before diagnosing a concurrent read as incorrect.

Ref. R328, Isolation-level anomaly table; implementation differences. See also: Database blocking; Retry; Query plan.

Translated Exception

Memory patterns. Exception handling changes an exception into another representation, so the code or record seen later may differ from the original. We compare exception records, handler activity, and execution residue to reconstruct the translation. Several exception-related values need not represent independent faults. Establishing the transformation helps distinguish this pattern from an additional exception raised during nested exception processing.

Ref. R655, Translated Exception pattern and diagnostic examples. See also: Nested Exceptions (unmanaged code); Software Exception; Exception record.

Translated Message

Trace patterns. A readable description derived from a lower-level error, flag, or status code, often emitted near the original value. Reading the pair preserves precision that the description alone may lose. Translation may be localized, abbreviated, or implementation-specific and should not replace the original status convention.

Ref. R002, pp. 349–350. See also: Error Message; Message Context; Gossip.

Troubleshooting

Foundations. Problem solving directed at identifying a useful intervention for an observed software problem. It includes understanding the environment, narrowing the affected subsystem, investigating origin, and obtaining a remedy or workaround. Diagnostic analysis supplies evidence for these choices; some cases require source-level debugging, while others can be resolved through configuration, replacement, or operational changes.

Ref. R001, pp. 31–32. See also: Software diagnostics; Debugging; Workaround.

Troubleshooting Unit of Work

Debuggers. A self-contained troubleshooting check or sequence of steps, performed manually or implemented as a reusable function, class, or module. It performs a defined task and records a result for a coordinating workflow. We make its assumptions and output explicit so that the unit can be combined with other checks without losing diagnostic context.

Ref. R857, Pattern discussion. See also: Checklist; Supporting Module; Debugware.

Truncated Data

Trace patterns. A message value or state dump that contains only part of the available data. Sorting or changing presentation can alter which part is visible. We identify the truncation rule before treating missing elements as absent from the underlying state or comparing two displayed collections.

Ref. R002, p. 351. See also: State Dump; Corrupt Message; Visibility Limit.

Truncated Dump

Memory patterns. A dump whose capture or storage ended before the intended artifact was completely written. Missing ranges can prevent stack reconstruction, object inspection, or even normal loading of the file. We compare the artifact with its declared format and acquisition history before interpreting unreadable addresses. Deliberately omitted memory in a limited dump is a different condition from an unexpectedly missing portion of the file.

Ref. R357, Opening truncation discussion and debugger missing-data example. See also: Corrupt Dump; Minidump; No System Dumps.

Truncated Stack Trace

Memory patterns. A stack trace contains valid frames but omits part of the call path. We can compare corresponding threads or allocation records in other regions, process instances, or captures to investigate the missing portion. A matching fragment can suggest a fuller path, but does not recover the absent frames by itself. Preserve the distinction between observed frames and reconstruction.

Ref. R643, Truncated Stack Trace pattern and diagnostic examples. See also: Incorrect Stack Trace; Glued Stack Trace; Stack Trace (database).

Truncated Trace

Trace patterns. A trace whose captured beginning or ending omits activity needed to represent the investigated occurrence. Short duration and missing expected boundaries are clues, not sufficient tests alone. The relevant question is whether the artifact preserves the significant interval for the diagnostic task.

Ref. R002, p. 352. See also: Significant Interval; Trace Partition; Ruptured Trace.

U

Ubiquitous Component (kernel space)

Memory patterns. A kernel component recurs across many thread stacks or blocking locations, potentially across multiple processes. We examine its role, frame position, and shared dependencies to decide whether it connects the observed waits. Repetition can direct attention to a common blocking path, but ubiquity alone cannot separate expected infrastructure from a component relevant to the failure.

Ref. R502, Pattern description: Ubiquitous Component (kernel space). See also: Ubiquitous Component (user space); Blocking Module; Distributed Wait Chain.

Ubiquitous Component (user space)
Memory patterns. A component appears repeatedly across many user-mode thread stacks of a process. This wide presence can reveal shared instrumentation, integration, or interference relevant to the incident. We compare the observed distribution with the component's expected role and a suitable baseline. Frequency guides investigation, but widespread participation alone cannot distinguish a common dependency from the component responsible for abnormal behavior.

Ref. R747, Crash Dump Analysis Patterns (Part 85). See also: Pervasive System; Module Collection (predicate); Coupled Modules.

UI Message
Trace patterns. A trace message describing a user-interface action or state, such as a window's creation, a command, or a dialog's closure. Such messages connect reported visual behavior with internal activity. They can provide anchors for inspecting what the system did before and after the user's observable operation.

Ref. R002, p. 353. See also: Basic Facts; Significant Event; Use Case Trail.

Ultrasimilar Messages
Trace patterns. Messages differing by a very small amount, such as a single byte, within a common message family. These differences can be diagnostically significant even when coarse matching treats the messages as identical. The analysis examines the varying content and its sequence rather than assuming that near-identical text has the same meaning.

Ref. R002, p. 354. See also: Message Invariant; Data Flow; Equivalent Messages.

Undefined behavior
Execution. Program behavior on which the applicable language specification places no requirements. In C++, certain invalid constructs or data can enter this category without a guaranteed diagnostic or crash. We identify the violated rule before reasoning from source code to machine behavior. A result that appears stable across several runs does not give the program a defined meaning.

Ref. R406, [defns.undefined], definition and Note 1. See also: UndefinedBehaviorSanitizer; Memory corruption; Data race.

UndefinedBehaviorSanitizer
Program analysis. A collection of runtime checks for selected operations whose behavior is undefined or otherwise suspicious under the configured language checks. Examples include some arithmetic, alignment, and type-related violations. A report identifies an executed violation of an enabled check. A clean run covers neither every form of undefined behavior nor unexecuted code; the precise check set and recovery settings matter.

Ref. R180, Available checks and check groups. See also: AddressSanitizer; Dynamic analysis; Assertion.

Unified Debugging Pattern Language
Foundations. A proposed organization connecting analysis patterns with the architecture, design, implementation, and use of debugging facilities. A recognized problem can therefore lead to a corresponding way of observing execution and using the debugger. We keep these levels explicit: a breakpoint is an implementation facility, while the diagnostic purpose and the method of using it belong to other levels.

Ref. R878, Pattern levels and proposal. See also: Analysis pattern; Debugware; Breakpoint.

Unified Stack Trace

Memory patterns. A stack collection is combined into an aggregate representation that shares common frames or subtraces and records their multiplicities. This reveals common paths and components without repeating each complete stack. We preserve branching and the meaning of the counts when comparing collections. The representation generalizes deduplication: partially shared paths remain visible alongside completely repeated traces.

Ref. R682, Unified Stack Trace pattern and diagnostic examples. See also: Stack Trace Set; Constant Subtrace; Flame graph.

Uninitialized memory

Memory. Storage read before the program has established the value required by its logic. Residual bytes may come from earlier use of the same storage, but their apparent structure does not make them valid current data. We examine control paths and initialization requirements. Repeated values across runs can reflect allocator or environment behavior and do not prove that the program initialized them.

Ref. R051, Description; Extended Description. See also: Memory examination; Memory corruption; Null pointer dereference.

Unit test

Testing. A test of a selected unit of software behavior, commonly automated, with controlled setup and explicit expected results. The unit boundary is a design choice: a function, object, or small cooperating group may be appropriate. Isolation makes a failure easier to interpret, but substituted dependencies can omit significant interactions. A passing unit test therefore supports the behavior exercised within that chosen boundary.

Ref. R165, Test cases and assertion methods; R882, §2.2.1, p. 29; §1.4.1, p. 19. See also: Test fixture; Mock; Regression test.

Unknown Component

Memory patterns. A module's name and ordinary debugger metadata do not identify its product or vendor sufficiently for the investigation. We inspect the captured image's resource information and embedded strings for additional provenance. Such recovered descriptions are evidence to corroborate against the loaded image and deployment context; an unfamiliar name alone does not establish that the component is defective.

Ref. R472, Pattern description: Unknown Component. See also: Module Variety; Same Vendor; Module Hint.

Unloaded Module

Memory patterns. A failing execution address or callback points into the former address range of an unloaded module. We compare the exception context, indirect call target, and unloading information to investigate a stale code reference. An unloaded-module list supplies a lifetime clue; the relationship to the actual target must be established before assigning the failure to that module.

Ref. R659, Unloaded Module pattern and diagnostic examples. See also: Use-after-free; Function Pointer; Module list.

Unrecognizable Symbolic Information

Memory patterns. Symbol files exist, but incomplete or unrecognized module information prevents the debugger from locating them automatically. Another process context containing the same component may reveal the required symbol path. We recover and validate the appropriate symbols before interpreting frames, distinguishing a failure of symbol discovery from the absence of debugging information altogether.

Ref. R531, Pattern description: Unrecognizable Symbolic Information. See also: No Component Symbols; Incorrect Stack Trace; Stack Trace Collection.

Unresponsive Window

Execution. A named User Interface Problem Analysis pattern in which a window does not respond to expected interaction. Its thread may be waiting, blocked, or occupied with lengthy computation. We inspect its stack, processor use, and dependencies before naming a mechanism. The freeze can be temporary; an unresponsive interface does not by itself establish deadlock.

Ref. R825, Pattern discussion. See also: Hang; Main Thread; Wait Chain; Spiking Thread.

Unsynchronized Dumps

Memory patterns. Related memory dumps were captured at different times, so their states cannot automatically be combined into one simultaneous picture. We compare capture timestamps, process or thread ages, and the expected communication sequence before joining evidence. An apparent contradiction may come from capture order. The time difference remains part of the analysis when reconstructing dependencies across artifacts.

Ref. R646, Unsynchronized Dumps pattern and diagnostic examples. See also: Memory dump; Thread Age; Coupled Processes.

Unsynchronized Messages

Trace patterns. Messages whose relative order cannot be determined from their displayed timestamps, even when their sources use synchronized clocks. Equal timestamps at finite resolution are a common example. Additional sequence or semantic evidence may constrain order; an arbitrary tie-breaking rule only changes presentation.

Ref. R002, pp. 355–356. See also: Unsynchronized Traces; Event Sequence Order; Trace Mask.

Unsynchronized Traces

Trace patterns. Traces whose time bases, capture intervals, or recorded occurrences do not establish the alignment required for correlation. Clock offset and time-zone representation are possible issues, but captures may also describe different intervals. Alignment requires corroborating events or metadata; matching clock fragments alone is insufficient.

Ref. R002, pp. 357–358. See also: Calibrating Trace; Shared Point; Inter-Correlation.

Uprobe

Native tools. A Linux mechanism for instrumenting supported locations in user-space executable code, commonly identified through an object file and offset. Return probes can observe function completion. We match the probe to the exact binary and interpret arguments using its calling convention. A symbolic function name alone may not identify all generated code paths, especially with inlining, optimization, or multiple loaded versions.

Ref. R086, Overview; Synopsis of uprobe_tracer. See also: Kprobe; Build ID; Calling convention.

Use-after-free

Memory. Access to an allocation after its lifetime has ended and its storage has been released. The old address can still be readable or can already belong to a new allocation. We reconstruct allocation and release history instead of treating current page accessibility as proof of validity. The same defect may produce a crash, silent corruption, or apparently correct behavior under different reuse patterns.

Ref. R046, Description. See also: Dangling pointer; Double free; Memory corruption.

Use-case Deviation
Foundations. An elementary diagnostic pattern in which observed behavior departs from the expected use case. We describe the intended sequence, the observed departure, and the conditions under which it occurs before looking for a detailed mechanism. This description can guide artifact acquisition without assuming that the deviation will present as a crash, hang, or explicit error message.

Ref. R001, p. 170. See also: Elementary diagnostic pattern; Problem description; Reproducer.

Use Case Trail
Trace patterns. Messages corresponding to the execution of a use-case instance across its implementing components. Several trails may be interleaved in one artifact. Instance identifiers or other correlation evidence separate them, while sequence-phase analysis can provide provisional grouping when explicit tags are absent.

Ref. R002, p. 359. See also: Event Sequence Phase; Milestones; Motif.

USE method Utilization, Saturation, Errors
Performance. A resource-oriented performance investigation method: enumerate relevant resources, then examine utilization, saturation, and errors for each. Beginning with resources and questions helps reveal missing measurements as well as visible bottlenecks. The method organizes an initial search; it does not establish cause from one high counter. Software limits and virtual resources may need the same examination as physical processors, disks, and networks.

Ref. R301, The USE Method; terminology. See also: Saturation; RED method; Resource limit.

User mode
Execution. An execution mode with restricted access to privileged processor and operating-system resources. Ordinary application code typically runs here and requests kernel services through defined transitions. We distinguish the application's accessible address space from kernel-owned state during analysis. A user-mode failure can result from an application defect, a dependency, or a system interaction; the mode identifies an execution boundary, not the ultimate cause.

Ref. R120, User mode; Kernel mode. See also: Kernel mode; System call [Execution]; Address space.

User space
Memory. The part of an address-space model made available to an application process, subject to its mappings and access restrictions. The earlier Dictionary of Debugging blog entry illustrates it as a lower interval of a divided memory space. We distinguish that diagram from the actual layout being analyzed and from user mode, which describes execution privilege. Some user-space regions may remain unavailable in a dump.

Ref. R420, Definition and examples. See also: Kernel space; User mode; Virtual memory; Memory dump.

User Space Evidence
Memory patterns. A kernel memory dump contains indirect evidence about user-space behavior despite lacking the full user-space memory needed for direct inspection. Wait relationships, process and thread state, handles, and kernel stacks can identify promising processes or dependencies. We use those indicators to guide analysis and further capture. The inferred user-space story remains constrained by the artifact's actual coverage.

Ref. R675, User Space Evidence pattern and diagnostic examples. See also: Kernel memory dump; Wait Chain; Complete memory dump.

V

Valgrind
Program analysis. A framework that runs programs under dynamic binary instrumentation and supplies analysis tools such as Memcheck, Helgrind, and Massif. It can inspect selected behavior without requiring the target to be rebuilt with a compiler sanitizer. Its execution environment introduces substantial changes in timing and other behavior. Conclusions must therefore identify the selected tool, options, workload, and instrumented execution context.

Ref. R184, Introduction; Valgrind core and tools. See also: Memcheck; Helgrind; Massif.

Value Adding Process
Memory patterns. An additional process supplies optional experience or integration features, often appearing in many user sessions. When its purpose overlaps the reported malfunction, we examine its interactions and consider controlled isolation as a diagnostic comparison. The pattern identifies a potentially relevant environmental participant; its optional role or repeated presence does not establish that it is defective.

Ref. R780, Crash Dump Analysis Patterns (Part 173). See also: Pervasive System; Coupled Processes; Changed Environment.

Value Deviation (stack trace)
Memory patterns. A stack's depth differs markedly from the relevant collection or expected execution role. We first verify that the trace is correctly reconstructed, then relate the deviation to processor use, code, and raw-stack evidence. An unusually short trace can be as informative as a long one, but the appropriate comparison depends on the artifact and workload.

Ref. R505, Pattern description: Value Deviation (stack trace). See also: Stack Trace Collection; Incorrect Stack Trace; Execution Residue.

Value Deviation (structure field)
Memory patterns. A structure field contains a value unusual for the expected state, exposing a transition or condition not obvious from ordinary stack listings. A process-deletion indicator, for example, can suggest incomplete termination while threads remain. We interpret the field using the correct structure and lifecycle semantics, then seek corroborating events or artifacts rather than treating one uncommon value as a complete explanation.

Ref. R629, Value Deviation (structure field) pattern and diagnostic examples. See also: Structure Field Collection; Diagnostic indicator; Missing Process.

Value References

Memory patterns. Memory locations that contain a selected value, pointer, or handle. We search the relevant virtual or physical memory range and then interpret each match in its structural context. Restricting a search to modules can help locate code references, while object-tracing facilities may supply lifetime evidence; equal bit patterns alone do not establish a semantic relationship.

Ref. R529, Pattern description: Value References. See also: Wild Pointer; Data Correlation (CPU times); Execution Residue.

Variable Subtrace

Memory patterns. The portion of one thread's stack that varies across a sequence of dumps while a surrounding portion remains stable. Changes in return addresses, frame positions, and arguments help identify the call region containing repeated activity. We use the boundary to investigate a suspected loop, with confidence limited by snapshot count, timing, and the validity of reconstructed frames.

Ref. R551, Pattern description: Variable Subtrace. See also: Spiking Thread; Stack Trace Change; False Function Parameters.

Vector clock

Distributed systems. A logical timestamp containing components associated with participating processes or actors. Under its defined update and comparison rules, it represents causal history and can distinguish ordered events from concurrent ones. This is stronger information than a scalar ordering alone. Its interpretation depends on membership and propagation assumptions; incomplete vectors, resets, or missing communication records can invalidate a reconstructed causal relationship.

Ref. R318, Vector time; causal relationships. See also: Logical clock; Causal order; Partial failure.

Version-Specific Extension

Memory patterns. A debugger extension or supporting runtime component must match the target's version closely enough to interpret its internal data. We compare the captured module version with the available analysis binaries and follow the loader's compatibility diagnostics. The pattern explains why a general debugger session can work while managed inspection fails, and why a matching support library may restore access.

Ref. R766, Crash Dump Analysis Patterns (Part 136). See also: Platform-Specific Debugger; Managed Code Exception; Not My Version.

Virtual address

Memory. An address presented by executing software and interpreted through the memory-management configuration. The same numerical address can refer to different physical storage in different processes, or to different objects at different times. During dump analysis, we first establish the process context and mapping. An apparently plausible address is not sufficient evidence that the intended object still exists.

Ref. R011, Virtual address space and address translation. See also: Physical address; Address space; Use-after-free.

Virtualized Process (ARM64EC and CHPE)

Memory patterns. A Windows ARM64 process contains execution associated with x64 or x86 compatibility mechanisms, visible through ARM64EC (Emulation Compatible) or CHPE (Compiled Hybrid Portable Executable) frames and debugger contexts. We identify the architecture of each relevant frame before reading registers, instructions, or parameters. A mixed stack can be legitimate evidence of the execution environment, rather than an indication of corrupt unwinding.

Ref. R475, Pattern description: Virtualized Process (ARM64EC and CHPE). See also: Virtualized Process (WOW64); ISA-Specific Code; ABI.

Virtualized Process (WOW64)

Memory patterns. A 32-bit Windows application is captured within the 64-bit WOW64 execution environment. The initial native view may expose transition machinery while obscuring the application frames needed for diagnosis. We select the appropriate execution context and stack view, and verify debugger support for the target. WOW64 scaffolding or an unfamiliar mixed stack is not, by itself, corruption.

Ref. R720, Crash Dump Analysis Patterns (Part 26). See also: Platform-Specific Debugger; Module Stack Trace; Incorrect Stack Trace.

Virtualized System

Memory patterns. The captured operating system executes within a virtualization environment whose drivers, hypercalls, and execution transitions affect the dump's interpretation. We identify that environment and consider its contribution to unusual stack shapes or behavior. Virtualization evidence supplies context and additional components to examine; it does not establish that the hypervisor caused an observed guest crash or hang.

Ref. R748, Crash Dump Analysis Patterns (Part 87). See also: Virtualized Process (WOW64); Inconsistent Dump; Dump Context.

Virtual memory

Memory. The memory view presented to a computational process, with contents supplied through mappings to host memory and storage. In the earlier Dictionary of Debugging blog entry, this view is represented as an ordered address range that can contain inaccessible regions. We interpret a virtual address in its process and mapping context; the view need not correspond to one contiguous physical-memory region.

Ref. R417, Definition and examples. See also: Virtual address; Physical memory; Address space; Memory map.

Visibility Limit

Trace patterns. An intrinsic boundary of what a tracing mechanism can observe, such as activity before that mechanism starts. It differs from an omitted component or insufficient statements in otherwise observable code. A complementary observer may extend the available evidence, but the original trace retains its own limit.

Ref. R002, p. 360. See also: Sparse Trace; Missing Component; Inter-Correlation.

Visitor Trace

Trace patterns. Repeated message structures that successively describe different objects or data items, analogous to visiting each object in a collection. The changing identifying data distinguishes the visits. Recognizing the pattern helps separate traversal structure from the status or anomalies reported for individual objects.

Ref. R002, p. 361. See also: Periodic Message Block; Data Flow; Punctuated Activity.

vmcore

Artifacts. A conventional name for a Linux kernel crash-memory artifact, including the ELF view exposed as /proc/vmcore to a kdump capture environment. It preserves material for reconstructing the failed kernel's state. We associate it with the matching kernel image, debug information, and collection settings. The name alone does not identify whether the saved file is complete, filtered, compressed, or readable by a particular analysis tool.

Ref. R035, Overview. See also: kdump; crash utility; ELF.

VMMap

Native tools. A Sysinternals tool that presents a Windows process's virtual and physical memory usage by allocation category and region. It helps distinguish address-space reservation, committed memory, and resident pages. We compare equivalent captures when investigating growth. A large category identifies where memory is accounted for, but does not alone establish allocation ownership, effective lifetime, or whether the application still needs the memory.

Ref. R073, Introduction. See also: Memory map; Committed memory; Working set [Memory].

Vocabulary Index

Trace patterns. Domain vocabulary used to locate messages relevant to a problem description. Searching for terms associated with the reported function can reduce a very large trace to an initial region. The vocabulary is a navigation aid; synonyms, implementation wording, and missing instrumentation can leave relevant activity outside the initial matches.

Ref. R002, p. 362. See also: Basic Facts; Implementation Discourse; Data Interval.

W

Wait Chain

Memory patterns. A sequence of dependencies in which one thread's progress requires an action or resource associated with another participant. Following owners, replies, notifications, or termination dependencies can move the investigation toward the obstructing activity. We distinguish the observed wait from the inferred responsibility for satisfying it. A cycle suggests deadlock only when the relevant dependencies cannot be broken by another permitted action.

Ref. R376, Opening dependency-chain definition and thread relationship examples. See also: Blocked Thread; Deadlock; Coupled Processes.

Wait Chain (C++11, condition variable)

Memory patterns. A thread waits through a C++ condition-variable implementation, but the party responsible for progress may not be directly identifiable from the wait object. We compare shared subtraces and references to synchronization values in other threads' raw stacks. Such matches provide candidate relationships that require corroboration; a condition variable does not inherently identify an owning thread.

Ref. R486, Pattern description: Wait Chain (C++11, condition variable). See also: Constant Subtrace; Place Trace; Execution Residue.

Wait Chain (CLR monitors)

Memory patterns. A managed thread is waiting through the CLR monitor mechanism. Native frames such as AwareLock::Enter and CLREvent::Wait identify this wait in the illustrated stacks. We need managed-runtime evidence to establish ownership and follow the dependency. The source suggests requesting a process memory dump for analysis with SOS or another suitable extension.

Ref. R730, Crash Dump Analysis Patterns (Part 42l). See also: Wait Chain; Managed Stack Trace; Managed Code Exception.

Wait Chain (critical sections)

Memory patterns. A dependency chain formed by threads waiting for critical sections held by other threads. Automatic hang analysis may expose only one chain and miss a wider or longer one. We inspect all relevant held critical sections and their owners, then follow each owner's next wait. The final dependency may leave critical-section synchronization entirely, for example through a socket operation.

Ref. R605, Wait Chain (critical sections) pattern and diagnostic examples. See also: Wait Chain; Critical section; High Contention.

Wait Chain (executive resources)

Memory patterns. A thread waits for an executive resource held by another thread, which may itself depend on another resource or operation. We follow owners and waiters through the chain until the endpoint explains the lost progress. Multiple linked waits establish dependency; an unresponsive endpoint does not by itself establish a deadlock.

Ref. R727, Crash Dump Analysis Patterns (Part 42c). See also: Wait Chain; Deadlock (self); Blocked Thread.

Wait Chain (LPC/ALPC)

Memory patterns. A wait chain that crosses LPC or ALPC request-and-reply relationships. A client waiting for a message reply can lead to the server thread processing that message, whose own synchronization or communication wait extends the chain. We retain message and thread identities while traversing it. Correlating the transport relationship is necessary before attributing the delay to either process.

Ref. R606, Wait Chain (LPC/ALPC) pattern and diagnostic examples. See also: Wait Chain; Blocked Thread; Process.

Wait Chain (modules)

Memory patterns. A provisional dependency view in which a calling module waits for a called module to return. Modules act as pseudo-synchronization objects, allowing callback inversions or repeated intermodule dependencies to suggest a deeper investigation. We then examine actual synchronization objects or source code. The module-level diagram organizes suspicion but does not by itself establish an operating-system deadlock.

Ref. R609, Wait Chain (modules) pattern and diagnostic examples. See also: Wait Chain; Module Collection; Deadlock.

Wait Chain (mutex objects)

Memory patterns. A blocked thread waits for a mutex or Windows mutant owned by another thread. We use the recorded owner to connect thread and process contexts, then examine the owner's next dependency. Several mutex links can join otherwise separate applications into one chain; their presence alone does not establish deadlock unless the dependency cycle closes.

Ref. R484, Pattern description: Wait Chain (mutex objects). See also: Wait Chain; Deadlock; Coupled Processes.

Wait Chain (named pipes)

Memory patterns. A thread's pending operation depends on a named-pipe endpoint. We recover the pipe identity from its I/O or file context and follow the server-side processing and any subsequent waits. The chain can cross process or machine boundaries, so a local waiting stack is only one segment of the dependency that must complete before the operation returns.

Ref. R729, Crash Dump Analysis Patterns (Part 42i). See also: Wait Chain; Hidden Parameter; Coupled Processes.

Wait Chain (nonstandard synchronization)

Memory patterns. A wait dependency is implemented through custom synchronization that is absent from standard lock listings. We compare recurring stack fragments, inspect the custom routines and object fields, and connect lower-level waits to the intended lock behavior. The resulting ownership or signaling relationship must be established from the implementation; a familiar-looking function name is only a starting clue.

Ref. R731, Crash Dump Analysis Patterns (Part 238). See also: Wait Chain; Constant Subtrace; Hidden Parameter.

Wait Chain (process objects)

Memory patterns. A dependency chain includes a wait for another process object to become signaled on process termination. We identify that process and inspect the activity that keeps it alive, continuing across any further waits. This can connect an apparently local blocked call to a different process or driver; the process handle alone does not explain why termination has not completed.

Ref. R728, Crash Dump Analysis Patterns (Part 42f). See also: Wait Chain; Coupled Processes; Zombie Processes.

Wait Chain (pushlocks)

Memory patterns. A wait-chain specialization in which kernel threads are blocked acquiring shared or exclusive pushlocks. The wait reason and acquisition frames identify the synchronization family, while stack and thread context guide the search for dependencies. We preserve the distinction between recognizing a pushlock wait and establishing its owner or a cycle; a waiting stack alone supplies only part of the chain.

Ref. R608, Wait Chain (pushlocks) pattern and diagnostic examples. See also: Wait Chain; Blocked Thread; Deadlock.

Wait Chain (RPC)

Memory patterns. A waiting RPC client depends on a server-side execution that may itself wait on another resource or remote call. We identify transport-specific client and server stacks, follow the endpoint relationship, and continue through any local synchronization dependency. Combining these links explains distributed waiting that cannot be understood from one blocked stack alone.

Ref. R483, Pattern description: Wait Chain (RPC). See also: Wait Chain (LPC/ALPC); Distributed Wait Chain; Coupled Processes.

Wait Chain (RTL_RESOURCE)

Memory patterns. A wait passes through the Windows RTL_RESOURCE implementation and its internal synchronization objects. We reconstruct the resource address from the calling context, inspect its ownership information, and follow the owning thread. Because layouts and calling code vary, the analysis must verify the captured implementation rather than assume that a structure definition from another platform or version matches.

Ref. R485, Pattern description: Wait Chain (RTL_RESOURCE). See also: Wait Chain; Wait Chain (critical sections); Not My Version.

Wait Chain (SRW lock)

Memory patterns. A chain involving slim reader/writer lock acquisition. Where ownership is not directly exposed by the available debugger view, similarities in thread-stack semantics can suggest a candidate owner or related operation. We validate that relationship using the lock's address, execution context, and surrounding evidence. Matching names or activities are investigative clues rather than conclusive ownership information.

Ref. R610, Wait Chain (SRW lock) pattern and diagnostic examples. See also: Wait Chain; Reader-writer lock; Stack Trace Collection.

Wait Chain (thread objects)

Memory patterns. A thread waits for another thread object to reach its signaled termination state. We identify the target thread, inspect its stack and dependencies, and continue until the chain reaches an explanatory activity or unresolved endpoint. The same process may contain several such links; a thread-object wait identifies a dependency, without establishing a circular deadlock.

Ref. R482, Pattern description: Wait Chain (thread objects). See also: Wait Chain; Wait Chain (process objects); Thread.

Wait Chain (window messaging)

Memory patterns. A chain of blocked synchronous window-message calls. Sending threads wait for the corresponding window procedures, and the destination threads can make further calls or wait elsewhere. We correlate window handles, procedure identities, and stacks across involved processes, including kernel-visible waits. The visible messaging function identifies the dependency mechanism; the blocking operation may lie farther along the chain.

Ref. R607, Wait Chain (window messaging) pattern and diagnostic examples. See also: Wait Chain; Main Thread; Stack Trace Collection.

Wait event

Data systems. A database-reported description of what a session is waiting for at an observation point. PostgreSQL separates the wait-event type from the particular event and exposes them with session activity. Waiting is not automatically pathological: idle clients and expected coordination also wait. To interpret a delay, relate repeated observations to the active statement, blocker, resource, and total time experienced by the caller.

Ref. R299, pg_stat_activity; wait_event and wait_event_type. See also: Database blocking; Query plan; Latency.

Waiting Thread Time

Memory patterns. The duration of a thread's current wait, reconstructed from available scheduler information and the capture time. Comparing this interval with the reported symptom helps us distinguish a long-standing dependency from a recently entered wait. We account for the operating system's representation and time units. A short current wait does not prove overall progress, because the thread may repeatedly return to the same dependency.

Ref. R358, Wait-start tick examples and elapsed-wait interpretation. See also: Blocked Thread; Accidental Lock; Wait Chain.

Waiting Thread Time (user dumps)

Memory patterns. A process dump can provide thread CPU times and elapsed lifetime from which we estimate how much time the thread was not executing. We use this together with its current wait and timeout arguments. The difference is accumulated non-running time, not necessarily one continuous wait, so it cannot by itself date the start of the currently observed blockage.

Ref. R715, Crash Dump Analysis Patterns (Part 19b). See also: Waiting Thread Time; Blocked Thread; Spike Interval.

Wall-clock time

Performance. A term used for civil-clock readings and, in profiling, for elapsed duration including waiting rather than accumulated CPU execution. These uses need an explicit convention. A settable clock can jump when corrected, distorting differences between timestamps. Use an appropriate monotonic clock for interval measurement and preserve civil timestamps when the investigation needs alignment with externally recorded events.

Ref. R398, CLOCK_REALTIME; R403, Wall-clock time versus CPU utilization. See also: Elapsed time; Monotonic clock; CPU time.

Watchpoint

Debuggers. A debugger request to stop when a selected expression changes or when supported memory access conditions occur. Hardware watchpoints use target facilities; software watchpoints may repeatedly evaluate an expression during execution. We choose an address and lifetime consistent with the suspected object. Watching reused storage can expose a later occupant, and a value-change watchpoint may not identify every read or write to the underlying bytes.

Ref. R020, Section 5.1.2. See also: Hardware breakpoint; Use-after-free; Memory examination.

Watch Thread

Trace patterns. An independently added sequence of observations of selected memory locations or state, collected at events or intervals alongside a main trace. It augments existing logging without requiring those observations in the original source statements. The observation schedule and any effects on execution remain part of its interpretation.

Ref. R002, pp. 363–364. See also: State Dump; Counter Value; Adjoint Space.

Weak reference

Runtimes. A reference that permits an object to be collected when the relevant strong reachability is absent. It allows observation or caching without independently guaranteeing the referent's lifetime. The referent may disappear between checks unless the program obtains an appropriate strong reference for use. Weak-reference processing and finalization interactions depend on the runtime and should be interpreted using that implementation's rules.

Ref. R193, Weak-reference behavior and use. See also: Garbage collection; GC root; Finalization.

Weave of Activity

Trace patterns. A combined view connecting an activity's messages with associated traces, such as call stacks, and with matching adjoint activities found within those supplementary traces. The resulting weave exposes relationships across two structural levels. Each connection retains the identity and capture context of the evidence it links.

Ref. R002, p. 366. See also: Fiber Bundle; Adjoint Thread of Activity; Trace Quilt.

Well-Tested Function

Memory patterns. A familiar, widely exercised function provides a provisional boundary for investigating its caller and parameters. A failure inside a string-copy routine, for example, can direct attention to the supplied buffers. We use the function's established role to prioritize inspection, while retaining the possibility of implementation or environment defects. Familiarity is an investigative heuristic rather than a proof of correctness.

Ref. R627, Well-Tested Function pattern and diagnostic examples. See also: Parameter Flow; Stack trace [Artifacts]; Diagnostic hypothesis.

Well-Tested Module

Memory patterns. A familiar module with substantial testing history is provisionally given lower investigative priority while we examine less understood participants in the failing path. This is an analysis heuristic, not a guarantee of correctness. We retain the possibility of version-specific defects and unusual interactions, especially when direct evidence points back to the module despite its established reputation.

Ref. R759, Crash Dump Analysis Patterns (Part 119). See also: Top Module; Exception Module; Not My Version.

Whisker Trace

Trace patterns. A derived composition that combines compatible adjoint activities and may introduce message copies to align their structures. The name uses horizontal composition and whiskering as categorical analogies. Copied messages belong to the derived representation and must remain distinguishable from independently observed events in the original traces.

Ref. R002, p. 365. See also: Serial Trace; Trace Mask; Adjoint Thread of Activity.

White-box monitoring

Reliability. Monitoring information exposed by a system's implementation, such as queues, resource usage, counters, and internal errors. It can explain contributing conditions that external probes cannot see. Normal component metrics do not prove correct user outcomes. We combine this internal view with measurements at relevant service boundaries.

Ref. R285, Definitions. See also: Black-box monitoring; Instrumentation; USE method.

Wild Code

Memory patterns. Execution has followed a corrupted function pointer or return address into memory that still contains decodable instructions. The resulting path may modify data, loop, or eventually fault rather than fail immediately at the transfer. We examine the branch origin, destination bytes, and memory protections. Valid instruction decoding alone does not establish that the bytes belong to an intended code path.

Ref. R612, Wild Code pattern and diagnostic examples. See also: Wild Pointer; Local Buffer Overflow; Disassembly.

Wild Explanation

Foundations. Replacing a diagnostic account with loose associations, unsupported analogies, or an unverified reading of technical fields. A convincing narrative can then outrun the artifact. We decode the relevant structures using their actual definitions and test the proposed relationships. Mathematical or physical imagery may organize a model, but it does not supply evidence for the mechanism being claimed.

Ref. R876, Antipattern discussion. See also: Defect mechanism; Diagnostic evidence; Causal model.

Wild Pointer

Memory patterns. A supposed pointer contains a value whose structure suggests unrelated data, such as encoded text, or otherwise points outside the intended object or code. We interpret its bits and the operation that used it to investigate possible overwrite or propagation. Some wild values happen to address readable memory, so accessibility alone does not establish pointer validity.

Ref. R488, Pattern description: Wild Pointer. See also: Invalid Pointer (general); Wild Code; Local Buffer Overflow.

WinDbg

Debuggers. Microsoft's debugger for supported Windows user-mode and kernel-mode targets, including live execution and saved dumps. Its commands, extensions, and symbol facilities expose different views of the same captured system. We choose the target architecture, process, thread, and register context before interpreting output. Availability of a command does not establish that the current dump includes the memory needed to execute it meaningfully.

Ref. R080, Debugging environments. See also: Debugger context; !analyze; Symbol path.

Window Hint

Memory patterns. Window class names or related strings retained in a process's memory suggest the presence of other applications in its environment. We examine the containing region and corroborate the names before using them as troubleshooting leads. Such strings can survive as execution residue and may be stale or fabricated, so they do not independently prove that the corresponding window was active.

Ref. R556, Pattern description: Window Hint. See also: Execution Residue; Special Process; Changed Environment.

Word of Mouth

Foundations. Acquiring diagnostic-tool knowledge only through commands or techniques encountered informally in other people's work. The resulting repertoire depends on accidental exposure and can leave useful capabilities undiscovered. We use references, examples, and deliberate exploration to connect tool operations with analysis questions, while treating a borrowed command as something whose assumptions and output still require understanding.

Ref. R865, Antipattern discussion. See also: Pattern repertoire; Elementary analysis pattern; Analysis pattern.

Workaround

Foundations. A change that avoids or reduces the effects of a problem while the underlying defect may remain. It can alter conditions, replace a component, disable a feature, or change an operational procedure. We record the conditions under which it helps, because a successful workaround supplies causal clues but does not by itself prove a complete mechanism.

Ref. R001, pp. 31–32; p. 82. See also: Troubleshooting; Workaround pattern; Defect mechanism.

Workaround pattern

Foundations. A reusable form of workaround for a recurrent problem in a particular context. The pattern records the conditions, intervention, and resulting behavior so that another case can be assessed rather than handled by blind repetition. Its applicability should be checked against diagnostic indicators and operational consequences in the current environment.

Ref. R001, p. 82. See also: Workaround; Problem pattern; Troubleshooting.

Working Set
Trace patterns. The subset of trace messages currently used in an investigation. It changes as selection, navigation, and interpretation proceed. Naming the working set makes the current evidence boundary explicit and helps record how a conclusion was reached; this sense differs from resident memory pages in operating-system analysis.

Ref. R002, pp. 367–368. See also: Message Set; CoTrace; Trace Path.

Working set
Memory. On Windows, the pageable virtual-memory pages of a process that are currently resident in physical memory. This is a residency measure, influenced by access history and operating-system trimming. We compare it with committed memory and workload before interpreting growth. A smaller working set can follow reclamation without any application deallocation, while shared resident pages can appear in several processes.

Ref. R013, Working set and hard and soft page faults. See also: Committed memory; Memory pressure; VMMap.

WPA Windows Performance Analyzer
Native tools. A Windows Performance Toolkit application for exploring recorded performance data through tables, graphs, and related views. It commonly supports analysis of ETW traces collected with WPR. We choose a time interval and correlate relevant execution and resource views. A visually prominent interval or stack becomes diagnostic evidence only when its measurement semantics and relationship to the reported symptom are understood.

Ref. R105, Introduction. See also: ETW; WPR; Stack trace [Artifacts].

WPR Windows Performance Recorder
Native tools. A Windows Performance Toolkit recorder that collects event-trace data using configured recording profiles and options. Its choices determine which parts of system behavior can later be analyzed. We match providers, stacks, duration, and recording mode to the diagnostic question before collection. A trace file may open successfully in an analyzer while lacking the particular events needed to explain the incident.

Ref. R106, Introduction. See also: ETW; WPA; Stack trace [Artifacts].

Wrong Dump
Foundations. Supplying a dump whose process, machine, capture type, or scope does not match the diagnostic request. A valid artifact can therefore be unsuitable for the intended question. We verify its identity and contents before analysis or handoff and explain any substitution. The filename or accompanying description alone is insufficient to establish which state the artifact contains.

Ref. R866, Antipattern discussion. See also: Missing Space; Memory dump; Diagnostic artifact.

Y

Young System
Memory patterns. The system or process lifetime is too short for the reported scenario to have developed under the expected conditions. We compare uptime with initialization progress, sessions, workload, and the problem description. A capture taken after a restart may describe a fresh environment rather than the original failure, although genuine initialization defects can occur in a young system.

Ref. R497, Pattern description: Young System. See also: Overaged System; Early Crash Dump; Dump Context.

Z

Zero Fault diagnostics

Foundations. A proposed preventive approach combining general structure and behavior catalogs, product-specific problem catalogs, live monitoring, and premortem artifact analysis. The aim is to discover concerning indicators before a reported failure. The name expresses a diagnostic objective; applying the approach does not establish that software is defect-free or that all future faults are predictable.

Ref. R001, pp. 171–172. See also: Premortem diagnostics; Software prognostics; Diagnostics-driven development.

Zippocricy

Foundations. Forwarding a diagnostic archive without opening it and checking the artifacts it contains. Corrupt files, irrelevant captures, or missing evidence may then be discovered only after a further handoff. We inspect identity, scope, and readability before relying on the package. Receiving a file with the expected extension does not establish that the requested evidence was captured.

Ref. R864, Antipattern discussion. See also: Diagnostic artifact; Wrong Dump; Corrupt Dump.

Zombie process

Execution. A terminated Unix-like process whose parent has not yet collected its termination status through the appropriate wait mechanism. Its remaining process-table information supports that collection; it is not an ordinary running process consuming a full live address space. We investigate the parent's child-reaping behavior when zombies accumulate, rather than looking for a thread inside the terminated child to resume.

Ref. R060, DESCRIPTION; NOTES on zombie processes. See also: Process; Signal [Execution]; File descriptor.

Zombie Processes

Memory patterns. In this Windows analysis pattern, terminated processes leave process objects behind because references, often open handles, keep them alive. They may have no active threads or ordinary private memory yet still consume related resources. We examine ownership and lifecycle evidence to find the retaining process. The term has a different operating-system meaning from the familiar Unix unreaped-child state.

Ref. R733, Crash Dump Analysis Patterns (Part 54). See also: Process Factory; Memory Leak (page tables); Crashed Process.

Ambiguity guide

The domain tag and surrounding evidence determine the intended meaning. The entries below separate identical or similar wording that can otherwise lead an investigation in the wrong direction.

Signal

Signal [Execution] is an operating-system notification. Signal [Telemetry] identifies a form of collected information. Signal [Trace patterns] identifies an analysis pattern in trace evidence.

Entries: Signal [Trace patterns]; Signal [Telemetry]; Signal [Execution].

Timeout

The distributed-systems entry defines a waiting boundary and its expiry. The trace-pattern entry addresses timeout evidence in an artifact. Expiry does not establish the final state of remote work.

Entries: Timeout [Trace patterns]; Timeout [Distributed systems]; Deadline.

Working Set / working set

Working Set [Trace patterns] is the subset of trace data selected for an analysis task. Working set [Memory] is the Windows measure of resident pageable process memory.

Entries: Working Set [Trace patterns]; Working set [Memory].

PDB / pdb

PDB is a Program Database containing debug information. Python’s pdb is an interactive debugger. Capitalization is significant.

Entries: PDB [Artifacts]; pdb [Runtimes].

Fault / error / failure

These terms identify different levels of a causal account. State the system boundary, and distinguish an Error Message from the state or outcome it reports.

Entries: Fault; Error; Failure; Error Message.

Pattern / mechanism

A problem pattern organizes recurrent indicators. An analysis pattern describes how to investigate. A defect mechanism explains how a problem can produce those indicators. Recognition and causal explanation are separate steps.

Entries: Problem pattern; Analysis pattern; Defect mechanism; APM.

Pattern-based / pattern-driven

Pattern-based diagnostics develops and maintains the catalog. Pattern-driven diagnostics applies it to a case. Pattern-oriented diagnostics includes both, together with systemic analysis.

Entries: Pattern-based diagnostics; Pattern-driven diagnostics; Pattern-oriented diagnostics.

Systematic / systemic

Systematic diagnostics organizes coherent diagnostic knowledge. Systemic diagnostics applies general patterns across artifacts and platforms and follows relevant relationships across boundaries.

Entries: Systematic diagnostics; Systemic diagnostics.

Thread / thread of activity

An operating-system thread is an execution entity. A Thread of Activity is a sequence identified in trace evidence; an Adjoint Thread of Activity can organize related messages across ordinary thread boundaries.

Entries: Thread; Thread of Activity; Adjoint Thread of Activity.

Counter / Counter Value

A metric Counter accumulates nonnegative increments and may reset. Counter Value is a trace-analysis pattern for interpreted measured values and is not restricted to that metric type.

Entries: Counter; Counter Value; Gauge.

Time / order / cause

Timestamp order, recording order, and causal dependence are different relationships. Clock disagreement and buffering can change apparent order; a logical clock supplies its defined ordering properties rather than physical elapsed time.

Entries: Event timestamp; Observed timestamp; Clock skew; Logical clock.

Elapsed time / CPU time

Elapsed operation time can include execution, queueing, waiting, and remote work. CPU time measures processor service under the stated accounting rules. A long span is not a measurement of CPU consumption.

Entries: Span; CPU time; Elapsed time.

Data race / race condition

Data race is a memory-model condition involving conflicting accesses and required ordering. Race condition also covers timing-dependent logic errors that can remain even when individual memory accesses use synchronization.

Entries: Data race; Race condition; Happens-before.

Flame graph / flame chart

A flame graph aggregates weighted stacks; its horizontal axis is not a timeline. A flame chart preserves chronological placement. Read the measurement unit before interpreting rectangle width.

Entries: Flame graph; Flame chart; Inclusive time.

Exporter

An OpenTelemetry Exporter sends telemetry from a pipeline. A Prometheus exporter exposes metrics for scraping. The same word therefore describes different positions and transport behavior.

Entries: Exporter; Prometheus exporter; Scrape.

MTTR

Repair, recovery, restoration, and resolution can use different boundaries. State the chosen expansion, start event, end event, population, and treatment of incidents still in progress before comparing averages.

Entries: MTTR; MTTD; MTBF.

JFR / jfr

JFR is the Java Flight Recorder framework and recorded-event facility. The jfr command inspects or manipulates recording files. A framework, a recording artifact, and a file utility are distinct objects of analysis.

Entries: JFR; jfr command.

Generative debugging / agentic debugging

Generative debugging is the earlier defect-model construction proposal summarized here. Agentic debugging describes a workflow using an AI agent to plan and perform investigative actions. Neither name establishes that a proposed fix has been verified.

Entries: Generative debugging; Agentic debugging.

Malnarrative / malware narrative

A malnarrative is maliciously modified evidence. A malware narrative records malicious software or activity and may itself be an accurate artifact. A missing or corrupt record alone does not establish either interpretation.

Entries: Malnarrative; Malware narrative.

Trace / narrative / projection

An instrumented execution trace is one selected representation of behavior. General traces extend analysis to other artifact structures. Projective debugging builds an executable model from recorded observations; its fidelity depends on the trace and translation.

Entries: Software narrative; General trace; Projective debugging.

Alias index

Selected alternative names, acronym expansions, and common shorthand point to their primary entries and printed pages. Capitalization is retained where it distinguishes meanings.

Abductive reasoning See Abduction, 6.

Abstract Debugging Commands See ADC, 10.

Accelerated Disassembly, Reconstruction and Reversing See ADDR, 10.

Address space layout randomization See ASLR, 18.

Analysis Pattern Network See APN, 17.

Application binary interface See ABI, 6.

ARM64EC (Emulation Compatible) See Virtualized Process (ARM64EC and CHPE), 255.

Artifacts, Checklists, Patterns See ACP, 8.

Artifacts, Patterns, Mechanisms See APM, 17.

Artificial Debugger See ArtDbg, 18.

ASan See AddressSanitizer, 11.

ATID (Adjoint Thread ID) See Adjoint Thread of Activity, 11.

Auto-instrumentation See Automatic instrumentation, 19.

Backtrace See Stack trace [Artifacts], 216.

BLOB Memory Snapshot See Memory Snapshot, 140.

Blocked Thread (software) See Blocked Thread, 22.

Bug See Defect, 55.

Bugcheck See Bug check, 25.

Call stack See Stack trace [Artifacts], 216.

C Bounded Model Checker See CBMC, 29.

CFG See Control-flow graph, 38.

CHPE (Compiled Hybrid Portable Executable) See Virtualized Process (ARM64EC and CHPE), 255.

Closed Memory Region See Memory Region, 140.

Cloud Analysis Patterns See CAPS, 27.

Cloud memory space See Orbifold Memory Space, 163.

CLR Thread See Runtime Thread (CLR), 195.

CoData See CoTrace, 40.

Collapsed Stack Trace See Quotient Stack Trace, 184.

CoLog See CoTrace, 40.

Compact Stack Trace See Quotient Stack Trace, 184.

Coupled Processes (strong) See Coupled Processes, 41.

Custom Exception Handler (user space) See Custom Exception Handler, 47.

Data codiagnostics See Software codiagnostics, 206.

Data Execution Prevention See DEP, 57.

Defect Mechanism Patterns See DMP, 64.

Deferred procedure call See DPC, 67.

Dereference Fixpoints See Dereference Fixpoint, 57.

DFSan See DataFlowSanitizer, 50.

Diagnostic Analysis Gestures See Diags, 60.

Diagnostic problem pattern See Problem pattern, 178.

Diagnostics See Software diagnostics, 206.

DLQ See Dead-letter queue, 51.

Dump Artifact + Trace Artifact See Small DA+TA, 205.

Event Tracing for Windows See ETW, 76.

Exception Thread See Exception Stack Trace [Memory patterns], 79.

Executable and Linkable Format See ELF, 73.

Execution Residue (unmanaged space, user) See Execution Residue, 79.

Extended Berkeley Packet Filter See eBPF, 72.

Filter Stack Trace See Stack Trace (file system filters), 217.

Flight Recorder See JFR, 119.

Frame Usage Signature See Aggregated Frames, 12.

Garbage collection root See GC root, 91.

GC See Garbage collection, 91.

General Memory Hierarchy See Memory Hierarchy (General), 138.

Git bisect See Regression bisection, 189.

GNU Debugger See GDB, 92.

Hidden Exception (user space) See Hidden Exception, 100.

High Contention (critical sections) See High Contention, 102.

Hooked Functions (user space) See Hooked Functions, 104.

hs_err_pid log See JVM fatal error log, 121.

HWASan See Hardware-assisted AddressSanitizer, 97.

Imageless Analysis See Symbolless or Imageless Analysis, 225.

Instruction pointer See Program counter, 180.

Interrupt request level See IRQL, 118.

I/O request packet See IRP, 118.

Java Debug Wire Protocol See JDWP, 119.

Java Flight Recorder See JFR, 119.

Java Management Extensions See JMX, 120.

Java Platform Debugger Architecture See JPDA, 120.

jfr See jfr command, 119.

JVM Tool Interface See JVM TI, 121.

Last-chance exception See Second-chance exception, 197.

Lateral Damage (general) See Lateral Damage, 124.

Little's law See Little's law, 127.

Local Buffer Overflow (user space) See Local Buffer Overflow, 129.

Local Interpretable Model-agnostic Explanations See LIME, 125.

LOH See Large object heap, 123.

LSan See LeakSanitizer, 125.

Manual Dump (process) See Manual Dump, 134.

Mean time between failures See MTBF, 151.

Mean time to detection See MTTD, 151.

Mean time to repair / recovery / restore / resolution See MTTR, 151.

Mechanism See Defect mechanism, 55.

Memosemiotics See Memiotics, 136.

Metadefect Template Library See MdTL, 136.

Missing Thread (user space) See Missing Thread, 147.

Mixed (Nested) Exception See Mixed Exception, 148.

Modified Condition/Decision Coverage See MC/DC, 135.

MSan See MemorySanitizer, 140.

NarraChain See Narrachain, 153.

Native Memory Tracking See NMT, 156.

Non-uniform memory access See NUMA, 160.

Not My Version (software) See Not My Version, 158.

Object Distribution Anomaly (IRP) See Object Distribution Anomaly, 160.

Open Memory Region See Memory Region, 140.

OpenTelemetry See OTel, 164.

OpenTelemetry Protocol See OTLP, 164.

Out-of-memory killer See OOM killer, 162.

Passive Thread (user space) See Passive Thread, 168.

Pattern-Oriented Software Diagnostics See Pattern-oriented diagnostics, 170.

Patterns–View–Controller See PVC, 183.

pcap See PCAP, 171.

Performance monitoring unit See PMU, 174.

Physical space See Physical memory, 173.

POD See Pattern-oriented diagnostics, 170.

Portable Executable See PE, 171.

Probe effect See Observer effect, 161.

Problem Solving as Code See PSaC, 182.

Procmon See Process Monitor, 180.

Program Database See PDB [Artifacts], 171.

Program Lines of Trace See PLOT, 174.

Prometheus Query Language See PromQL, 181.

Rapid Software Diagnostics Process See RSDP, 194.

Rate, Errors, Duration See RED method, 187.

RCA See Root cause analysis, 193.

Read-copy-update See RCU, 186.

Read-write lock See Reader-writer lock, 186.

RED See RED method, 187.

REPL Streaming See REPLS, 190.

Retrieval-augmented generation See RAG, 185.

Riemann Root Cause Analysis Pattern Language See Riemann RCA Pattern Language, 193.

Round-trip time See RTT, 194.

RSS See Resident set size, 191.

SBFL See Spectrum-based fault localization, 212.

Self time See Exclusive time, 79.

Service level agreement See SLA, 205.

Service level indicator See SLI, 205.

Service level objective See SLO, 205.

SHapley Additive exPlanations See SHAP, 201.

Site reliability engineering See SRE, 214.

SM See Supporting Module, 223.

Software defect See Defect, 55.

Software execution artifact See Diagnostic artifact, 59.

Span ID See Span identifier, 209.

Spatial Trace See Spatial Form, 210.

Stack Trace Collection (unmanaged space) See Stack Trace Collection, 216.

Stop error See Bug check, 25.

Structured Memory Snapshot See Memory Snapshot, 140.

Symbolless Analysis See Symbolless or Imageless Analysis, 225.

System Call (memory analysis pattern) See System Call [Memory patterns], 226.

TLS storage See Thread-local storage, 232.

Tool Facade See Tool Façade, 235.

Total time See Inclusive time, 108.

Trace D'Enfant See Trace D'Enfant, 237.

Trace Hyperparameters See Trace Parameters and Hyperparameters, 242.

Trace ID See Trace identifier, 239.

Trace Parameters See Trace Parameters and Hyperparameters, 242.

Trace Precision See Trace Precision and Recall, 243.

Trace Recall See Trace Precision and Recall, 243.

TSan See ThreadSanitizer, 233.

TTD See Time Travel Debugging, 234.

TUW See Troubleshooting Unit of Work, 248.

UBSan See UndefinedBehaviorSanitizer, 249.

Unknown Module See Unknown Component, 250.

USE See USE method, 252.

Utilization, Saturation, Errors See USE method, 252.

Vendor Collection See Module Collection, 148.

Virtual space See Virtual memory, 255.

Wait Chain (general) See Wait Chain, 257.

Waiting Thread Time (kernel dumps) See Waiting Thread Time, 260.

Windows Performance Analyzer See WPA, 264.

Windows Performance Recorder See WPR, 264.

XAI See Explainable AI, 80.

Zero-code instrumentation See Automatic instrumentation, 19.

Zero Fault Software Diagnostics See Zero Fault diagnostics, 265.

Domain index

Each headword links to its entry and is followed by its page number. The primary domain gives a practical reading route; related terms connect concepts across domains.

Foundations

Diagnostic reasoning, pattern languages, mechanisms, and problem solving. 92 entries.

Abduction, 6; ACP, 8; ADC, 10; Agile diagnostics, 13; Alien Component, 14; Analysis pattern, 15; Analysis pattern duality, 15; Analysis Pattern Square, 15; APM, 17; APN, 17; ArtDbg, 18; Behavioral pattern, 21; Be Language, 21; Bohrbug, 23; Causal model, 29; Concrete analysis pattern, 35; Concrete problem pattern, 35; Debugging, 54; Debugware, 55; Defect, 55; Defect mechanism, 55; Dependability, 57; Diagnostic artifact, 59; Diagnostic evidence, 59; Diagnostic hypothesis, 59; Diagnostic indicator, 59; Diagnostic operad, 59; Diagnostic report, 60; Diagnostics Canvas, 60; Diagnostics-driven development, 60; Diags, 60; Diamathics, 61; DMP, 64; Elementary analysis pattern, 72; Elementary diagnostic pattern, 73; Error, 75; Failure, 81; Fault, 83; First fault, 86; Fooled by Abbreviation, 88; Fooled by Description, 88; General analysis pattern, 92; General problem pattern, 92; Generative debugging, 92; Habitual Reply, 95; Hooksware, 105; MdTL, 136; Memiotics, 136; Memoretics, 136; Metaphorical bijectionism, 144; Missing Space, 147; Myopic Troubleshooting and Debugging, 153; Need the crash dump. Period., 154; Negative evidence, 154; No Question, 157; Pattern-based diagnostics, 169; Pattern-driven diagnostics, 170; Pattern language, 170; Pattern-oriented diagnostics, 170; Pattern repertoire, 170; Premortem diagnostics, 176; Problem description, 177; Problem description pattern, 177; Problem pattern, 178; Projective debugging, 181; PSaC, 182; PVC, 183; REPLS, 190; Root cause analysis, 193; RSDP, 194; Software codiagnostics, 206; Software diagnostics, 206; Software diagnostics engineering, 207; Software diagnostic space, 207; Software forensics, 207; Software pathology, 208; Software prognostics, 208; Structural pattern, 222; Symbolless or Imageless Analysis, 225; Systematic diagnostics, 225; Systemic diagnostics, 226; Thinking-based diagnostics, 231; Troubleshooting, 247; Unified Debugging Pattern Language, 249; Use-case Deviation, 252; Wild Explanation, 262; Word of Mouth, 263; Workaround, 263; Workaround pattern, 263; Wrong Dump, 264; Zero Fault diagnostics, 265; Zippocricy, 265.

Narratology

Software narratives, interpretation, models, and visual representation. 21 entries.

Analysis narrative, 14; Dia|gram, 60; Extended software trace, 81; Generalized software trace, 92; General trace, 92; Logue, 131; Loxel, 132; Malnarrative, 132; Malware narrative, 132; Mexel, 145; Narrachain, 153; Narralog, 153; Narreme, 153; Pattern narrative, 170; PLOT, 174; Riemann RCA Pattern Language, 193; Software antinarrative, 206; Software narrative, 207; Software narratology, 207; Software plot, 208; Special trace, 211.

Trace patterns

Named patterns for trace, log, text, and narrative analysis. 268 entries.

Abnormal Value, 6; Activity Disruption, 9; Activity Divergence, 9; Activity Overlap, 9; Activity Packet, 9; Activity Quantum, 9; Activity Region, 9; Activity Theatre, 10; Adjoint Message, 11; Adjoint Space, 11; Adjoint Thread of Activity, 11; Anchor Messages, 16; Background and Foreground Components, 20; Back Trace, 20; Basic Facts, 21; Bethe Ansatz, 21; Bifurcation

Point, 21; Blackout, 22; Braid Group, 24; Braid of Activity, 24; Break-in Activity, 24; Calibrating Trace, 26; Canonical Trace, 27; Cartesian Trace, 28; Case Messages, 28; Causal Chains, 28; Causal History, 28; Causal Messages, 29; Characteristic Message Block, 30; Circular Trace, 31; CoActivity, 32; Collapsed Message, 33; Combed Trace, 34; Container Trace, 37; Cord of Activity, 39; Correlated Discontinuity, 39; Corrupt Message, 40; CoTrace, 40; Counter Value, 40; Coupled Activities, 41; Critical Point, 45; Data Association, 48; Data Flow, 49; Data Interval, 50; Data Reversal, 50; Data Selector, 51; De Broglie Trace Duality, 53; Declarative Trace, 55; Defamiliarizing Effect, 55; Defect Group, 55; Definition Trace, 56; Delay Dynamics, 56; Denormalized Message, 56; Density Distribution, 56; Dialogue, 61; Diegetic Messages, 61; Discontinuity, 62; Dominant Event Sequence, 65; Drone Message, 68; Embedded Trace, 73; Empty Trace, 74; Equivalent Messages, 74; Error Distribution, 75; Error Message, 75; Error Powerset, 75; Error Thread, 76; Event Sequence Order, 77; Event Sequence Phase, 77; Exception Stack Trace, 78; Explanation Trace, 80; Factor Group, 81; False Positive Error, 83; Feature of Activity, 85; Feynman Trace, 85; Fiber Bundle, 85; Fiber of Activity, 86; File Size, 86; Flag, 87; Focus of Tracing, 87; Fourier Activity, 88; Galois Trace, 91; Generative Trace, 93; Glued Activity, 93; Gossip, 94; Guest Component, 94; Hedges, 99; Hidden Error, 100; Hidden Facts, 101; Iconic Trace, 106; Identification Messages, 107; Implementation Discourse, 107; Implicit Message, 107; Impossible Trace, 107; Incomplete History, 108; Indexical Trace, 109; Indirect Facts, 109; Indirect Message, 109; Inter-Correlation, 115; Interspace, 116; Intra-Correlation, 116; Intrinsic ID, 116; Karnaugh Map, 121; Last Activity, 123; Layered Periodization, 125; Linked Messages, 126; Macrofunction, 132; Marked Message, 135; Master Trace, 135; Measurement, 136; Message Annotations, 141; Message Bond, 141; Message Change, 141; Message Complex, 142; Message Context, 142; Message Cover, 142; Message Directory, 142; Message Embedding, 142; Message Essence, 142; Message Flow, 143; Message Interleave, 143; Message Invariant, 143; Message Kinetics, 143; Message Metadata, 143; Message Pattern, 144; Message Set, 144; Message Space, 144; Message Type, 144; Meta Trace, 145; Milestones, 145; Minimal Trace, 146; Missing Component, 146; Missing Data, 146; Missing Message, 147; Moduli Trace, 150; Motif, 150; Motivic Trace, 151; Multidimensional Message, 151; News Value, 155; No Activity, 156; No Trace Metafile, 159; Null Reference, 159; Opposition Messages, 162; Original Message, 163; Ornament, 164; Palimpsest Messages, 167; Pattern Vector, 170; Periodic Error, 172; Periodic Message Block, 172; Phantom Activity, 172; Phase Transition, 173; Piecewise Activity, 173; Pivot Message, 173; Poincaré Trace, 174; Polytrace, 175; Projective Space, 181; Punctuated Activity, 182; Quotient Trace, 184; Random Data, 185; Recovered Messages, 187; Relative Density, 189; Renormalization, 190; Resume Activity, 192; Ruptured Trace, 195; Script Messages, 197; Semantic Field, 199; Semantic Mapping, 199; Sequence Repeat Anomaly, 200; Serial Trace, 200; Shared Point, 202; Sheaf of Activities, 203; Signal, 203; Significant Event, 203; Significant Interval, 204; Silent Messages, 204; Singleton Event, 204; Singleton Trace, 204; Small DA+TA, 205; Sorted Trace, 208; Sparse Trace, 210; Spatial Form, 210; Split Message, 213; Split Trace, 213; State and Event, 220; State Dump, 220; Statement Density and Current, 220; Strand of Activity, 222; Surveyor, 223; Tensor Trace, 230; Text Trace, 231; Thread of Activity, 232; Time Delta, 233; Timeout, 234; Time Scale, 234; Trace Acceleration, 236; Trace Amplitude, 236; Trace Benchmark, 236;

Trace Bias, 236; Trace Braidoids, 236; Trace Capacitance, 237; Trace Class, 237; Trace Constants, 237; Trace Context, 237; Trace Contour, 237; Trace D'Enfant, 237; Trace Dimension, 238; Trace Divergence, 238; Trace Extension, 238; Trace Fabric, 238; Trace Field, 238; Trace Flux, 239; Trace Foliation, 239; Trace Frames, 239; Trace Graph, 239; Trace Homotopy, 239; Trace Inductance, 240; Trace Join, 240; Trace Lattice, 240; Trace Mask, 240; Traceme, 240; Trace Model, 241; Trace Molecule, 241; Trace Multiphysics, 241; Trace Nerve, 241; Trace Network, 241; Trace Ontology, 241; Trace Parameters and Hyperparameters, 242; Trace Partition, 242; Trace Path, 242; Trace Plan, 242; Trace Polynomial, 242; Trace Precision and Recall, 243; Trace Presheaf, 243; Trace Pressure, 243; Trace Quilt, 243; Trace Reactance, 243; Trace Retract, 243; Trace Schema, 244; Trace Shape, 244; Trace Sharding, 244; Trace Similarity, 244; Trace Skeleton, 244; Trace Sketch, 244; Trace Skill, 245; Traces of Individuality, 245; Trace String, 245; Trace Summary, 245; Trace Temperature, 245; Trace Tessellation, 246; Trace Variance, 246; Trace Viewpoints, 246; Trace Volume, 246; Trace Window, 246; Trace World, 246; Translated Message, 247; Truncated Data, 248; Truncated Trace, 248; UI Message, 249; Ultrasimilar Messages, 249; Unsynchronized Messages, 251; Unsynchronized Traces, 251; Use Case Trail, 252; Visibility Limit, 256; Visitor Trace, 256; Vocabulary Index, 256; Watch Thread, 261; Weave of Activity, 261; Whisker Trace, 262; Working Set, 264.

Memory patterns

Behavioral, structural, and malware-related memory analysis patterns. 433 entries.

Abridged Dump, 7; Accidental Lock, 7; Activation Context, 8; Active Space, 8; Active Thread (Mac OS X), 8; Active Thread (Windows), 8; Activity Resonance, 10; Address Representations, 10; Affine Thread, 11; Aggregated Frames, 12; Aggregate Snapshot, 12; Analysis Summary, 15; Anchor Region, 16; Annotated Disassembly (JIT .NET code), 16; Annotated Disassembly (unmanaged code), 16; Annotated Stack Trace, 17; Black Box, 21; Blocked DPC, 22; Blocked Queue (LPC/ALPC), 22; Blocked Thread, 22; Blocked Thread (hardware), 23; Blocked Thread (timeout), 23; Blocking File, 23; Blocking Module, 23; Broken Link, 24; Busy System, 25; Caller-n-Callee, 26; C++ Exception, 29; Changed Environment, 30; Clone Dump, 31; Cloud Environment, 32; C++ Object, 32; Code Path Locality, 33; Coincidental Error Code, 33; Coincidental Frames, 33; Coincidental Symbolic Information, 33; COM Exception, 34; COM Object, 34; Constant Subtrace, 36; Context Pointer, 37; Corrupt Dump, 40; Corrupt Structure, 40; Coupled Machines, 41; Coupled Modules, 41; Coupled Processes, 41; Coupled Processes (semantics), 41; Coupled Processes (weak), 42; Crashed Process, 44; Crash Signature, 44; Crash Signature Invariant, 44; Critical Region, 45; Critical Section Corruption, 46; Critical Stack Trace, 46; Custom Exception Handler, 47; Custom Exception Handler (kernel space), 47; Data Alignment (page boundary), 48; Data Contents Locality, 49; Data Correlation (CPU times), 49; Data Correlation (function parameters), 49; Deadlock (critical sections), 51; Deadlock (executive resources), 51; Deadlock (LPC), 52; Deadlock (managed space), 52; Deadlock (mixed objects, kernel space), 52; Deadlock (mixed objects, user space), 52; Deadlock (.NET Finalizer), 52; Deadlock (self), 53; Debugger Bug, 53; Debugger Omission, 54; Dereference Fixpoint, 57; Dereference Nearpoint, 57; Design Value, 57; Deviant Module, 58; Deviant Token, 58; Diachronic Module, 58; Dialog Box, 60; Directing Module, 61; Disassembly Ambiguity, 62; Disassembly Summary, 62; Disconnected Network Adapter, 62; Disk Packet Buildup, 63;

Dispatch Level Spin, 63; Distributed Exception (managed code), 63; Distributed Spike, 63; Distributed Wait Chain, 64; Divide by Zero (kernel mode), 64; Divide by Zero (user mode), 64; Double Free (kernel pool), 66; Double Free (process heap), 66; Double IRP Completion, 67; DPC Stack Collection, 67; Driver Device Collection, 68; Dry Weight, 68; Dual Stack Trace, 69; Dump Context, 69; Duplicated Module, 69; Duplicate Extension, 70; Dynamic Memory Corruption (kernel pool), 71; Dynamic Memory Corruption (managed heap), 71; Dynamic Memory Corruption (process heap), 71; Early Crash Dump, 72; Effect Component, 72; Embedded Comment, 73; Empty Stack Trace, 73; Encoded Pointer, 74; Environment Hint, 74; Error Reporting Fault, 76; Eventual Dumps, 76; Exception Collection, 78; Exception Module, 78; Exception Reporting Thread, 78; Exception Stack Trace, 79; Execution Residue, 79; Execution Residue (managed space), 79; Execution Residue (unmanaged space, kernel), 80; Fake Module, 82; False Effective Address, 82; False Frame, 82; False Function Parameters, 82; False Local Address, 82; False Memory, 83; False Positive Dump, 83; Fat Process Dump, 83; Fault Context, 84; First Fault Stack Trace, 87; Foreign Module Frame, 88; Foreign Stack, 88; FPU Exception, 88; Frame Pointer Omission, 89; Frame Regularity, 89; Frame Trace, 89; Frozen Process, 90; Function Pointer, 90; Ghost Thread, 93; Glued Stack Trace, 93; Handled Exception (kernel space), 95; Handled Exception (.NET CLR), 95; Handled Exception (user space), 96; Handle Leak, 96; Handle Limit (GDI, kernel space), 96; Handle Limit (GDI, user space), 96; Hardware Activity, 97; Hardware Error, 98; Hidden Call, 100; Hidden Exception, 100; Hidden Exception (kernel space), 101; Hidden Exception (managed space), 101; Hidden IRP, 101; Hidden Module, 101; Hidden Parameter, 102; Hidden Process, 102; Hidden Stack, 102; Hidden Stack Trace, 102; High Contention, 102; High Contention (executive resources), 103; High Contention (.NET CLR monitors), 103; High Contention (.NET heap), 103; High Contention (processors), 103; High Contention (sockets), 103; Historical Information, 104; Hooked Functions, 104; Hooked Functions (kernel space), 105; Hooked Modules, 105; Hooking Level, 105; Hyperdump, 106; Incomplete Session, 108; Inconsistent Dump, 108; Incorrect Stack Trace, 109; Incorrect Symbolic Information, 109; Injected Symbols, 110; Inline Function Optimization (managed code), 110; Inline Function Optimization (unmanaged code), 110; Instrumentation Information, 111; Instrumentation Side Effect, 112; Insufficient Memory (committed memory), 112; Insufficient Memory (control blocks), 112; Insufficient Memory (handle leak), 112; Insufficient Memory (kernel pool), 113; Insufficient Memory (module fragmentation), 113; Insufficient Memory (physical memory), 113; Insufficient Memory (PTE), 113; Insufficient Memory (region), 114; Insufficient Memory (reserved virtual memory), 114; Insufficient Memory (session pool), 114; Insufficient Memory (stack), 114; Insufficient Memory (stack trace database), 115; Internal Stack Trace, 115; Interrupt Stack, 115; Interrupt Stack Collection, 116; Invalid Exception Information, 116; Invalid Handle (general), 116; Invalid Handle (managed space), 117; Invalid Parameter (process heap), 117; Invalid Parameter (runtime function), 117; Invalid Pointer (general), 117; Invalid Pointer (objects), 117; ISA-Specific Code, 118; JIT Code (Java), 120; JIT Code (.NET), 120; Last Error Collection, 123; Last Object, 124; Late Crash Dump, 124; Latent Structure, 124; Lateral Damage, 124; Lateral Damage (CPU mode), 125; Least Common Frame, 125; Livelock, 127; Local Buffer Overflow, 129; Local Buffer Overflow (kernel space), 129; Lost Opportunity, 131; Main Thread, 132; Managed Code Exception, 133; Managed Code Exception (Python), 133;

Managed Code Exception (Scala), 133; Managed Stack Trace, 133; Managed Stack Trace (Python), 134; Managed Stack Trace (Scala), 134; Manual Dump, 134; Manual Dump (kernel), 135; Memory Fibration, 137; Memory Fluctuation (process heap), 138; Memory Hierarchy (General), 138; Memory Leak (I/O completion packets), 138; Memory Leak (.NET heap), 139; Memory Leak (page tables), 139; Memory Leak (process heap), 139; Memory Leak (regions), 139; Memory Region, 140; Memory Snapshot, 140; Message Box, 141; Message Hooks, 143; Mirror Dump Set, 146; Missing Component (general), 146; Missing Component (static linking, user mode), 146; Missing Process, 147; Missing Thread, 147; Missing Thread (kernel space), 147; Mixed Exception, 148; Module Collection, 148; Module Collection (predicate), 149; Module Hint, 149; Module Product Process, 149; Module Stack Trace, 149; Module Variable, 149; Module Variety, 150; Multiple Exceptions (kernel mode), 152; Multiple Exceptions (managed space), 152; Multiple Exceptions (stowed), 152; Multiple Exceptions (user mode), 152; Namespace, 153; Near Exception, 154; Nested Exceptions (managed code), 154; Nested Exceptions (unmanaged code), 155; Nested Offender, 155; Network Packet Buildup, 155; No Component Symbols, 156; No Current Thread, 156; No Data Types, 156; No Process Dumps, 157; Normalized Region, 158; No System Dumps, 158; Not My Thread, 158; Not My Version, 158; Not My Version (hardware), 159; NULL Pointer (code), 159; NULL Pointer (data), 159; Object Distribution Anomaly, 160; Object Distribution Anomaly (.NET heap), 160; OMAP Code Optimization, 161; One-Thread Process, 161; Optimized Code (function parameter reuse), 163; Optimized VM Layout, 163; Origin Module, 164; Out-of-Module Pointer, 164; Overaged System, 165; Packed Code, 165; Paged Out Data, 166; Parameter Flow, 167; Paratext, 167; Passive System Thread (kernel space), 168; Passive Thread, 168; Pass Through Function, 168; Past Process, 168; Past Stack Trace, 169; Patched Code, 169; Pervasive System, 172; Place Trace, 173; Platform-Specific Debugger, 174; Pleiades, 174; Pointer Class, 175; Pointer Cone, 175; Pre-Obfuscation Residue, 176; Problem Exception Handler, 178; Problem Module, 178; Problem Vocabulary, 178; Procedure Call Chain, 179; Process Factory, 180; Punctuated Memory Leak, 183; Quiet Dump, 184; Quotient Stack Trace, 184; Random Object, 185; Raw Pointer, 185; Reduced Symbolic Information, 187; Reference Leak, 187; Region Boundary, 187; Region Clusters, 188; Region Profile, 188; Region Spectrum, 188; Region Strata, 188; Region Summary, 188; Regular Data, 189; Relative Memory Leak, 189; RIP Stack Trace, 193; Rough Stack Trace Collection (unmanaged space), 194; Rough Stack Trace (managed space), 194; Rough Stack Trace (unmanaged space), 194; Runtime Thread (CLR), 195; Runtime Thread (Python, Linux), 195; Same Vendor, 195; Saved Exception Context (.NET), 196; Screwbolt Wait Chain, 197; Self-Diagnosis (kernel mode), 198; Self-Diagnosis (registry), 198; Self-Diagnosis (user mode), 198; Self-Dump, 199; Semantic Split, 199; Semantic Structure (PID.TID), 200; Shadow Stack Trace, 201; Shared Buffer Overwrite, 201; Shared Page, 202; Shared Structure, 202; Shared Thread, 202; Small Value, 206; Snapshot Collection, 206; Software Exception, 207; Source Stack Trace, 209; Special Process, 211; Special Stack Trace, 211; Special Thread (.NET CLR), 211; Spike Interval, 212; Spiking Interrupts, 212; Spiking Interrupts (Linux), 212; Spiking Thread, 213; Stack Overflow (insufficient memory), 214; Stack Overflow (kernel mode), 215; Stack Overflow (managed space), 215; Stack Overflow (software implementation), 215; Stack Overflow (user mode), 215; Stack Trace, 216; Stack Trace Change, 216; Stack Trace Collection, 216;

Stack Trace Collection (CPUs), 216; Stack Trace Collection (I/O requests), 217; Stack Trace Collection (managed space), 217; Stack Trace Collection (predicate), 217; Stack Trace (database), 217; Stack Trace (file system filters), 217; Stack Trace (I/O devices), 218; Stack Trace (I/O request), 218; Stack Trace Motif, 218; Stack Trace Race, 218; Stack Trace Set, 218; Stack Trace Signature, 219; Stack Trace Surface, 219; Step Dumps, 221; Stored Exception, 221; String Hint, 222; String Parameter, 222; Structure Field Collection, 222; Structure Sheaf, 223; Subsystem Modules, 223; Suspended Thread, 223; Swarm of Shared Locks, 224; System Call, 226; System Object, 226; Tampered Dump, 227; Technology-Specific Subtrace (COM client call), 228; Technology-Specific Subtrace (COM interface invocation), 228; Technology-Specific Subtrace (dynamic memory), 228; Technology-Specific Subtrace (JIT .NET code), 229; Template Module, 229; Thread Age, 231; Thread Cluster, 232; Thread Poset, 232; Thread Starvation (normal priority), 233; Thread Starvation (realtime priority), 233; Top Module, 235; Translated Exception, 247; Truncated Dump, 248; Truncated Stack Trace, 248; Ubiquitous Component (kernel space), 248; Ubiquitous Component (user space), 249; Unified Stack Trace, 250; Unknown Component, 250; Unloaded Module, 250; Unrecognizable Symbolic Information, 250; Unsynchronized Dumps, 251; User Space Evidence, 253; Value Adding Process, 253; Value Deviation (stack trace), 253; Value Deviation (structure field), 253; Value References, 254; Variable Subtrace, 254; Version-Specific Extension, 254; Virtualized Process (ARM64EC and CHPE), 255; Virtualized Process (WOW64), 255; Virtualized System, 255; Wait Chain, 257; Wait Chain (C++11, condition variable), 257; Wait Chain (CLR monitors), 257; Wait Chain (critical sections), 257; Wait Chain (executive resources), 257; Wait Chain (LPC/ALPC), 258; Wait Chain (modules), 258; Wait Chain (mutex objects), 258; Wait Chain (named pipes), 258; Wait Chain (nonstandard synchronization), 258; Wait Chain (process objects), 258; Wait Chain (pushlocks), 259; Wait Chain (RPC), 259; Wait Chain (RTL_RESOURCE), 259; Wait Chain (SRW lock), 259; Wait Chain (thread objects), 260; Wait Chain (window messaging), 260; Waiting Thread Time, 260; Waiting Thread Time (user dumps), 261; Well-Tested Function, 262; Well-Tested Module, 262; Wild Code, 262; Wild Pointer, 263; Window Hint, 263; Young System, 264; Zombie Processes, 265.

Telemetry

Instrumentation, collection, transport, and evidence quality. 23 entries.

Attribute, 19; Automatic instrumentation, 19; Backpressure, 20; Collector, 34; Exporter, 81; Head sampling, 98; Instrumentation, 110; Instrumentation scope, 111; Manual instrumentation, 135; Observability, 160; OTel, 164; OTLP, 164; Processor, 180; Receiver, 186; Resource attributes, 191; Sampling, 196; Semantic conventions, 199; Signal, 203; Tail sampling, 227; Telemetry, 229; Telemetry loss, 229; Telemetry pipeline, 229; Trace acquisition pattern, 236.

Logs

Records, timestamps, severity, structured logging, and retention. 12 entries.

Audit log, 19; Correlation identifier, 39; Event timestamp, 77; Log correlation, 130; Log injection, 130; Log level, 130; Log record, 131; Log retention, 131; Log rotation, 131; Observed timestamp, 161; Structured logging, 222; Syslog, 225.

Metrics

Measurements, aggregation, histograms, rates, and queries. 24 entries.

Aggregation, 13; Cardinality, 27; Counter, 40; Cumulative temporality, 46; Delta

temporality, 56; Exemplar, 80; Gauge, 91; Histogram, 104; Histogram bucket, 104; Label, 123; Metric, 145; OpenMetrics, 162; Percentile, 172; Prometheus, 181; Prometheus exporter, 181; PromQL, 181; Quantile, 183; Rate, 185; Recording rule, 186; Scrape, 196; Scrape interval, 197; Staleness, 219; Summary, 223; Time series, 234.

Tracing

Spans, context, correlation, sampling, and distributed traces. 16 entries.

Baggage, 20; Context propagation, 38; Dapper, 48; Distributed tracing, 64; Service map, 200; Span, 209; Span context, 209; Span event, 209; Span identifier, 209; Span kind, 210; Span link, 210; Span metrics, 210; Span status, 210; Trace identifier, 239; traceparent, 242; tracestate, 245.

Performance

Latency, throughput, utilization, queues, and resource analysis. 32 entries.

Cache miss, 26; Call graph, 26; Coordinated omission, 38; CPU profile, 42; CPU time, 43; CPU utilization, 43; Critical path, 45; Elapsed time, 72; Exclusive time, 79; False sharing, 83; Flame chart, 87; Flame graph, 87; Hotspot, 105; Inclusive time, 108; Instrumentation overhead, 111; Instrumentation profiler, 111; Latency, 124; Little’s law, 127; Monotonic clock, 150; Observer effect, 161; Off-CPU profile, 161; Profiling, 180; Queueing delay, 184; Queue length, 184; RED method, 187; Response time, 192; Sampling profiler, 196; Saturation, 196; Tail latency, 227; Throughput, 233; USE method, 252; Wall-clock time, 261.

Reliability

Objectives, alerts, incidents, budgets, and recovery measurements. 26 entries.

Alert, 13; Alert fatigue, 13; Alert grouping, 13; Alerting rule, 13; Alert inhibition, 14; Availability, 19; Black-box monitoring, 22; Burn rate, 25; Dashboard, 48; Error budget, 75; Golden signals, 93; Incident, 107; Incident commander, 108; Monitoring, 150; MTBF, 151; MTTD, 151; MTTR, 151; Postmortem, 176; Runbook, 195; Silence, 204; SLA, 205; SLI, 205; SLO, 205; SRE, 214; Toil, 235; White-box monitoring, 262.

Distributed systems

Causality, clocks, retries, deadlines, and partial failure. 17 entries.

Backoff, 20; Cascading failure, 28; Causal order, 29; Circuit breaker, 31; Clock skew, 31; Deadline, 51; Graceful degradation, 94; Idempotency, 107; Jitter, 120; Load shedding, 129; Logical clock, 130; Network partition, 155; Partial failure, 168; Retry, 192; Retry storm, 193; Timeout, 234; Vector clock, 254.

Networks

Transport behavior, packet evidence, DNS, HTTP, and TLS. 9 entries.

DNS failure, 65; HTTP status code, 106; Packet capture, 165; Packet loss, 166; PCAP, 171; Retransmission, 192; RTT, 194; TCP zero window, 228; TLS alert, 234.

Data systems

Queries, database activity, locks, storage, and consumer lag. 10 entries.

Consumer lag, 36; Database blocking, 48; Dead-letter queue, 51; EXPLAIN, 80; EXPLAIN ANALYZE, 80; Query plan, 183; Replication lag, 190; Slow query log, 205; Transaction isolation, 247; Wait event, 260.

Cloud

Containers, orchestration, resource limits, and health probes. 13 entries.

CAPS, 27; Container log, 37; CrashLoopBackOff, 44; Ephemeral container, 74; Eviction, 77; Health check, 98;

Liveness probe, 128; OOMKilled, 162; Pod event, 174; Readiness probe, 186; Resource limit, 192; Startup probe, 220; Termination message, 230.

Memory

Address spaces, allocation, lifetime, corruption, and pressure. 52 entries.

7, 6; 8, 6; Address space, 11; Anonymous memory, 17; ASLR, 18; Buffer overflow, 25; Committed memory, 34; Copy-on-write, 39; Dangling pointer, 47; DEP, 57; Double free, 66; Frame pointer, 89; Guard page, 94; Heap, 98; Heap corruption, 99; Huge page, 106; Kernel space, 122; Manifold Memory Space, 134; Memory corruption, 137; Memory fragmentation, 138; Memory leak, 138; Memory map, 139; Memory-mapped file, 140; Memory pressure, 140; Memory protection, 140; Memory space, 141; Memory tagging, 141; Nonpaged pool, 157; Null pointer dereference, 159; NUMA, 160; OOM killer, 162; Orbifold Memory Space, 163; Out-of-bounds read, 164; Page cache, 166; Paged pool, 166; Page fault, 166; Page table, 167; Physical address, 173; Physical memory, 173; Pool tag, 175; Reserved memory, 190; Resident set size, 191; Stack, 214; Stack frame, 214; Stack overflow, 214; Stack pointer, 215; Uninitialized memory, 250; Use-after-free, 251; User space, 252; Virtual address, 255; Virtual memory, 255; Working set, 264.

Execution

Threads, synchronization, scheduling, exceptions, and interrupts. 55 entries.

ABI, 6; Access violation, 7; Atomic operation, 18; Calling convention, 27; Condition variable, 36; Contention, 37; Context switch, 38; CPU register, 43; Crash, 43; Critical section, 45; Data race, 50; Deadlock, 51; DPC, 67; errno, 74; Error Message Box, 75; Exception, 77; Fiber, 85; File descriptor, 86; First-chance exception, 86; Futex, 90; Hang, 96; Happens-before, 97; Hard lockup, 97; Integer overflow, 115; IRQL, 118; Kernel mode, 122; Kernel panic, 122; Livelock, 128; Lock convoy, 129; Memory barrier, 137; Mutex, 152; Priority inheritance, 177; Priority inversion, 177; Process, 179; Program counter, 180; Race condition, 184; RCU, 186; Reader-writer lock, 186; Resource exhaustion, 191; Resource leak, 191; Second-chance exception, 197; Segmentation fault, 198; Semaphore, 200; Signal, 203; Soft lockup, 206; Spinlock, 213; Starvation, 220; System call, 226; Thread, 231; Thread-local storage, 232; Thread pool, 232; Undefined behavior, 249; Unresponsive Window, 251; User mode, 252; Zombie process, 265.

Artifacts

Dumps, symbols, object formats, and captured execution state. 32 entries.

Active memory dump, 8; Analytic memory dump, 15; Bug check, 25; Build ID, 25; Complete memory dump, 35; Core dump, 39; Crash dump, 43; dSYM, 68; DWARF, 70; ELF, 73; Exception record, 78; Executable image, 79; Full user-mode dump, 90; IRP, 118; Kernel memory dump, 121; Mach-O, 132; Memory acquisition pattern, 136; Memory dump, 137; Memory dump analysis, 137; Minidump, 145; Module, 148; Module list, 149; PDB, 171; PE, 171; Private symbols, 177; Public symbols, 182; Small memory dump, 205; Stack trace, 216; Symbol file, 224; Symbol server, 225; System Description Snapshot, 226; vmcore, 256.

Debuggers

Execution control, inspection, replay, elementary actions, and debugware. 68 entries.

!analyze, 16; API Query, 17; Axed Code, 19; Axed Data, 20; Breakpoint, 24; Catchpoint, 28; Checking Instrumentation, 30; Checklist, 30; Conditional breakpoint, 36; Configuration Wrapper, 36; Counting Objects, 41; Debug

Event Subscription / Notification, 53; Debugger, 53; Debugger context, 54; Debugger extension, 54; Disassembly, 62; Dual Interface, 68; Dumping Memory, 69; Dumping Object State, 69; Dumping Thread State, 69; Expression evaluation, 81; Fake API, 82; Frozen Process, 90; GDB, 92; Hardware breakpoint, 98; !heap, 98; Heisenbug, 99; Hidden Output, 101; Listing CPUs, 126; Listing Heap Entries, 126; Listing Heaps, 126; Listing Memory Regions, 126; Listing Modules, 126; Listing Object Names, 127; Listing Processes, 127; Listing Stack Traces, 127; Listing Threads, 127; LLDB, 128; Memory examination, 137; Optimized-out variable, 163; !process, 179; Punctuated Execution, 182; Register context, 188; Remote debugging, 189; Reverse debugging, 193; Searching Memory, 197; Self Extractor, 199; Setting Processes, 200; Setting Symbolic Information, 201; Setting Threads, 201; Shared Debug Event State, 202; Shared Variable, 202; Single stepping, 204; Source mapping, 208; Span Differentiator, 209; Stack unwinding, 219; Supporting Module, 223; Symbolication, 224; Symbol path, 225; !thread, 231; Time Travel Debugging, 234; Tool Box, 235; Tool Chain, 235; Tool Façade, 235; Trace Expert, 238; Troubleshooting Unit of Work, 248; Watchpoint, 261; WinDbg, 263.

Native tools

Windows, Linux, and macOS diagnostic tools and facilities. 23 entries.

coredumpctl, 39; crash utility, 44; Driver Verifier, 68; eBPF, 72; ETW, 76; ftrace, 90; kdump, 121; Kernel tracepoint, 122; Kprobe, 123; Lockdep, 130; PageHeap, 166; perf, 172; PMU, 174; PoolMon, 175; ProcDump, 178; Process Explorer, 179; Process Monitor, 180; ptrace, 182; strace, 221; Uprobe, 251; VMMap, 256; WPA, 264; WPR, 264.

Program analysis

Static and dynamic reasoning, slicing, models, and profiling. 44 entries.

Abstract interpretation, 7; ADDR, 10; AddressSanitizer, 11; Alias analysis, 14; Assertion, 18; Bounded model checking, 23; Cachegrind, 26; Callgrind, 27; CBMC, 29; Clang Static Analyzer, 31; Concolic execution, 35; Contract, 38; Control-flow graph, 38; Daikon, 47; Data-flow analysis, 50; DataFlowSanitizer, 50; DRD, 67; Dynamic analysis, 70; Dynamic invariant detection, 70; Dynamic slicing, 71; Go race detector, 94; Hardware-assisted AddressSanitizer, 97; Helgrind, 100; Invariant, 118; KLEE, 122; LeakSanitizer, 125; Loop Construct, 131; Massif, 135; Memcheck, 136; MemorySanitizer, 140; Model checking, 148; Path condition, 169; Path explosion, 169; Postcondition, 175; Precondition, 176; Program slicing, 181; Runtime verification, 195; SPIN, 213; Static analysis, 220; Symbolic execution, 224; Taint analysis, 227; ThreadSanitizer, 233; UndefinedBehaviorSanitizer, 249; Valgrind, 253.

Testing

Reproduction, oracles, coverage, reduction, fuzzing, and injection. 37 entries.

AFL++, 12; Branch coverage, 24; Chaos engineering, 30; Code coverage, 33; Combinatorial testing, 34; Coverage-guided fuzzing, 42; C-Reduce, 45; Delta debugging, 56; Determinism, 58; Differential testing, 61; Fault injection, 84; Fault localization, 84; Flaky test, 87; Fuzzing, 91; Hermetic test, 100; libFuzzer, 125; MC/DC, 135; Metamorphic testing, 144; Minimal reproducer, 145; Mock, 148; Mutation testing, 152; Pairwise testing, 167; Property-based testing, 182; Random seed, 185; Regression bisection, 189; Regression test, 189; Reproducer, 190; Reproducible build, 190; Shrinking, 203; Spectrum-based fault localization, 212;

Statistical debugging, 221; Test-case reduction, 230; Test double, 230; Test fixture, 230; Test harness, 230; Test oracle, 231; Unit test, 250.

Runtimes

JVM, .NET, Python, JavaScript, Go, and managed artifacts. 50 entries.

Allocation profiling, 14; asyncio debug mode, 18; ClrMD, 32; cProfile, 42; Diagnostic port, 59; Dominator tree, 65; dotnet-counters, 65; dotnet-dump, 65; dotnet-gcdump, 65; dotnet-stack, 66; dotnet-trace, 66; Event-loop blocking, 76; EventPipe, 77; faulthandler, 84; Finalization, 86; Garbage collection, 91; GC root, 91; Go execution trace, 93; Goroutine profile, 94; Heap dump, 99; Heap histogram, 99; Heap snapshot, 99; HPROF, 106; jcmd, 118; jdb, 119; JDWP, 119; JFR, 119; jfr command, 119; jmap, 120; JMX, 120; JPDA, 120; jstack, 121; JVM fatal error log, 121; JVM TI, 121; Large object heap, 123; Managed heap, 133; NMT, 156; Node.js diagnostic report, 156; Node.js Inspector, 157; Object pinning, 160; pdb, 171; pprof, 176; Reference cycle, 187; Retained size, 192; Shallow size, 201; SOS, 208; Stop-the-world pause, 221; Thread dump, 232; tracemalloc, 240; Weak reference, 261.

AI and ML

Evaluation, drift, attribution, model behavior, and agent diagnostics. 28 entries.

Agentic debugging, 12; Analysis Pattern Agent, 15; Anomaly detection, 17; Clustering, 32; Concept drift, 35; Confusion matrix, 36; Cross-validation, 46; Data drift, 49; Data leakage, 50; Diagnostic Analysis Pattern Tool, 58; Embedding, 73; Explainable AI, 80; F1 score, 81; Feature extraction, 85; Feature vector, 85; Hallucination, 95; Indirect prompt injection, 109; LIME, 125; LLM-as-a-judge, 128; LLM-assisted debugging, 129; Operational MCP Tool, 162; Overfitting, 165; Precision, 176; RAG, 185; Recall, 186; SHAP, 201; Tool-augmented language model, 235; Training-serving skew, 247.

Standards and formats

These entries identify common interchange conventions, binary and debug formats, transport protocols, and specification-backed terminology. The linked entries carry the relevant primary references and qualifications.

Term	Standard or owner	Diagnostic use
traceparent	W3C Trace Context	Distributed trace identity and context propagation
Baggage	W3C / OpenTelemetry	Contextual values propagated across service boundaries. W3C Candidate Recommendation Snapshot (30 May 2024).
OTLP	OpenTelemetry	Telemetry transport and protocol behavior
Semantic conventions	OpenTelemetry	Common names and meanings for telemetry attributes
OpenMetrics	OpenMetrics / Prometheus	Metric exposition model and wire format
Syslog	IETF	Structured system-log message conventions
PCAP	libpcap / tcpdump	Packet capture artifacts and link-layer context
Retransmission	IETF	Transport recovery and observed packet timing
DNS failure	IETF	Name resolution and response evidence
HTTP status code	IETF	HTTP response semantics and interpretation
TLS alert	IETF	TLS protocol error and closure signaling
ELF	System V ABI / toolchain specifications	Executable, linking, and core-file structure
DWARF	DWARF Debugging Standard	Debug information for types, locations, and unwinding
PDB	Microsoft / LLVM documentation	Program Database debug information
PE	Microsoft	Portable Executable and object-file structures
Mach-O	Apple	Mach object-file and executable structure
Minidump	Microsoft	Selectable process-state streams and capture scope
HPROF	JVM tooling	Heap-profile artifacts and object-graph analysis
JFR	OpenJDK / Java platform	Runtime event recording and diagnostic files
Node.js Inspector	Node.js / Chrome DevTools	Runtime inspection through the DevTools protocol

Primary references

Every dictionary entry cites one or more references below. The reference codes are internal links. Visible URLs identify the publications or official records and remain clickable in digital copies. The two supplied books were consulted in full-text form, and their locators refer to printed pages. Online references were checked on 23 October 2026.

The list contains author publications, primary papers, standards, and official documentation. A page title or version is retained where it is material to the definition. R174, R221, and R298 identify specific documentation releases. Access restrictions or later documentation changes do not alter the identity of the cited publication.

Foundational books and diagnostic theory

R001 Dmitry Vostokov and Software Diagnostics Institute. 2024. *Theoretical Software Diagnostics: Collected Articles, Fourth Edition.* OpenTask. ISBN 978-1-912636-91-4.
https://www.dumpanalysis.org/theoretical-software-diagnostics-book

R002 Dmitry Vostokov and Software Diagnostics Institute. 2023. *Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection, Fifth Edition.* OpenTask. ISBN 978-1-912636-58-7.
https://www.dumpanalysis.org/trace-log-analysis-pattern-reference

R003 Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* DOI 10.1109/TDSC.2004.2.
https://www.landwehr.org/2004-aviz-laprie-randell.pdf

Recent trace analysis patterns

R004 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 263): Trace Ontology.*
https://www.dumpanalysis.org/blog/index.php/2026/07/06/trace-analysis-patterns-part-263/

R005 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 264): Trace World.*
https://www.dumpanalysis.org/blog/index.php/2026/07/08/trace-analysis-patterns-part-264/

R006 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 265): Trace Skill.*
https://www.dumpanalysis.org/blog/index.php/2026/07/10/trace-analysis-patterns-part-265/

R007 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 266): Trace Multiphysics.*
https://www.dumpanalysis.org/blog/index.php/2026/07/13/trace-analysis-patterns-part-266/

R008 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 262): Trace Reactance.*
https://www.dumpanalysis.org/blog/index.php/2026/07/02/trace-analysis-patterns-part-262/

R009 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 261): Karnaugh Map.*
https://www.dumpanalysis.org/blog/index.php/2026/06/28/trace-analysis-patterns-part-261/

R010 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 259): Bethe Ansatz.*
https://www.dumpanalysis.org/blog/index.php/2026/06/23/trace-analysis-patterns-part-259/

Native systems, artifacts, and debugging

R011 Microsoft. *Virtual Address Space.*
https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space

R012 Microsoft. *Memory Protection.*
https://learn.microsoft.com/en-us/windows/win32/memory/memory-protection

R013 Microsoft. *Working Set.*
https://learn.microsoft.com/en-us/windows/win32/memory/working-set

R014 Microsoft. *Heap Functions.*
https://learn.microsoft.com/en-us/windows/win32/memory/heap-functions

R015 Microsoft. *Processes and Threads.* https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads

R016 Microsoft. *Context Switches.* https://learn.microsoft.com/en-us/windows/win32/procthread/context-switches

R017 Microsoft. *Synchronization Objects.* https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-objects

R018 Microsoft. *Structured Exception Handling.* https://learn.microsoft.com/en-us/windows/win32/debug/structured-exception-handling

R019 Free Software Foundation. *Debugging with GDB — Breakpoints, Watchpoints, and Catchpoints.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Breakpoints.html

R020 Free Software Foundation. *Debugging with GDB — Setting Watchpoints.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Set-Watchpoints.html

R021 Free Software Foundation. *Debugging with GDB — Backtraces.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Backtrace.html

R022 Free Software Foundation. *Debugging with GDB — Continuing and Stepping.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Continuing-and-Stepping.html

R023 Free Software Foundation. *Debugging with GDB — Debugging Programs with Multiple Threads.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Threads.html

R024 Free Software Foundation. *Debugging with GDB — Examining Memory.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Memory.html

R025 Free Software Foundation. *Debugging with GDB — Debugging Information in Separate Files.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Separate-Debug-Files.html

R026 Free Software Foundation. *Debugging with GDB — Running Programs Backward.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Reverse-Execution.html

R027 Free Software Foundation. *Debugging with GDB — Stack Frames.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Frames.html

R028 Free Software Foundation. *Debugging with GDB — Registers.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Registers.html

R029 Free Software Foundation. *Debugging with GDB — Source and Machine Code.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Machine-Code.html

R030 Free Software Foundation. *Debugging with GDB — Program Variables.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Variables.html

R031 Free Software Foundation. *Debugging with GDB — Calling Program Functions.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Calling.html

R032 Free Software Foundation. *Debugging with GDB — Memory Tagging.* https://sourceware.org/gdb/current/onlinedocs/gdb.html/Memory-Tagging.html

R033 The LLDB Team. *LLDB Tutorial.* https://lldb.llvm.org/use/tutorial.html

R034 The LLDB Team. *LLDB Symbolication.* https://lldb.llvm.org/use/symbolication.html

R035 The Linux Kernel development community. *Documentation for Kdump — The kexec-based Crash Dumping Solution.* https://docs.kernel.org/admin-guide/kdump/kdump.html

R036 The Linux Kernel development community. *Memory Management Concepts Overview.* https://docs.kernel.org/admin-guide/mm/concepts.html

R037 The Linux Kernel development community. *The /proc Filesystem.*
https://docs.kernel.org/filesystems/proc.html

R038 Steven Rostedt and the Linux Kernel development community. *ftrace — Function Tracer.*
https://docs.kernel.org/trace/ftrace.html

R039 Mathieu Desnoyers. *Using the Linux Kernel Tracepoints.*
https://docs.kernel.org/trace/tracepoints.html

R040 The Linux Kernel development community. *BPF Design Q&A.*
https://docs.kernel.org/bpf/bpf_design_QA.html

R041 Ingo Molnar, Arjan van de Ven, and the Linux Kernel development community. *Runtime Locking Correctness Validator.*
https://docs.kernel.org/locking/lockdep-design.html

R042 Steven Rostedt. *RT-mutex Implementation Design.*
https://docs.kernel.org/locking/rt-mutex-design.html

R043 MITRE. *CWE-119 — Improper Restriction of Operations within the Bounds of a Memory Buffer.*
https://cwe.mitre.org/data/definitions/119.html

R044 MITRE. *CWE-125 — Out-of-bounds Read.*
https://cwe.mitre.org/data/definitions/125.html

R045 MITRE. *CWE-787 — Out-of-bounds Write.*
https://cwe.mitre.org/data/definitions/787.html

R046 MITRE. *CWE-416 — Use After Free.*
https://cwe.mitre.org/data/definitions/416.html

R047 MITRE. *CWE-415 — Double Free.*
https://cwe.mitre.org/data/definitions/415.html

R048 MITRE. *CWE-401 — Missing Release of Memory after Effective Lifetime.*
https://cwe.mitre.org/data/definitions/401

R049 MITRE. *CWE-476 — NULL Pointer Dereference.*
https://cwe.mitre.org/data/definitions/476.html

R050 MITRE. *CWE-190 — Integer Overflow or Wraparound.*
https://cwe.mitre.org/data/definitions/190.html

R051 MITRE. *CWE-457 — Use of Uninitialized Variable.*
https://cwe.mitre.org/data/definitions/457.html

R052 MITRE. *CWE-362 — Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition').*
https://cwe.mitre.org/data/definitions/362.html

R053 MITRE. *CWE-833 — Deadlock.*
https://cwe.mitre.org/data/definitions/833.html

R054 Linux man-pages project. *core(5) — Core Dump File.*
https://man7.org/linux/man-pages/man5/core.5.html

R055 Linux man-pages project. *signal(7) — Overview of Signals.*
https://man7.org/linux/man-pages/man7/signal.7.html

R056 Linux man-pages project. *ptrace(2) — Process Trace.*
https://man7.org/linux/man-pages/man2/ptrace.2.html

R057 Linux man-pages project. *perf_event_open(2) — Set Up Performance Monitoring.*
https://man7.org/linux/man-pages/man2/perf_event_open.2.html

R058 Linux man-pages project. *mmap(2) — Map or Unmap Files or Devices into Memory.*
https://man7.org/linux/man-pages/man2/mmap.2.html

R059 Linux man-pages project. *futex(2) — Fast User-space Locking.*
https://man7.org/linux/man-pages/man2/futex.2.html

R060 Linux man-pages project. *wait(2) — Wait for Process to Change State.*
https://man7.org/linux/man-pages/man2/wait.2.html

R061 Linux man-pages project. *open(2) — Open and Possibly Create a File.* https://man7.org/linux/man-pages/man2/open.2.html

R062 Linux man-pages project. *errno(3) — Number of Last Error.* https://man7.org/linux/man-pages/man3/errno.3.html

R063 The strace project. *strace(1) — Trace System Calls and Signals.* https://man7.org/linux/man-pages/man1/strace.1.html

R064 Microsoft. *User-mode Dump Files.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/user-mode-dump-files

R065 Microsoft. *Complete Memory Dump.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/complete-memory-dump

R066 Microsoft. *Kernel Memory Dump.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/kernel-memory-dump

R067 Microsoft. *Active Memory Dump.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/active-memory-dump

R068 Microsoft. *GFlags and PageHeap.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap

R069 Microsoft. *Driver Verifier.* https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/driver-verifier

R070 Microsoft. *About Event Tracing.* https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing

R071 Mark Russinovich. *Process Monitor.* https://learn.microsoft.com/en-us/sysinternals/downloads/procmon

R072 Mark Russinovich and Andrew Richards. *ProcDump.* https://learn.microsoft.com/en-us/sysinternals/downloads/procdump

R073 Mark Russinovich. *VMMap.* https://learn.microsoft.com/en-us/sysinternals/downloads/vmmap

R074 Microsoft. *PE Format.* https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

R075 The LLVM Project. *The PDB File Format.* https://llvm.org/docs/PDB/index.html

R076 DWARF Debugging Information Format Committee. 2017. *DWARF Debugging Information Format, Version 5.* https://dwarfstd.org/doc/DWARF5.pdf

R077 Xinuos. *ELF Object File Format — ELF Header.* https://gabi.xinuos.com/elf/02-eheader.html

R078 Microsoft. *x64 Calling Convention.* https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

R079 Microsoft. *Overview of ARM64 ABI Conventions.* https://learn.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions?view=msvc-170

R080 Microsoft. *Debugging Tools for Windows SDK and WDK.* https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

R081 The Linux Kernel development community. *Locking Lessons.* https://docs.kernel.org/locking/spinlocks.html

R082 The Linux Kernel development community. *Atomic Types.* https://docs.kernel.org/core-api/wrappers/atomic_t.html

R083 The Linux Kernel development community. *What Is RCU.* https://docs.kernel.org/RCU/whatisRCU.html

R084 The Linux Kernel development community. *Softlockup Detector and Hardlockup Detector.* https://docs.kernel.org/admin-guide/lockup-watchdogs.html

R085 Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. *Kernel Probes.* https://docs.kernel.org/trace/kprobes.html

R086 Srikar Dronamraju. *Uprobe-tracer — Uprobe-based Event Tracing.*
https://docs.kernel.org/trace/uprobetracer.html

R087 Microsoft. *Changing Contexts.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/changing-contexts

R088 Microsoft. *Time Travel Debugging — Overview.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview

R089 Microsoft. *Bug Check Code Reference.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-code-reference2

R090 Microsoft. *Condition Variables.*
https://learn.microsoft.com/en-us/windows/win32/sync/condition-variables

R091 Microsoft. *Page State.*
https://learn.microsoft.com/en-us/windows/win32/memory/page-state

R092 Microsoft. *Creating Guard Pages.*
https://learn.microsoft.com/en-us/windows/win32/memory/creating-guard-pages

R093 Microsoft. *Use PoolMon to Find a Kernel-mode Memory Leak.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/using-poolmon-to-find-a-kernel-mode-memory-leak

R094 Microsoft. *Managing Hardware Priorities.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/managing-hardware-priorities

R095 Microsoft. *Introduction to DPC Objects.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-dpc-objects

R096 Microsoft. *Debugger Exception Handling.*
https://learn.microsoft.com/en-us/windows/win32/debug/debugger-exception-handling

R097 Microsoft. *Thread Local Storage.*
https://learn.microsoft.com/en-us/windows/win32/procthread/thread-local-storage

R098 Microsoft. *Critical Section Objects.*
https://learn.microsoft.com/en-us/windows/win32/sync/critical-section-objects

R099 Apple. *Locating a Missing Debug Symbol File.*
https://developer.apple.com/documentation/xcode/locating-a-missing-debug-symbol-file

R100 Apple. *TN3178 — Checking for and Resolving Build UUID Problems.*
https://developer.apple.com/documentation/technotes/tn3178-checking-for-and-resolving-build-uuid-problems

R101 Microsoft. *Memory Pools.*
https://learn.microsoft.com/en-us/windows/win32/memory/memory-pools

R102 Oracle. *Starvation and Livelock.*
https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

R103 Microsoft. *Packet-driven I/O with Reusable IRPs.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/packet-driven-i-o-with-reusable-irps

R104 Microsoft. *Public and Private Symbols.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/public-and-private-symbols

R105 Microsoft. *Windows Performance Analyzer.*
https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer

R106 Microsoft. *Windows Performance Recorder.*
https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder

R107 Microsoft. *!analyze.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/-analyze

R108 Microsoft. *!heap.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/-heap

R109 Microsoft. *!process.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/-process

R110 Microsoft. *!thread.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/-thread

R111 Microsoft. *Symbol Path for Windows Debuggers.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/symbol-path

R112 Free Software Foundation. *Debugging with GDB — Summary of GDB.*
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Summary.html

R113 Free Software Foundation. *Debugging with GDB — Break Conditions.*
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Conditions.html

R114 Free Software Foundation. *Debugging with GDB — Setting Breakpoints.*
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Set-Breaks.html

R115 Free Software Foundation. *Debugging with GDB — Debugging Remote Programs.*
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Debugging.html

R116 Microsoft. *EXCEPTION_RECORD Structure.*
https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

R117 Microsoft. *Small Memory Dump.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/small-memory-dump

R118 Microsoft. *lm — List Loaded Modules.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/lm--list-loaded-modules-

R119 Microsoft. *Debugger Engine Overview.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-engine-overview

R120 Microsoft. *User Mode and Kernel Mode.*
https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

R121 Mark Russinovich. *Process Explorer.*
https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer

R122 The systemd project. *coredumpctl.*
https://www.freedesktop.org/software/systemd/man/latest/coredumpctl.html

R123 David Anderson. *White Paper: Crash Utility.*
https://crash-utility.github.io/crash_whitepaper.html

R124 The Linux Kernel development community. *NUMA Memory Policy.*
https://docs.kernel.org/admin-guide/mm/numa_memory_policy.html

R125 The Linux Kernel development community. *HugeTLB Pages.*
https://docs.kernel.org/admin-guide/mm/hugetlbpage.html

R126 Johannes Weiner. *PSI — Pressure Stall Information.*
https://docs.kernel.org/accounting/psi.html

R127 The Linux Kernel development community. *Page Cache.*
https://docs.kernel.org/mm/page_cache.html

R128 Microsoft. *Data Execution Prevention.*
https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

R129 Microsoft. *Exploit Protection Reference.*
https://learn.microsoft.com/en-us/defender-endpoint/exploit-protection-reference

R130 Microsoft. *x64 Stack Usage.*
https://learn.microsoft.com/en-us/cpp/build/stack-usage?view=msvc-170

R131 Microsoft. *x64 Exception Handling.*
https://learn.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

R132 Microsoft. *Thread Stack Size.*
https://learn.microsoft.com/en-us/windows/win32/procthread/thread-stack-size

R133 Microsoft. *Fibers.*
https://learn.microsoft.com/en-us/windows/win32/procthread/fibers

R134 Microsoft. *Thread Pools.* https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools

R135 The perf project. *perf(1) — Performance Analysis Tools for Linux.* https://man7.org/linux/man-pages/man1/perf.1.html

Program analysis, testing, runtimes, and AI

R136 Patrick Cousot and Radhia Cousot. 1977. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml

R137 LLVM Project. *LLVM Alias Analysis Infrastructure.* https://llvm.org/docs/AliasAnalysis.html

R138 Python Software Foundation. *The Python Language Reference: Simple statements.* https://docs.python.org/3/reference/simple_stmts.html

R139 Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2023. *CBMC: The C Bounded Model Checker.* https://arxiv.org/abs/2302.02384

R140 LLVM Project. *Source-based Code Coverage.* https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

R141 Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. *DART: directed automated random testing.* https://osl.cs.illinois.edu/publications/conf/pldi/GodefroidKS05.html

R142 Eiffel Software. *Invitation to Eiffel: Design by Contract and Assertions.* https://www.eiffel.org/doc/eiffelstudio/I2E-_Design_by_Contract_and_Assertions

R143 Eiffel Software. *Design by Contract and Assertions.* https://www.eiffel.org/doc/solutions/Design_by_Contract_and_Assertions

R144 LLVM Project. *Data flow analysis: an informal introduction.* https://clang.llvm.org/docs/DataFlowAnalysisIntro.html

R145 Michael D. Ernst and colleagues. 2007. *The Daikon system for dynamic detection of likely invariants.* https://homes.cs.washington.edu/~mernst/pubs/daikon-tool-scp2007-abstract.html

R146 Bogdan Korel and Janusz Laski. 1988. *Dynamic program slicing.* https://www.sciencedirect.com/science/article/pii/0020019088900543

R147 Gerard J. Holzmann / Spin project. *Basic Spin Manual.* https://spinroot.com/spin/Man/Manual.html

R148 James C. King. 1976. *Symbolic Execution and Program Testing.* https://research.ibm.com/publications/symbolic-execution-and-program-testing

R149 Mark Weiser. 1981. *Program slicing.* https://dl.acm.org/doi/10.5555/800078.802557

R150 Linux kernel contributors. *Runtime Verification.* https://docs.kernel.org/trace/rv/runtime-verification.html

R151 LLVM Project. *Clang Static Analyzer.* https://clang.llvm.org/docs/ClangStaticAnalyzer.html

R152 LLVM Project. *Taint Analysis Configuration.* https://clang.llvm.org/docs/analyzer/user-docs/TaintAnalysisConfiguration.html

R153 KLEE project. *Tutorial One — Testing a Small Function.* https://klee-se.org/tutorials/testing-function/

R154 Andreas Zeller and Ralf Hildebrandt. 2002. *Simplifying and Isolating Failure-Inducing Input.* https://www.st.cs.uni-saarland.de/publications/details/zeller-tse-2002/

R155 Git project. *git-bisect.* https://git-scm.com/docs/git-bisect

R156 James A. Jones, Mary Jean Harrold, and John Stasko. 2002. *Visualization of Test Information to Assist Fault Localization.* https://faculty.cc.gatech.edu/~john.stasko/papers/icse02.pdf

R157 Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. *Scalable Statistical Bug Isolation.* https://pages.cs.wisc.edu/~liblit/pldi-2005/

R158 William M. McKeeman. 1998. *Differential Testing for Software.* https://www.cs.tufts.edu/comp/150FP/archive/bill-mckeeman/DifferentailTesting.pdf

R159 T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology; deposited as arXiv:2002.12543 on 28 February 2020. https://arxiv.org/abs/2002.12543

R160 Koen Claessen and John Hughes. 2000. *QuickCheck: a lightweight tool for random testing of Haskell programs.* https://dl.acm.org/doi/10.1145/351240.351266

R161 AFL++ project. *AFL++ Documentation.* https://aflplus.plus/docs/

R162 LLVM Project. *libFuzzer — a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html

R163 PIT project. *Basic concepts.* https://pitest.org/quickstart/basic_concepts/

R164 Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. *The Oracle Problem in Software Testing: A Survey.* https://discovery.ucl.ac.uk/id/eprint/1471263/

R165 Python Software Foundation. *unittest — Unit testing framework.* https://docs.python.org/3/library/unittest.html

R166 LLVM Project. *How to submit an LLVM bug report.* https://llvm.org/docs/HowToSubmitABug.html

R167 pytest project. *Flaky tests.* https://docs.pytest.org/en/stable/explanation/flaky.html

R168 Bazel project. *Test Encyclopedia.* https://bazel.build/reference/test-encyclopedia

R169 Linux kernel contributors. *Fault injection capabilities infrastructure.* https://docs.kernel.org/fault-injection/fault-injection.html

R170 Principles of Chaos Engineering contributors. 2019. *Principles of Chaos Engineering.* https://principlesofchaos.org/

R171 Rick Kuhn, Yu Lei, Raghu Kacker, and Justin Hunter. 2009. *Combinatorial Software Testing.* NIST-hosted preprint, publication 903128; authors as printed on the preprint. Journal version: Computer 42(8), 94–96, August 2009; DOI: 10.1109/MC.2009.253; author order Kuhn, Kacker, Lei, Hunter. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=903128

R172 Python Software Foundation. *unittest.mock — mock object library.* https://docs.python.org/3/library/unittest.mock.html

R173 Reproducible Builds project. *Reproducible Builds: definition.* https://reproducible-builds.org/docs/definition/

R174 PyTorch contributors. *Reproducibility.* PyTorch 2.9 documentation; page updated 26 November 2024. https://docs.pytorch.org/docs/2.9/notes/randomness.html

R175 Hypothesis contributors. *How many times will Hypothesis run my test?* https://hypothesis.readthedocs.io/en/latest/explanation/test-case-count.html

R176 LLVM Project. *AddressSanitizer.* https://clang.llvm.org/docs/AddressSanitizer.html

R177 LLVM Project. *ThreadSanitizer.* https://clang.llvm.org/docs/ThreadSanitizer.html

R178 LLVM Project. *MemorySanitizer.*
https://clang.llvm.org/docs/MemorySanitizer.html

R179 LLVM Project. *LeakSanitizer.*
https://clang.llvm.org/docs/LeakSanitizer.html

R180 LLVM Project. *UndefinedBehaviorSanitizer.*
https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

R181 LLVM Project. *Hardware-assisted AddressSanitizer Design Documentation.*
https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

R182 LLVM Project. *DataFlowSanitizer.*
https://clang.llvm.org/docs/DataFlowSanitizer.html

R183 C-Reduce project. *C-Reduce.*
https://github.com/csmith-project/creduce

R184 Valgrind developers. *Using and understanding the Valgrind core.*
https://valgrind.org/docs/manual/manual-core.html

R185 Valgrind developers. *Memcheck: a memory error detector.*
https://valgrind.org/docs/manual/mc-manual.html

R186 Valgrind developers. *Helgrind: a thread error detector.*
https://valgrind.org/docs/manual/hg-manual.html

R187 Valgrind developers. *DRD: a thread error detector.*
https://valgrind.org/docs/manual/drd-manual.html

R188 Valgrind developers. *Massif: a heap profiler.*
https://valgrind.org/docs/manual/ms-manual.html

R189 Valgrind developers. *Callgrind: a call-graph generating cache and branch prediction profiler.*
https://valgrind.org/docs/manual/cl-manual.html

R190 Valgrind developers. *Cachegrind: a high-precision tracing profiler.*
https://valgrind.org/docs/manual/cg-manual.html

R191 Microsoft. *Fundamentals of garbage collection.*
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

R192 Microsoft. *Large object heap.*
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap

R193 Microsoft. *Weak references.*
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/weak-references

R194 Microsoft. *Garbage collection and performance.*
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/performance

R195 Microsoft. *dotnet-dump.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump

R196 Microsoft. *dotnet-trace.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-trace

R197 Microsoft. *dotnet-counters.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-counters

R198 Microsoft. *dotnet-gcdump.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-gcdump

R199 Microsoft. *SOS debugging extension.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/sos-debugging-extension

R200 Microsoft. *EventPipe.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/eventpipe

R201 Microsoft. *Diagnostic Port.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/diagnostic-port

R202 Microsoft. *dotnet-stack.*
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-stack

R203 Microsoft. *ClrMD: Microsoft.Diagnostics.Runtime.*
https://github.com/microsoft/clrmd

R204 Eclipse Memory Analyzer project. *Dominator Tree.*
https://help.eclipse.org/latest/topic/org.eclipse.mat.ui.help/concepts/dominatortree.html

R205 Eclipse Memory Analyzer project. *Shallow vs. Retained Heap.* https://help.eclipse.org/latest/topic/org.eclipse.mat.ui.help/concepts/shallowretainedheap.html

R206 Eclipse Memory Analyzer project. *Garbage Collection Roots.* https://help.eclipse.org/latest/topic/org.eclipse.mat.ui.help/concepts/gcroots.html

R207 Oracle. *Java SE 25 Troubleshooting Guide: Diagnostic Tools.* https://docs.oracle.com/en/java/javase/25/troubleshoot/diagnostic-tools.html

R208 Oracle. *Java SE 25: The jcmd Command.* https://docs.oracle.com/en/java/javase/25/docs/specs/man/jcmd.html

R209 Oracle. *Java SE 25: The jfr Command.* https://docs.oracle.com/en/java/javase/25/docs/specs/man/jfr.html

R210 Oracle. *Java SE 25 Troubleshooting Guide: Fatal Error Log.* https://docs.oracle.com/en/java/javase/25/troubleshoot/location-fatal-error-log.html

R211 Oracle. *Java SE 25 Garbage Collection Tuning Guide: Introduction to Garbage Collection Tuning.* https://docs.oracle.com/en/java/javase/25/gctuning/introduction-garbage-collection-tuning.html

R212 Oracle. *Java SE 25: Introduction to JMX Technology.* https://docs.oracle.com/en/java/javase/25/jmx/introduction-jmx-technology.html

R213 Oracle. *Java Platform Debugger Architecture.* https://docs.oracle.com/en/java/javase/25/docs/specs/jpda/jpda.html

R214 Oracle. *Java SE 25: The jstack Command.* https://docs.oracle.com/en/java/javase/25/docs/specs/man/jstack.html

R215 Oracle. *Java SE 25: The jmap Command.* https://docs.oracle.com/en/java/javase/25/docs/specs/man/jmap.html

R216 Oracle. *Java SE 25: The jdb Command.* https://docs.oracle.com/en/java/javase/25/docs/specs/man/jdb.html

R217 OpenJDK. 2022. *JEP 421: Deprecate Finalization for Removal.* https://openjdk.org/jeps/421

R218 Python Software Foundation. *faulthandler — Dump the Python traceback.* https://docs.python.org/3/library/faulthandler.html

R219 Python Software Foundation. *tracemalloc — Trace memory allocations.* https://docs.python.org/3/library/tracemalloc.html

R220 Python Software Foundation. *pdb — The Python Debugger.* https://docs.python.org/3/library/pdb.html

R221 Python Software Foundation. *profiling.tracing — Deterministic profiler.* Python 3.15 documentation. https://docs.python.org/3.15/library/profiling.tracing.html

R222 Python Software Foundation. *gc — Garbage Collector interface.* https://docs.python.org/3/library/gc.html

R223 Python Software Foundation. *Developing with asyncio.* https://docs.python.org/3/library/asyncio-dev.html

R224 Node.js contributors. *Diagnostics: Using Heap Snapshot.* https://nodejs.org/learn/diagnostics/memory/using-heap-snapshot

R225 Node.js contributors. *Diagnostic report.* https://nodejs.org/api/report.html

R226 Node.js contributors. *Inspector.* https://nodejs.org/api/inspector.html

R227 Node.js contributors. *Don't Block the Event Loop (or the Worker Pool).* https://nodejs.org/learn/asynchronous-work/dont-block-the-event-loop

R228 Go project. *A Guide to the Go Garbage Collector.* https://go.dev/doc/gc-guide

R229 Go project. *Data Race Detector.* https://go.dev/doc/articles/race_detector

R230 Go project. *runtime/pprof package.* https://pkg.go.dev/runtime/pprof

R231 Go project. *runtime/trace package.* https://pkg.go.dev/runtime/trace

R232 scikit-learn developers. *Novelty and Outlier Detection.* https://scikit-learn.org/stable/modules/outlier_detection.html

R233 scikit-learn developers. *Clustering.* https://scikit-learn.org/stable/modules/clustering.html

R234 scikit-learn developers. *Metrics and scoring: quantifying the quality of predictions.* https://scikit-learn.org/stable/modules/model_evaluation.html

R235 scikit-learn developers. *Common pitfalls and recommended practices.* https://scikit-learn.org/stable/common_pitfalls.html

R236 scikit-learn developers. *Feature extraction.* https://scikit-learn.org/stable/modules/feature_extraction.html

R237 scikit-learn developers. *Cross-validation: evaluating estimator performance.* https://scikit-learn.org/stable/modules/cross_validation.html

R238 TensorFlow contributors. *Get started with TensorFlow Data Validation.* https://www.tensorflow.org/tfx/data_validation/get_started

R239 River contributors. *Concept drift detection.* https://riverml.xyz/0.8.0/examples/concept-drift-detection/

R240 Nils Reimers and Iryna Gurevych. 2019. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks.* https://aclanthology.org/D19-1410/

R241 Patrick Lewis and colleagues. 2020. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* https://arxiv.org/abs/2005.11401

R242 Lianmin Zheng and colleagues. 2023. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena.* https://arxiv.org/abs/2306.05685

R243 Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier.* https://arxiv.org/abs/1602.04938

R244 Scott M. Lundberg and Su-In Lee. 2017. *A Unified Approach to Interpreting Model Predictions.* https://arxiv.org/abs/1705.07874

R245 Shunyu Yao and colleagues. 2022. *ReAct: Synergizing Reasoning and Acting in Language Models.* https://arxiv.org/abs/2210.03629

R246 Timo Schick and colleagues. 2023. *Toolformer: Language Models Can Teach Themselves to Use Tools.* https://arxiv.org/abs/2302.04761

R247 Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. 2023. *SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models.* https://arxiv.org/abs/2303.08896

R248 Kai Greshake and colleagues. 2023. *Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection.* https://arxiv.org/abs/2302.12173

Telemetry, reliability, and distributed systems

R249 OpenTelemetry Authors. *What is OpenTelemetry?* https://opentelemetry.io/docs/what-is-opentelemetry/

R250 OpenTelemetry Authors. *Signals.* https://opentelemetry.io/docs/concepts/signals/

R251 OpenTelemetry Authors. *Instrumentation.* https://opentelemetry.io/docs/concepts/instrumentation/

R252 OpenTelemetry Authors. *Zero-code instrumentation.* https://opentelemetry.io/docs/concepts/instrumentation/zero-code/

R253 OpenTelemetry Authors. *Resources.* https://opentelemetry.io/docs/concepts/resources/

R254 OpenTelemetry Authors. *Instrumentation scope.* https://opentelemetry.io/docs/concepts/instrumentation-scope/

R255 OpenTelemetry Authors. *Semantic Conventions.* https://opentelemetry.io/docs/concepts/semantic-conventions/

R256 OpenTelemetry Authors. *Collector Architecture.* https://opentelemetry.io/docs/collector/architecture/

R257 OpenTelemetry Authors. *Collector Components.* https://opentelemetry.io/docs/collector/components/

R258 OpenTelemetry Authors. *Sampling.* https://opentelemetry.io/docs/concepts/sampling/

R259 OpenTelemetry Authors. *Tracing API.* https://opentelemetry.io/docs/specs/otel/trace/api/

R260 OpenTelemetry Authors. *Logs Data Model.* https://opentelemetry.io/docs/specs/otel/logs/data-model/

R261 OpenTelemetry Authors. *Metrics Data Model.* https://opentelemetry.io/docs/specs/otel/metrics/data-model/

R262 OpenTelemetry Authors. *OTLP Specification.* https://opentelemetry.io/docs/specs/otlp/

R263 OpenTelemetry Authors. *Resiliency.* https://opentelemetry.io/docs/collector/resiliency/

R264 W3C Distributed Tracing Working Group. 2021. *Trace Context.* https://www.w3.org/TR/trace-context/

R265 W3C Distributed Tracing Working Group. 2024. *Propagation format for distributed context: Baggage.* W3C Candidate Recommendation Snapshot, 30 May 2024. https://www.w3.org/TR/baggage/

R266 OpenTelemetry Authors. *Traces.* https://opentelemetry.io/docs/concepts/signals/traces/

R267 OpenTelemetry Authors. *Context propagation.* https://opentelemetry.io/docs/concepts/context-propagation/

R268 OpenTelemetry Authors. *Metrics SDK.* https://opentelemetry.io/docs/specs/otel/metrics/sdk/

R269 Prometheus Authors. *Data model.* https://prometheus.io/docs/concepts/data_model/

R270 Prometheus Authors. *Metric types.* https://prometheus.io/docs/concepts/metric_types/

R271 Prometheus Authors. *Histograms and summaries.* https://prometheus.io/docs/practices/histograms/

R272 Prometheus Authors. *Querying basics.* https://prometheus.io/docs/prometheus/latest/querying/basics/

R273 Prometheus Authors. *Query functions.* https://prometheus.io/docs/prometheus/latest/querying/functions/

R274 Prometheus Authors. *Defining recording rules.* https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/

R275 Prometheus Authors. *Alerting rules.* https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/

R276 Prometheus Authors. *Alertmanager.* https://prometheus.io/docs/alerting/latest/alertmanager/

R277 Prometheus Authors. *Metric and label naming.*
https://prometheus.io/docs/practices/naming/

R278 Prometheus Authors. *Overview.*
https://prometheus.io/docs/introduction/overview/

R279 Benjamin H. Sigelman et al. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.*
https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/

R280 Chris Jones, John Wilkes, and Niall Murphy with Cody Smith; edited by Betsy Beyer. 2016. *Service Level Objectives.*
https://sre.google/sre-book/service-level-objectives/

R281 Marc Alvidrez; edited by Kavita Guliani. 2016. *Embracing Risk.*
https://sre.google/sre-book/embracing-risk/

R282 John Lunney and Sue Lueder; edited by Gary O'Connor. 2016. *Postmortem Culture: Learning from Failure.*
https://sre.google/sre-book/postmortem-culture/

R283 Andrew Stribblehill; edited by Kavita Guliani. 2016. *Managing Incidents.*
https://sre.google/sre-book/managing-incidents/

R284 Marc Brooker. 2019. *Timeouts, retries, and backoff with jitter.*
https://builder.aws.com/content/3EumjoZascWd1oZiEgL8ORlv3qE/timeouts-retries-and-backoff-with-jitter

R285 Rob Ewaschuk; edited by Betsy Beyer. 2016. *Monitoring Distributed Systems.*
https://sre.google/sre-book/monitoring-distributed-systems/

R286 Jess Frame, Anthony Lenton, Steven Thurgood, Anton Tolchanov, and Nejc Trdin with Carmela Quinito. 2018. *Monitoring.*
https://sre.google/workbook/monitoring/

R287 Jamie Wilkinson; edited by Kavita Guliani. 2016. *Practical Alerting from Time-Series Data.*
https://sre.google/sre-book/practical-alerting/

R288 Steven Thurgood with Jess Frame, Anthony Lenton, Carmela Quinito, Anton Tolchanov, and Nejc Trdin. 2018. *Alerting on SLOs.*
https://sre.google/workbook/alerting-on-slos/

R289 Vivek Rau; edited by Betsy Beyer. 2016. *Eliminating Toil.*
https://sre.google/sre-book/eliminating-toil/

R290 Amazon Web Services. *OPS07-BP03: Use runbooks to perform procedures.*
https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/ops_ready_to_support_use_runbooks.html

R291 Alejandro Forero Cuervo; edited by Sarah Chavis. 2016. *Handling Overload.*
https://sre.google/sre-book/handling-overload/

R292 Kubernetes Authors. *Pod Lifecycle.*
https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

R293 Kubernetes Authors. *Liveness, Readiness, and Startup Probes.*
https://kubernetes.io/docs/concepts/workloads/pods/probes/

R294 Kubernetes Authors. *Debug Running Pods.*
https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/

R295 Kubernetes Authors. *Determine the Reason for Pod Failure.*
https://kubernetes.io/docs/tasks/debug/debug-application/determine-reason-pod-failure/

R296 Kubernetes Authors. *Logging Architecture.*
https://kubernetes.io/docs/concepts/cluster-administration/logging/

R297 Kubernetes Authors. *Node-pressure Eviction.*
https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/

R298 PostgreSQL Global Development Group. *Using EXPLAIN.* PostgreSQL 18 documentation, §14.1.
https://www.postgresql.org/docs/18/using-explain.html

R299 PostgreSQL Global Development Group. *The Cumulative Statistics System.* https://www.postgresql.org/docs/current/monitoring-stats.html

R300 PostgreSQL Global Development Group. *Viewing Locks.* https://www.postgresql.org/docs/current/monitoring-locks.html

R301 Brendan Gregg. *The USE Method.* https://www.brendangregg.com/usemethod.html

R302 Julie Dam. 2018. *The RED Method: How to Instrument Your Services.* Write-up of a talk by Tom Wilkie; published 3 August 2018. https://grafana.com/blog/the-red-method-how-to-instrument-your-services/

R303 Jeffrey Dean and Luiz André Barroso. 2013. *The Tail at Scale.* https://research.google/pubs/the-tail-at-scale/

R304 HdrHistogram Authors. *AbstractHistogram API.* https://hdrhistogram.github.io/HdrHistogram/JavaDoc/org/HdrHistogram/AbstractHistogram.html

R305 Leslie Lamport. 1978. *Time, Clocks, and the Ordering of Events in a Distributed System.* https://lamport.azurewebsites.net/pubs/time-clocks.pdf

R306 David L. Mills et al. 2010. *RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification.* https://www.rfc-editor.org/rfc/rfc5905

R307 Wesley Eddy, editor. 2022. *RFC 9293: Transmission Control Protocol (TCP).* https://www.rfc-editor.org/rfc/rfc9293

R308 Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. *RFC 9110: HTTP Semantics.* https://httpwg.org/specs/rfc9110.html

R309 Eric Rescorla. 2018. *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3.* Obsoleted by RFC 9846. The 2018 RFC is retained here for the cited Section 6, Alert Protocol. https://www.rfc-editor.org/rfc/rfc8446

R310 Rainer Gerhards. 2009. *RFC 5424: The Syslog Protocol.* https://www.rfc-editor.org/rfc/rfc5424

R311 Wireshark Authors. *Capturing Live Network Data.* https://www.wireshark.org/docs/wsug_html_chunked/ChapterCapture.html

R312 Wireshark Authors. *TCP Analysis.* https://www.wireshark.org/docs/wsug_html_chunked/ChAdvTCPAnalysis.html

R313 The Tcpdump Group. *pcap-savefile: libpcap savefile format.* https://github.com/the-tcpdump-group/libpcap/blob/master/pcap-savefile.manfile.in

R314 Guy Almes, Sunil Kalidindi, Matt Zekauskas, and Al Morton. 2016. *RFC 7680: A One-Way Loss Metric for IP Performance Metrics (IPPM).* https://www.rfc-editor.org/rfc/rfc7680.html

R315 Guy Almes, Sunil Kalidindi, and Matt Zekauskas. 1999. *RFC 2681: A Round-trip Delay Metric for IPPM.* https://www.rfc-editor.org/rfc/rfc2681.html

R316 Paul Mockapetris. 1987. *RFC 1035: Domain Names—Implementation and Specification.* https://www.rfc-editor.org/rfc/rfc1035.html

R317 Shaler Stidham, Jr. 1974. *Technical Note—A Last Word on L = λW.* Operations Research 22(2), 417–421, March–April 1974. DOI: 10.1287/opre.22.2.417. https://pubsonline.informs.org/doi/10.1287/opre.22.2.417

R318 Friedemann Mattern. 1989. *Virtual Time and Global States of Distributed Systems.* https://vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

R319 David Yanacek. 2019. *Avoiding insurmountable queue backlogs.* https://builder.aws.com/content/3EuRcgkTP1MI0c7zM8W6HL3WIqA/avoiding-insurmountable-queue-backlogs

R320 Microsoft. *Circuit Breaker pattern.* https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

R321 Marc Brooker. 2015. *Exponential Backoff and Jitter.* https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

R322 gRPC Authors. *Deadlines.* https://grpc.io/docs/guides/deadlines/

R323 Seth Gilbert and Nancy Lynch. 2012. *Perspectives on the CAP Theorem.* Computer 45(2), 30–36, February 2012. DOI: 10.1109/MC.2011.389. https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

R324 Giuseppe DeCandia et al. 2007. *Dynamo: Amazon's Highly Available Key-value Store.* https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

R325 Kubernetes Authors. *Resource Management for Pods and Containers.* https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

R326 Kubernetes Authors. *Assign Memory Resources to Containers and Pods.* https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/

R327 Microsoft. *Health Endpoint Monitoring pattern.* https://learn.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring

R328 PostgreSQL Global Development Group. *Transaction Isolation.* https://www.postgresql.org/docs/current/transaction-iso.html

R329 Oracle. *MySQL 8.4 Reference Manual: The Slow Query Log.* https://dev.mysql.com/doc/refman/8.4/en/slow-query-log.html

R330 Apache Kafka Authors. *Monitoring (Kafka 3.6).* https://kafka.apache.org/36/operations/monitoring/

R331 Karen Kent and Murugiah Souppaya. 2006. *NIST SP 800-92: Guide to Computer Security Log Management.* https://csrc.nist.gov/pubs/sp/800/92/final

R332 OWASP Cheat Sheet Series Team. *Logging Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

R333 OpenMetrics Authors. *OpenMetrics Specification.* https://prometheus.io/docs/specs/om/open_metrics_spec/

R334 Prometheus Authors. *Configuration.* https://prometheus.io/docs/prometheus/latest/configuration/configuration/

R335 OpenTelemetry Authors. *Common specification concepts.* https://opentelemetry.io/docs/specs/otel/common/

R336 Jaeger Authors. *Features (Jaeger 2.22).* https://www.jaegertracing.io/docs/2.22/features/

R337 Mike Ulrich. 2016. *Addressing Cascading Failures.* https://sre.google/sre-book/addressing-cascading-failures/

R338 Benjamin Treynor Sloss; edited by Betsy Beyer. 2016. *Introduction.* https://sre.google/sre-book/introduction/

R339 OpenTelemetry Authors. *Logs.* https://opentelemetry.io/docs/concepts/signals/logs/

R340 OpenTelemetry Authors. *Metrics.* https://opentelemetry.io/docs/concepts/signals/metrics/

R341 Brendan Gregg. *Performance Analysis Methodology.* https://www.brendangregg.com/methodology.html

R342 Jaeger Authors. *Service Performance Monitoring (Jaeger 2.22).* https://www.jaegertracing.io/docs/2.22/architecture/spm/

R343 Zhizhou Zhang et al. 2022. *CRISP: Critical Path Analysis of Large-Scale Microservice Architectures.*
https://www.usenix.org/system/files/atc22-zhang-zhizhou.pdf

R344 Amazon Web Services. *Understanding availability.*
https://docs.aws.amazon.com/whitepapers/latest/availability-and-beyond-improving-resilience/understanding-availability.html

R345 Atlassian. *Common Incident Management Metrics.*
https://www.atlassian.com/incident-management/kpis/common-metrics

R346 Amazon Web Services. *Lagging indicators.*
https://docs.aws.amazon.com/prescriptive-guidance/latest/failure-mode-effects-analysis/lagging-indicators.html

R347 Prometheus Authors. *Exporters and integrations.*
https://prometheus.io/docs/instrumenting/exporters/

Memory analysis patterns

R348 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 3) [False Positive Dump].*
https://www.dumpanalysis.org/blog/index.php/2006/11/01/crash-dump-analysis-patterns-part-3/

R349 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 4) [Lateral Damage].*
https://www.dumpanalysis.org/blog/index.php/2006/11/03/crash-dump-analysis-patterns-part-4/

R350 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 7) [Inconsistent Dump].*
https://www.dumpanalysis.org/blog/index.php/2007/01/24/crash-dump-analysis-patterns-part-7/

R351 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 8) [Hidden Exception].*
https://www.dumpanalysis.org/blog/index.php/2007/02/02/crash-dump-analysis-patterns-part-8/

R352 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 10) [Changed Environment].*
https://www.dumpanalysis.org/blog/index.php/2007/03/19/crash-dump-analysis-patterns-part-10/

R353 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 11) [Incorrect Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2007/04/03/crash-dump-analysis-patterns-part-11/

R354 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 12) [No Component Symbols].*
https://www.dumpanalysis.org/blog/index.php/2007/04/20/crash-dump-analysis-patterns-part-12/

R355 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 14) [Spiking Thread].*
https://www.dumpanalysis.org/blog/index.php/2007/05/11/crash-dump-analysis-patterns-part-14/

R356 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 15) [Module Variety].*
https://www.dumpanalysis.org/blog/index.php/2007/05/24/crash-dump-analysis-patterns-part-15/

R357 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 18) [Truncated Dump].*
https://www.dumpanalysis.org/blog/index.php/2007/07/20/crash-dump-analysis-patterns-part-18/

R358 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 19) [Waiting Thread Time].*
https://www.dumpanalysis.org/blog/index.php/2007/07/22/crash-dump-analysis-patterns-part-19/

R359 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 21) [Missing Thread].*
https://www.dumpanalysis.org/blog/index.php/2007/08/12/crash-dump-analysis-patterns-part-21/

R360 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 24) [Coincidental Symbolic Information].*
https://www.dumpanalysis.org/blog/index.php/2007/08/30/crash-dump-analysis-patterns-part-24/

R361 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 27) [Stack Trace Collection].*
https://www.dumpanalysis.org/blog/index.php/2007/09/14/crash-dump-analysis-patterns-part-27/

R362 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 28) [Coupled Processes].*
https://www.dumpanalysis.org/blog/index.php/2007/09/26/crash-dump-analysis-patterns-part-28/

R363 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 29b) [High Contention].*
https://www.dumpanalysis.org/blog/index.php/2008/09/10/crash-dump-analysis-patterns-part-29b/

R364 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 30) [Accidental Lock].*
https://www.dumpanalysis.org/blog/index.php/2007/10/15/crash-dump-analysis-patterns-part-30/

R365 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 31) [Passive Thread].*
https://www.dumpanalysis.org/blog/index.php/2007/10/17/crash-dump-analysis-patterns-part-31/

R366 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 32) [Main Thread].*
https://www.dumpanalysis.org/blog/index.php/2007/10/23/crash-dump-analysis-patterns-part-32/

R367 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 33) [Busy System].*
https://www.dumpanalysis.org/blog/index.php/2007/11/05/crash-dump-analysis-patterns-part-33/

R368 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 34) [Historical Information].*
https://www.dumpanalysis.org/blog/index.php/2007/11/06/crash-dump-analysis-patterns-part-34/

R369 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 35) [Object Distribution Anomaly].*
https://www.dumpanalysis.org/blog/index.php/2007/11/12/crash-dump-analysis-patterns-part-35/

R370 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 36) [Local Buffer Overflow].*
https://www.dumpanalysis.org/blog/index.php/2007/11/14/crash-dump-analysis-patterns-part-36/

R371 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 37) [Early Crash Dump].*
https://www.dumpanalysis.org/blog/index.php/2007/11/21/crash-dump-analysis-patterns-part-37/

R372 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 38) [Hooked Functions].*
https://www.dumpanalysis.org/blog/index.php/2007/11/22/crash-dump-analysis-patterns-part-38/

R373 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 39) [Custom Exception Handler].*
https://www.dumpanalysis.org/blog/index.php/2007/11/23/crash-dump-analysis-patterns-part-39/

R374 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 40a) [Special Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2007/12/10/crash-dump-analysis-patterns-part-40a/

R375 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 41b) [Manual Dump].*
https://www.dumpanalysis.org/blog/index.php/2007/12/17/crash-dump-analysis-patterns-part-41b/

R376 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 42a) [Wait Chain].*
https://www.dumpanalysis.org/blog/index.php/2007/12/14/crash-dump-analysis-patterns-part-42a/

R377 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 43) [Corrupt Dump].*
https://www.dumpanalysis.org/blog/index.php/2008/01/24/crash-dump-analysis-patterns-part-43/

R378 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 44) [Dispatch Level Spin].*
https://www.dumpanalysis.org/blog/index.php/2008/01/25/crash-dump-analysis-patterns-part-44/

R379 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 45) [No Process Dumps].* https://www.dumpanalysis.org/blog/index.php/2008/01/30/crash-dump-analysis-patterns-part-45/

R380 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 46) [No System Dumps].* https://www.dumpanalysis.org/blog/index.php/2008/01/31/crash-dump-analysis-patterns-part-46/

R381 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 47) [Suspended Thread].* https://www.dumpanalysis.org/blog/index.php/2008/02/06/crash-dump-analysis-patterns-part-47/

R382 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 48) [Special Process].* https://www.dumpanalysis.org/blog/index.php/2008/02/12/crash-dump-analysis-patterns-part-48/

R383 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 49) [Frame Pointer Omission].* https://www.dumpanalysis.org/blog/index.php/2008/02/13/crash-dump-analysis-patterns-part-49/

R384 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 50) [False Function Parameters].* https://www.dumpanalysis.org/blog/index.php/2008/02/15/crash-dump-analysis-patterns-part-50/

R385 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 52) [Self-Dump].* https://www.dumpanalysis.org/blog/index.php/2008/02/22/crash-dump-analysis-patterns-part-52/

R386 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 60) [Execution Residue].* https://www.dumpanalysis.org/blog/index.php/2008/04/29/crash-dump-analysis-patterns-part-60/

R387 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 71) [Critical Section Corruption].* https://www.dumpanalysis.org/blog/index.php/2008/07/11/crash-dump-analysis-patterns-part-71/

R388 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 53) [Blocked Thread].* https://www.dumpanalysis.org/blog/index.php/2008/02/27/crash-dump-analysis-patterns-part-53/

R389 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 65) [Not My Version].* https://www.dumpanalysis.org/blog/index.php/2008/06/19/crash-dump-analysis-patterns-part-65/

Performance analysis and measurement

R390 GNU Binutils Authors. *GNU gprof: Introduction to Profiling.* https://sourceware.org/binutils/docs/gprof/Introduction.html

R391 Microsoft. *Understand profiler performance collection methods.* https://learn.microsoft.com/en-us/visualstudio/profiling/understanding-performance-collection-methods-perf-profiler?view=vs-2022

R392 Brendan Gregg. *CPU Flame Graphs.* https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

R393 Brendan Gregg. *Off-CPU Analysis.* https://www.brendangregg.com/offcpuanalysis.html

R394 Brendan Gregg. *Flame Graphs.* https://www.brendangregg.com/flamegraphs.html

R395 GNU Binutils Authors. *GNU gprof: The Call Graph.* https://sourceware.org/binutils/docs/gprof/Call-Graph.html

R396 Microsoft. *Analyze performance by using CPU profiling in the Performance Profiler.* https://learn.microsoft.com/en-us/visualstudio/profiling/cpu-usage?view=vs-2022

R397 Linux man-pages project. *time(7): overview of time and timers.* https://man7.org/linux/man-pages/man7/time.7.html

R398 Linux man-pages project. *clock_gettime(2): clock and time functions.* https://man7.org/linux/man-pages/man2/clock_gettime.2.html

R399 Brendan Gregg. 2017. *CPU Utilization is Wrong.*
https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html

R400 Intel. 2022. *Intel VTune Profiler User Guide 2023.0: CPU Metrics Reference.*
https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/cpu-metrics-reference.html

R401 Linux Kernel Authors. *False Sharing.*
https://kernel.org/doc/html/latest/kernel-hacking/false-sharing.html

R402 Jason Gait. 1986. *A probe effect in concurrent programs.*
https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380160304

R403 Microsoft. *Instrument your .NET application in Visual Studio (C#, C++, Visual Basic, F#).*
https://learn.microsoft.com/en-us/visualstudio/profiling/instrumentation?view=vs-2022

R404 Jim Gray. 1985. *Why Do Computers Stop and What Can Be Done About It?*
https://jimgray.azurewebsites.net/papers/TandemTR85.7_WhyDoComputersStop.pdf

Execution and resource behavior

R405 The Go Authors. 2022. *The Go Memory Model.*
https://go.dev/ref/mem

R406 ISO/IEC JTC 1/SC 22/WG 21. *C++ working draft: undefined behavior [defns.undefined].*
https://eel.is/c++draft/defns.undefined

R407 MITRE. *CWE-772: Missing Release of Resource after Effective Lifetime.*
https://cwe.mitre.org/data/definitions/772.html

R408 MITRE. *CWE-400: Uncontrolled Resource Consumption.*
https://cwe.mitre.org/data/definitions/400.html

R409 Oracle. *The Java Tutorials: Synchronization.*
https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

R410 Joe Duffy. 2006. *Anti-convoy locks in Windows Server 2003 SP1 and Windows Vista.*
https://joeduffyblog.com/2006/12/14/anticonvoy-locks-in-windows-server-2003-sp1-and-windows-vista/

Dictionary of Debugging blog entries

R411 Dmitry Vostokov. 2011. *Dictionary of Debugging: Orbifold Memory Space.*
https://www.dumpanalysis.org/blog/index.php/2011/02/16/dictionary-of-debugging-orbifold-memory-space/

R412 Dmitry Vostokov. 2010. *Dictionary of Debugging: 8.*
https://www.dumpanalysis.org/blog/index.php/2010/07/29/dictionary-of-debugging-8/

R413 Dmitry Vostokov. 2009. *Dictionary of Debugging: Manifold Memory Space.*
https://www.dumpanalysis.org/blog/index.php/2009/12/01/dictionary-of-debugging-manifold-memory-space/

R414 Dmitry Vostokov. 2009. *Dictionary of Debugging: 7.*
https://www.dumpanalysis.org/blog/index.php/2009/11/23/dictionary-of-debugging-7/

R415 Dmitry Vostokov. 2009. *Dictionary of Debugging: Thread.*
https://www.dumpanalysis.org/blog/index.php/2009/10/14/dictionary-of-debugging-thread/

R416 Dmitry Vostokov. 2009. *Dictionary of Debugging: Breakpoint.*
https://www.dumpanalysis.org/blog/index.php/2009/10/06/dictionary-of-debugging-breakpoint/

R417 Dmitry Vostokov. 2009. *Dictionary of Debugging: Virtual Memory.*
https://www.dumpanalysis.org/blog/index.php/2009/10/05/dictionary-of-debugging-virtual-memory/

R418 Dmitry Vostokov. 2009. *Dictionary of Debugging: Physical Memory.*
https://www.dumpanalysis.org/blog/index.php/2009/09/29/dictionary-of-debugging-physical-memory/

R419 Dmitry Vostokov. 2009. *Dictionary of Debugging: Kernel Space.*
https://www.dumpanalysis.org/blog/index.php/2009/09/25/dictionary-of-debugging-kernel-space/

R420 Dmitry Vostokov. 2009. *Dictionary of Debugging: User Space.*
https://www.dumpanalysis.org/blog/index.php/2009/09/24/dictionary-of-debugging-user-space/

R421 Dmitry Vostokov. 2009. *Dictionary of Debugging: Memory Space.*
https://www.dumpanalysis.org/blog/index.php/2009/09/22/dictionary-of-debugging-memory-space/

R422 Dmitry Vostokov. 2009. *Dictionary of Debugging: Memory Dump.*
https://www.dumpanalysis.org/blog/index.php/2009/07/04/dictionary-of-debugging-memory-dump/

R423 Dmitry Vostokov. 2009. *Dictionary of Debugging: Crash.*
https://www.dumpanalysis.org/blog/index.php/2009/06/24/dictionary-of-debugging-crash/

R424 Dmitry Vostokov. 2009. *Dictionary of Debugging: Hang.*
https://www.dumpanalysis.org/blog/index.php/2009/06/23/dictionary-of-debugging-hang/

Trace analysis catalog additions

R425 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 230) [Iconic Trace].*
https://www.dumpanalysis.org/blog/index.php/2023/03/06/trace-analysis-patterns-part-230/

R426 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 231) [Pattern Vector].*
https://www.dumpanalysis.org/blog/index.php/2023/04/02/trace-analysis-patterns-part-231/

R427 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 232) [Trace Tessellation].*
https://www.dumpanalysis.org/blog/index.php/2023/08/28/trace-analysis-patterns-part-232/

R428 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 233) [Canonical Trace].*
https://www.dumpanalysis.org/blog/index.php/2023/09/16/trace-analysis-patterns-part-233/

R429 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 234) [Trace Benchmark].*
https://www.dumpanalysis.org/blog/index.php/2023/09/17/trace-analysis-patterns-part-234/

R430 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 235) [Trace Graph].*
https://www.dumpanalysis.org/blog/index.php/2023/09/24/trace-analysis-patterns-part-235/

R431 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 236) [Trace Amplitude].*
https://www.dumpanalysis.org/blog/index.php/2023/11/06/trace-analysis-patterns-part-236/

R432 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 237) [Trace Class].*
https://www.dumpanalysis.org/blog/index.php/2023/11/18/trace-analysis-patterns-part-237/

R433 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 238) [Trace Volume].*
https://www.dumpanalysis.org/blog/index.php/2023/12/21/trace-analysis-patterns-part-238/

R434 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 239) [Trace Temperature].*
https://www.dumpanalysis.org/blog/index.php/2023/12/26/trace-analysis-patterns-part-239/

R435 Dmitry Vostokov. 2023. *Trace Analysis Patterns (Part 240) [Trace Pressure].*
https://www.dumpanalysis.org/blog/index.php/2023/12/28/trace-analysis-patterns-part-240/

R436 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 241) [Trace Lattice].*
https://www.dumpanalysis.org/blog/index.php/2024/01/05/trace-analysis-patterns-part-241/

R437 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 242) [Trace Bias].*
https://www.dumpanalysis.org/blog/index.php/2024/02/18/trace-analysis-patterns-part-242/

R438 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 243) [Trace Variance].*
https://www.dumpanalysis.org/blog/index.php/2024/02/18/trace-analysis-patterns-part-243/

R439 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 244) [Message Kinetics].*
https://www.dumpanalysis.org/blog/index.php/2024/03/01/trace-analysis-patterns-part-244/

R440 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 245) [Feynman Trace].* https://www.dumpanalysis.org/blog/index.php/2024/04/20/trace-analysis-patterns-part-245/

R441 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 246) [Trace Sketch].* https://www.dumpanalysis.org/blog/index.php/2024/06/19/trace-analysis-patterns-part-246/

R442 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 247) [Trace Model].* https://www.dumpanalysis.org/blog/index.php/2024/06/24/trace-analysis-patterns-part-247/

R443 Dmitry Vostokov. 2024. *Trace Analysis Patterns (Part 248) [Message Type].* https://www.dumpanalysis.org/blog/index.php/2024/12/12/trace-analysis-patterns-part-248/

R444 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 249) [Implicit Message].* https://www.dumpanalysis.org/blog/index.php/2025/03/16/trace-analysis-patterns-part-249/

R445 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 250) [Trace Parameters and Hyperparameters].* https://www.dumpanalysis.org/blog/index.php/2025/03/18/trace-analysis-patterns-part-250/

R446 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 251) [Trace Precision and Recall].* https://www.dumpanalysis.org/blog/index.php/2025/03/28/trace-analysis-patterns-part-251/

R447 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 252) [Trace Polynomial].* https://www.dumpanalysis.org/blog/index.php/2025/08/04/trace-analysis-patterns-part-252/

R448 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 253) [Message Embedding].* https://www.dumpanalysis.org/blog/index.php/2025/09/14/trace-analysis-patterns-part-253/

R449 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 254) [Trace Context].* https://www.dumpanalysis.org/blog/index.php/2025/10/26/trace-analysis-patterns-part-254/

R450 Dmitry Vostokov. 2025. *Trace Analysis Patterns (Part 255) [Trace Plan].* https://www.dumpanalysis.org/blog/index.php/2025/11/01/trace-analysis-patterns-part-255/

R451 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 256) [Trace Divergence].* https://www.dumpanalysis.org/blog/index.php/2026/02/28/trace-analysis-patterns-part-256/

R452 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 257) [Trace Network].* https://www.dumpanalysis.org/blog/index.php/2026/03/08/trace-analysis-patterns-part-257/

R453 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 258) [Spatial Form].* https://www.dumpanalysis.org/blog/index.php/2026/06/07/trace-analysis-patterns-part-258/

R454 Dmitry Vostokov. 2026. *Trace Analysis Patterns (Part 260) [Semantic Mapping].* https://www.dumpanalysis.org/blog/index.php/2026/06/27/trace-analysis-patterns-part-260/

Memory analysis catalog additions: first group

R455 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 1) [Multiple Exceptions (user mode)].* https://www.dumpanalysis.org/blog/index.php/2006/10/30/crash-dump-analysis-patterns-part-1/

R456 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 1d) [Multiple Exceptions (stowed)].* https://www.dumpanalysis.org/blog/index.php/2016/09/08/crash-dump-analysis-patterns-part-1d/

R457 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 2c) [Dynamic Memory Corruption (managed heap)].* https://www.dumpanalysis.org/blog/index.php/2011/10/24/crash-dump-analysis-patterns-part-2c/

R458 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 6) [Invalid Pointer (general)].* https://www.dumpanalysis.org/blog/index.php/2006/12/18/crash-dump-analysis-patterns-part-6/

R459 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 6b) [NULL Pointer (data)].*
https://www.dumpanalysis.org/blog/index.php/2009/04/14/crash-dump-analysis-patterns-part-6b/

R460 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 9a) [Deadlock (critical sections)].*
https://www.dumpanalysis.org/blog/index.php/2007/02/09/crash-dump-analysis-patterns-part-9a/

R461 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 9d) [Deadlock (LPC)].*
https://www.dumpanalysis.org/blog/index.php/2007/11/29/crash-dump-analysis-patterns-part-9d/

R462 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 9g) [Deadlock (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2011/10/17/crash-dump-analysis-patterns-part-9g/

R463 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 13a) [Insufficient Memory (committed memory)].*
https://www.dumpanalysis.org/blog/index.php/2007/05/09/crash-dump-analysis-patterns-part-13a/

R464 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 13d) [Insufficient Memory (PTE)].*
https://www.dumpanalysis.org/blog/index.php/2008/02/04/crash-dump-analysis-patterns-part-13d/

R465 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 13g) [Insufficient Memory (control blocks)].*
https://www.dumpanalysis.org/blog/index.php/2009/12/30/crash-dump-analysis-patterns-part-13g/

R466 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 13j) [Insufficient Memory (stack trace database)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/27/crash-dump-analysis-patterns-part-13j/

R467 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 16a) [Stack Overflow (kernel mode)].*
https://www.dumpanalysis.org/blog/index.php/2007/06/21/crash-dump-analysis-patterns-part-16a/

R468 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 16d) [Stack Overflow (insufficient memory)].*
https://www.dumpanalysis.org/blog/index.php/2019/05/23/crash-dump-analysis-patterns-part-16d/

R469 Dmitry Vostokov. *Managed Code Exception (Scala) and Managed Stack Trace (Scala).*
https://www.dumpanalysis.org/crash-dump-analysis-patterns-part-17b-139b

R470 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 20a) [Memory Leak (process heap)].*
https://www.dumpanalysis.org/blog/index.php/2007/08/06/crash-dump-analysis-patterns-part-20a/

R471 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 20d) [Memory Leak (I/O completion packets)].*
https://www.dumpanalysis.org/blog/index.php/2012/05/19/crash-dump-analysis-patterns-part-20d/

R472 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 22) [Unknown Component].*
https://www.dumpanalysis.org/blog/index.php/2007/08/16/crash-dump-analysis-patterns-part-22/

R473 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 25) [Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2007/09/10/crash-dump-analysis-patterns-part-25/

R474 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 25d) [Stack Trace (database)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/21/crash-dump-analysis-patterns-part-25d/

R475 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 26b) [Virtualized Process (ARM64EC and CHPE)].*
https://www.dumpanalysis.org/blog/index.php/2025/11/16/crash-dump-analysis-patterns-part-26b/

R476 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 27d) [Stack Trace Collection (I/O requests)].*
https://www.dumpanalysis.org/blog/index.php/2012/01/11/crash-dump-analysis-patterns-part-27d/

R477 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 103) [Coupled Processes (semantics)].* https://www.dumpanalysis.org/blog/index.php/2010/08/04/crash-dump-analysis-patterns-part-103/

R478 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 29d) [High Contention (.NET CLR monitors)].* https://www.dumpanalysis.org/blog/index.php/2012/09/24/crash-dump-analysis-patterns-part-29d/

R479 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 31a) [Passive System Thread (kernel space)].* https://www.dumpanalysis.org/blog/index.php/2007/11/20/crash-dump-analysis-patterns-part-31a/

R480 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 38b) [Hooked Functions (kernel space)].* https://www.dumpanalysis.org/blog/index.php/2010/05/07/crash-dump-analysis-patterns-part-38b/

R481 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 41a) [Manual Dump (kernel)].* https://www.dumpanalysis.org/blog/index.php/2007/12/12/crash-dump-analysis-patterns-part-41a/

R482 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 42d) [Wait Chain (thread objects)].* https://www.dumpanalysis.org/blog/index.php/2008/11/07/crash-dump-analysis-patterns-part-42d/

R483 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 42g) [Wait Chain (RPC)].* https://www.dumpanalysis.org/blog/index.php/2010/09/14/crash-dump-analysis-patterns-part-42g/

R484 Dmitry Vostokov. 2011. *Crash Dump Analysis (Part 42j) [Wait Chain (mutex objects)].* https://www.dumpanalysis.org/blog/index.php/2011/10/13/crash-dump-analysis-part-42j/

R485 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 210) [Wait Chain (RTL_RESOURCE)].* https://www.dumpanalysis.org/blog/index.php/2014/09/08/crash-dump-analysis-patterns-part-210/

R486 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 42n) [Wait Chain (C++11, condition variable)].* https://www.dumpanalysis.org/blog/index.php/2016/09/23/crash-dump-analysis-patterns-part-42n/

R487 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 53b) [Blocked Thread (hardware)].* https://www.dumpanalysis.org/blog/index.php/2010/07/14/crash-dump-analysis-patterns-part-53b/

R488 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 55) [Wild Pointer].* https://www.dumpanalysis.org/blog/index.php/2008/03/11/crash-dump-analysis-patterns-part-55/

R489 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 58a) [Handle Limit (GDI, kernel space)].* https://www.dumpanalysis.org/blog/index.php/2008/04/09/crash-dump-analysis-patterns-part-58a/

R490 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 59b) [Missing Component (static linking, user mode)].* https://www.dumpanalysis.org/blog/index.php/2008/06/12/crash-dump-analysis-patterns-part-59b/

R491 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 10a) [Optimized VM Layout].* https://www.dumpanalysis.org/blog/index.php/2008/05/07/crash-dump-analysis-patterns-part-10a/

R492 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 62) [Overaged System].* https://www.dumpanalysis.org/blog/index.php/2008/05/28/crash-dump-analysis-patterns-part-62/

R493 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 64) [Duplicated Module].* https://www.dumpanalysis.org/blog/index.php/2008/06/19/crash-dump-analysis-patterns-part-64/

R494 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 67) [Nested Exceptions (unmanaged code)].* https://www.dumpanalysis.org/blog/index.php/2008/06/24/crash-dump-analysis-patterns-part-67/

R495 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 69) [Self-Diagnosis (user mode)].*
https://www.dumpanalysis.org/blog/index.php/2008/07/09/crash-dump-analysis-patterns-part-69/

R496 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 70) [Inline Function Optimization (unmanaged code)].*
https://www.dumpanalysis.org/blog/index.php/2008/07/10/crash-dump-analysis-patterns-part-70/

R497 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 73) [Young System].*
https://www.dumpanalysis.org/blog/index.php/2008/07/29/crash-dump-analysis-patterns-part-73/

R498 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 76) [Data Alignment (page boundary)].*
https://www.dumpanalysis.org/blog/index.php/2008/10/06/crash-dump-analysis-patterns-part-76/

R499 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 78b) [Divide by Zero (kernel mode)].*
https://www.dumpanalysis.org/blog/index.php/2010/11/02/crash-dump-analysis-patterns-part-78b/

R500 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 81) [Paged Out Data].*
https://www.dumpanalysis.org/blog/index.php/2009/02/19/crash-dump-analysis-patterns-part-81/

R501 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 84) [JIT Code (.NET)].*
https://www.dumpanalysis.org/blog/index.php/2009/05/15/crash-dump-analysis-patterns-part-84/

R502 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 85b) [Ubiquitous Component (kernel space)].*
https://www.dumpanalysis.org/blog/index.php/2013/05/31/crash-dump-analysis-patterns-part-85b/

R503 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 88) [Effect Component].*
https://www.dumpanalysis.org/blog/index.php/2009/10/23/crash-dump-analysis-patterns-part-88/

R504 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 91) [Random Object].*
https://www.dumpanalysis.org/blog/index.php/2009/11/12/crash-dump-analysis-patterns-part-91/

R505 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 94a) [Value Deviation (stack trace)].*
https://www.dumpanalysis.org/blog/index.php/2009/11/30/crash-dump-analysis-patterns-part-94a/

R506 Dmitry Vostokov. *Runtime Thread (Python, Linux).*
https://www.dumpanalysis.org/crash-dump-analysis-patterns-part-95b-linux

R507 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 98) [Hardware Activity].*
https://www.dumpanalysis.org/blog/index.php/2010/05/08/crash-dump-analysis-patterns-part-98/

R508 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 101) [Coupled Machines].*
https://www.dumpanalysis.org/blog/index.php/2010/07/16/crash-dump-analysis-patterns-part-101/

R509 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 106) [Distributed Spike].*
https://www.dumpanalysis.org/blog/index.php/2010/09/19/crash-dump-analysis-patterns-part-106/

R510 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 109) [Invalid Exception Information].*
https://www.dumpanalysis.org/blog/index.php/2010/10/08/crash-dump-analysis-patterns-part-109/

R511 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 112) [Problem Exception Handler].*
https://www.dumpanalysis.org/blog/index.php/2010/10/27/crash-dump-analysis-patterns-part-112/

R512 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 115a) [Blocked Queue (LPC/ALPC)].*
https://www.dumpanalysis.org/blog/index.php/2010/11/11/crash-dump-analysis-patterns-part-115a/

R513 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 117b) [Invalid Parameter (runtime function)].* https://www.dumpanalysis.org/blog/index.php/2017/05/14/crash-dump-analysis-patterns-part-117b/

R514 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 120) [Embedded Comment].* https://www.dumpanalysis.org/blog/index.php/2010/12/08/crash-dump-analysis-patterns-part-120/

R515 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 123) [Dual Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2010/12/21/crash-dump-analysis-patterns-part-123/

R516 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 126) [Livelock].* https://www.dumpanalysis.org/blog/index.php/2011/01/08/crash-dump-analysis-patterns-part-126/

R517 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 127c) [Technology-Specific Subtrace (JIT .NET code)].* https://www.dumpanalysis.org/blog/index.php/2011/10/10/crash-dump-analysis-patterns-part-127c/

R518 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 129) [Instrumentation Side Effect].* https://www.dumpanalysis.org/blog/index.php/2011/02/09/crash-dump-analysis-patterns-part-129/

R519 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 132) [Least Common Frame].* https://www.dumpanalysis.org/blog/index.php/2011/03/12/crash-dump-analysis-patterns-part-132/

R520 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 134b) [Data Correlation (CPU times)].* https://www.dumpanalysis.org/blog/index.php/2017/11/25/crash-dump-analysis-patterns-part-134b/

R521 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 137) [Cloud Environment].* https://www.dumpanalysis.org/blog/index.php/2011/06/12/crash-dump-analysis-patterns-part-137/

R522 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 141) [Thread Age].* https://www.dumpanalysis.org/blog/index.php/2011/06/23/crash-dump-analysis-patterns-part-141/

R523 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 144) [Quiet Dump].* https://www.dumpanalysis.org/blog/index.php/2011/06/25/crash-dump-analysis-patterns-part-144/

R524 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 147) [Activation Context].* https://www.dumpanalysis.org/blog/index.php/2011/08/30/crash-dump-analysis-patterns-part-147/

R525 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 150) [Caller-n-Callee].* https://www.dumpanalysis.org/blog/index.php/2011/10/10/crash-dump-analysis-patterns-part-150/

R526 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 152a) [Handled Exception (user space)].* https://www.dumpanalysis.org/blog/index.php/2011/10/17/crash-dump-analysis-patterns-part-152a/

R527 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 153) [Duplicate Extension].* https://www.dumpanalysis.org/blog/index.php/2011/10/21/crash-dump-analysis-patterns-part-153/

R528 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 156) [FPU Exception].* https://www.dumpanalysis.org/blog/index.php/2011/11/29/crash-dump-analysis-patterns-part-156/

R529 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 159) [Value References].* https://www.dumpanalysis.org/blog/index.php/2011/12/05/crash-dump-analysis-patterns-part-159/

R530 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 162) [Problem Module].*
https://www.dumpanalysis.org/blog/index.php/2011/12/14/crash-dump-analysis-patterns-part-162/

R531 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 165) [Unrecognizable Symbolic Information].*
https://www.dumpanalysis.org/blog/index.php/2011/12/26/crash-dump-analysis-patterns-part-165/

R532 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 168) [Late Crash Dump].*
https://www.dumpanalysis.org/blog/index.php/2012/03/20/crash-dump-analysis-patterns-part-168/

R533 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 171) [Punctuated Memory Leak].*
https://www.dumpanalysis.org/blog/index.php/2012/04/28/crash-dump-analysis-patterns-part-171/

R534 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 174) [Activity Resonance].*
https://www.dumpanalysis.org/blog/index.php/2012/05/20/crash-dump-analysis-patterns-part-174/

R535 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 177) [Stack Trace Change].*
https://www.dumpanalysis.org/blog/index.php/2012/06/09/crash-dump-analysis-patterns-part-177/

R536 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 180, Mac OS X) [Paratext].*
https://www.dumpanalysis.org/blog/index.php/2012/07/28/crash-dump-analysis-patterns-part-180-mac-os-x/

R537 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 183) [First Fault Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2012/10/10/crash-dump-analysis-patterns-part-183/

R538 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 186) [Hidden Process].*
https://www.dumpanalysis.org/blog/index.php/2012/11/13/crash-dump-analysis-patterns-part-186/

R539 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 189) [Handle Leak].*
https://www.dumpanalysis.org/blog/index.php/2012/12/23/crash-dump-analysis-patterns-part-189/

R540 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 191) [Deviant Token].*
https://www.dumpanalysis.org/blog/index.php/2012/12/31/crash-dump-analysis-patterns-part-191/

R541 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 194) [Debugger Omission].*
https://www.dumpanalysis.org/blog/index.php/2013/01/18/crash-dump-analysis-patterns-part-194/

R542 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 197) [Injected Symbols].*
https://www.dumpanalysis.org/blog/index.php/2013/02/27/crash-dump-analysis-patterns-part-197/

R543 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 200) [Module Product Process].*
https://www.dumpanalysis.org/blog/index.php/2013/08/16/crash-dump-analysis-patterns-part-200/

R544 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 203) [Shared Structure].*
https://www.dumpanalysis.org/blog/index.php/2013/12/07/crash-dump-analysis-patterns-part-203/

R545 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 206) [Screwbolt Wait Chain].*
https://www.dumpanalysis.org/blog/index.php/2014/05/10/crash-dump-analysis-patterns-part-206/

R546 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 209) [Tampered Dump].*
https://www.dumpanalysis.org/blog/index.php/2014/09/06/crash-dump-analysis-patterns-part-209/

R547 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 213) [Rough Stack Trace (unmanaged space)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/07/crash-dump-analysis-patterns-part-213/

R548 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 215) [Ghost Thread].* https://www.dumpanalysis.org/blog/index.php/2014/11/16/crash-dump-analysis-patterns-part-215/

R549 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 218) [Reference Leak].* https://www.dumpanalysis.org/blog/index.php/2014/12/31/crash-dump-analysis-patterns-part-218/

R550 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 221) [Corrupt Structure].* https://www.dumpanalysis.org/blog/index.php/2015/02/21/crash-dump-analysis-patterns-part-221/

R551 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 224) [Variable Subtrace].* https://www.dumpanalysis.org/blog/index.php/2015/04/25/crash-dump-analysis-patterns-part-224/

R552 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 227) [Distributed Exception (managed code)].* https://www.dumpanalysis.org/blog/index.php/2015/05/17/crash-dump-analysis-patterns-part-227/

R553 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 230) [Hidden Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2015/10/14/crash-dump-analysis-patterns-part-230/

R554 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 233) [Clone Dump].* https://www.dumpanalysis.org/blog/index.php/2015/11/18/crash-dump-analysis-patterns-part-233/

R555 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 236) [Diachronic Module].* https://www.dumpanalysis.org/blog/index.php/2016/02/13/crash-dump-analysis-patterns-part-236/

R556 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 240) [Window Hint].* https://www.dumpanalysis.org/blog/index.php/2016/05/29/crash-dump-analysis-patterns-part-240/

R557 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 243) [Relative Memory Leak].* https://www.dumpanalysis.org/blog/index.php/2016/09/15/crash-dump-analysis-patterns-part-243/

R558 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 246) [Foreign Module Frame].* https://www.dumpanalysis.org/blog/index.php/2016/12/07/crash-dump-analysis-patterns-part-246/

R559 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 249) [Memory Fibration].* https://www.dumpanalysis.org/blog/index.php/2017/03/22/crash-dump-analysis-patterns-part-249/

R560 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 252) [Stack Trace Motif].* https://www.dumpanalysis.org/blog/index.php/2017/05/21/crash-dump-analysis-patterns-part-252/

R561 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 255) [Hyperdump].* https://www.dumpanalysis.org/blog/index.php/2018/03/11/crash-dump-analysis-patterns-part-255/

R562 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 258) [Active Space].* https://www.dumpanalysis.org/blog/index.php/2019/02/03/crash-dump-analysis-patterns-part-258/

R563 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 261) [Region Clusters].* https://www.dumpanalysis.org/blog/index.php/2019/10/13/crash-dump-analysis-patterns-part-261/

R564 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 264) [Interrupt Stack].* https://www.dumpanalysis.org/blog/index.php/2020/02/27/crash-dump-analysis-patterns-part-264/

R565 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 268) [Pointer Cone].* https://www.dumpanalysis.org/blog/index.php/2020/06/14/crash-dump-analysis-patterns-part-268/

R566 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 271) [False Frame].* https://www.dumpanalysis.org/blog/index.php/2020/08/29/crash-dump-analysis-patterns-part-271/

R567 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 274) [COM Exception].* https://www.dumpanalysis.org/blog/index.php/2021/01/31/crash-dump-analysis-patterns-part-274/

R568 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 277) [Shared Thread].* https://www.dumpanalysis.org/blog/index.php/2021/09/27/crash-dump-analysis-patterns-part-277/

R569 Dmitry Vostokov. 2022. *Crash Dump Analysis Patterns (Part 280) [Black Box].* https://www.dumpanalysis.org/blog/index.php/2022/03/09/crash-dump-analysis-patterns-part-280/

R570 Dmitry Vostokov. 2023. *Crash Dump Analysis Patterns (Part 283) [Shared Page].* https://www.dumpanalysis.org/blog/index.php/2023/04/08/crash-dump-analysis-patterns-part-283/

R571 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 286) [Address Representations].* https://www.dumpanalysis.org/blog/index.php/2024/01/28/crash-dump-analysis-patterns-part-286/

R572 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 289) [Past Process].* https://www.dumpanalysis.org/blog/index.php/2024/03/12/crash-dump-analysis-patterns-part-289/

R573 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 292) [Disassembly Summary].* https://www.dumpanalysis.org/blog/index.php/2024/03/17/crash-dump-analysis-patterns-part-292/

R574 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 295) [Region Spectrum].* https://www.dumpanalysis.org/blog/index.php/2024/03/30/crash-dump-analysis-patterns-part-295/

R575 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 299) [Interrupt Stack Collection].* https://www.dumpanalysis.org/blog/index.php/2025/06/03/crash-dump-analysis-patterns-part-299/

R576 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 302) [False Local Address].* https://www.dumpanalysis.org/blog/index.php/2025/11/17/crash-dump-analysis-patterns-part-302/

R577 Dmitry Vostokov. 2026. *Crash Dump Analysis Patterns (Part 305) [ISA-Specific Code].* https://www.dumpanalysis.org/blog/index.php/2026/01/09/crash-dump-analysis-patterns-part-305/

R578 Dmitry Vostokov. 2026. *Crash Dump Analysis Patterns (Part 278, Linux) [Spiking Interrupts (Linux)].* https://www.dumpanalysis.org/blog/index.php/2026/05/30/crash-dump-analysis-patterns-part-278-linux/

Memory analysis catalog additions: second group

R579 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 1b) [Multiple Exceptions (kernel mode)].* https://www.dumpanalysis.org/blog/index.php/2008/10/15/crash-dump-analysis-patterns-part-1b/

R580 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 2) [Dynamic Memory Corruption (process heap)].* https://www.dumpanalysis.org/blog/index.php/2006/10/31/crash-dump-analysis-patterns-part-2/

R581 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 4b) [Lateral Damage (CPU mode)].* https://www.dumpanalysis.org/blog/index.php/2020/07/10/crash-dump-analysis-patterns-part-4b/

R582 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 265) [Invalid Pointer (objects)].* https://www.dumpanalysis.org/blog/index.php/2020/04/30/crash-dump-analysis-patterns-part-265/

R583 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 8b) [Hidden Exception (kernel space)].* https://www.dumpanalysis.org/blog/index.php/2012/08/30/crash-dump-analysis-patterns-part-8b/

R584 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 9b) [Deadlock (executive resources)].* https://www.dumpanalysis.org/blog/index.php/2007/07/03/crash-dump-analysis-patterns-part-9b/

R585 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 9e) [Deadlock (mixed objects, kernel space)].*
https://www.dumpanalysis.org/blog/index.php/2008/10/25/crash-dump-analysis-patterns-part-9e/

R586 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 9h) [Deadlock (.NET Finalizer)].*
https://www.dumpanalysis.org/blog/index.php/2017/05/13/crash-dump-analysis-patterns-part-9h/

R587 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 13b) [Insufficient Memory (handle leak)].*
https://www.dumpanalysis.org/blog/index.php/2007/07/15/crash-dump-analysis-patterns-part-13b/

R588 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 13e) [Insufficient Memory (module fragmentation)].*
https://www.dumpanalysis.org/blog/index.php/2008/03/18/crash-dump-analysis-patterns-part-13e/

R589 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 13h) [Insufficient Memory (reserved virtual memory)].*
https://www.dumpanalysis.org/blog/index.php/2012/04/09/crash-dump-analysis-patterns-part-13h/

R590 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 13k) [Insufficient Memory (region)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/27/crash-dump-analysis-patterns-part-13k/

R591 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 16b) [Stack Overflow (user mode)].*
https://www.dumpanalysis.org/blog/index.php/2008/06/10/crash-dump-analysis-patterns-part-16b/

R592 Dmitry Vostokov. 2022. *Crash Dump Analysis Patterns (Part 16e) [Stack Overflow (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2022/02/04/crash-dump-analysis-patterns-part-16e/

R593 Dmitry Vostokov. *Managed Code Exception (Python) and Managed Stack Trace (Python).*
https://www.dumpanalysis.org/crash-dump-analysis-patterns-part-17c-139c

R594 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 20b) [Memory Leak (.NET heap)].*
https://www.dumpanalysis.org/blog/index.php/2007/08/19/crash-dump-analysis-patterns-part-20b/

R595 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 20e) [Memory Leak (regions)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/30/crash-dump-analysis-patterns-part-20e/

R596 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 23a) [Double Free (process heap)].*
https://www.dumpanalysis.org/blog/index.php/2007/08/19/crash-dump-analysis-patterns-part-23a/

R597 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 25b) [Stack Trace (I/O request)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/12/crash-dump-analysis-patterns-part-25b/

R598 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 25e) [Stack Trace (I/O devices)].*
https://www.dumpanalysis.org/blog/index.php/2017/05/06/crash-dump-analysis-patterns-part-25e/

R599 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 27b) [Stack Trace Collection (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2011/10/24/crash-dump-analysis-patterns-part-27b/

R600 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 27e) [Stack Trace Collection (CPUs)].*
https://www.dumpanalysis.org/blog/index.php/2015/08/24/crash-dump-analysis-patterns-part-27e/

R601 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 29) [High Contention (executive resources)].*
https://www.dumpanalysis.org/blog/index.php/2007/10/08/crash-dump-analysis-patterns-part-29/

R602 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 29e) [High Contention (.NET heap)].*
https://www.dumpanalysis.org/blog/index.php/2017/05/08/crash-dump-analysis-patterns-part-29e/

R603 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 35b) [Object Distribution Anomaly (.NET heap)].* https://www.dumpanalysis.org/blog/index.php/2015/09/12/crash-dump-analysis-patterns-part-35b/

R604 Dmitry Vostokov. 2008. *Hooked Modules (Crash Dump Analysis Patterns, Part 38c).* https://www.dumpanalysis.org/blog/index.php/2008/09/19/hooked-modules/

R605 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 42b) [Wait Chain (critical sections)].* https://www.dumpanalysis.org/blog/index.php/2007/12/19/crash-dump-analysis-patterns-part-42b/

R606 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 42e) [Wait Chain (LPC/ALPC)].* https://www.dumpanalysis.org/blog/index.php/2008/12/17/crash-dump-analysis-patterns-part-42e/

R607 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 42h) [Wait Chain (window messaging)].* https://www.dumpanalysis.org/blog/index.php/2010/12/16/crash-dump-analysis-patterns-part-42h/

R608 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 42k) [Wait Chain (pushlocks)].* https://www.dumpanalysis.org/blog/index.php/2013/01/08/crash-dump-analysis-patterns-part-42k/

R609 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 42m) [Wait Chain (modules)].* https://www.dumpanalysis.org/blog/index.php/2014/10/23/crash-dump-analysis-patterns-part-42m/

R610 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 42o) [Wait Chain (SRW lock)].* https://www.dumpanalysis.org/blog/index.php/2017/05/20/crash-dump-analysis-patterns-part-42o/

R611 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 53c) [Blocked Thread (timeout)].* https://www.dumpanalysis.org/blog/index.php/2012/04/28/crash-dump-analysis-patterns-part-53c/

R612 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 56) [Wild Code].* https://www.dumpanalysis.org/blog/index.php/2008/03/27/crash-dump-analysis-patterns-part-56/

R613 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 58b) [Handle Limit (GDI, user space)].* https://www.dumpanalysis.org/blog/index.php/2016/06/05/crash-dump-analysis-patterns-part-58b/

R614 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 60c) [Execution Residue (unmanaged space, kernel)].* https://www.dumpanalysis.org/blog/index.php/2021/10/03/crash-dump-analysis-patterns-part-60c/

R615 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 61) [Invalid Handle (general)].* https://www.dumpanalysis.org/blog/index.php/2008/05/20/crash-dump-analysis-patterns-part-61/

R616 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 63) [Thread Starvation (realtime priority)].* https://www.dumpanalysis.org/blog/index.php/2008/06/06/crash-dump-analysis-patterns-part-63/

R617 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 65b) [Not My Version (hardware)].* https://www.dumpanalysis.org/blog/index.php/2009/11/16/crash-dump-analysis-patterns-part-65b/

R618 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 67b) [Nested Exceptions (managed code)].* https://www.dumpanalysis.org/blog/index.php/2008/06/25/crash-dump-analysis-patterns-part-67b/

R619 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 69b) [Self-Diagnosis (kernel mode)].* https://www.dumpanalysis.org/blog/index.php/2011/04/26/crash-dump-analysis-patterns-part-69b/

R620 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 70b) [Inline Function Optimization (managed code)].* https://www.dumpanalysis.org/blog/index.php/2011/10/10/crash-dump-analysis-patterns-part-70b/

R621 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 74) [Last Error Collection].*
https://www.dumpanalysis.org/blog/index.php/2008/08/05/crash-dump-analysis-patterns-part-74/

R622 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 77) [C++ Exception].*
https://www.dumpanalysis.org/blog/index.php/2008/10/21/crash-dump-analysis-patterns-part-77/

R623 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 79) [Swarm of Shared Locks].*
https://www.dumpanalysis.org/blog/index.php/2009/02/09/crash-dump-analysis-patterns-part-79/

R624 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 82) [Semantic Split].*
https://www.dumpanalysis.org/blog/index.php/2009/03/09/crash-dump-analysis-patterns-part-82/

R625 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 84b) [JIT Code (Java)].*
https://www.dumpanalysis.org/blog/index.php/2016/09/20/crash-dump-analysis-patterns-part-84b/

R626 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 86) [Nested Offender].*
https://www.dumpanalysis.org/blog/index.php/2009/06/24/crash-dump-analysis-patterns-part-86/

R627 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 89) [Well-Tested Function].*
https://www.dumpanalysis.org/blog/index.php/2009/10/23/crash-dump-analysis-patterns-part-89/

R628 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 92) [Missing Process].*
https://www.dumpanalysis.org/blog/index.php/2009/11/24/crash-dump-analysis-patterns-part-92/

R629 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 94b) [Value Deviation (structure field)].*
https://www.dumpanalysis.org/blog/index.php/2017/05/02/crash-dump-analysis-patterns-part-94b/

R630 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 96) [Coincidental Frames].*
https://www.dumpanalysis.org/blog/index.php/2010/02/21/crash-dump-analysis-patterns-part-96/

R631 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 99a) [Incorrect Symbolic Information].*
https://www.dumpanalysis.org/blog/index.php/2010/05/29/crash-dump-analysis-patterns-part-99a/

R632 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 104) [Abridged Dump].*
https://www.dumpanalysis.org/blog/index.php/2010/08/04/crash-dump-analysis-patterns-part-104/

R633 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 107) [Instrumentation Information].*
https://www.dumpanalysis.org/blog/index.php/2010/09/27/crash-dump-analysis-patterns-part-107/

R634 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 110) [Shared Buffer Overwrite].*
https://www.dumpanalysis.org/blog/index.php/2010/10/18/crash-dump-analysis-patterns-part-110/

R635 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 113) [Same Vendor].*
https://www.dumpanalysis.org/blog/index.php/2010/10/29/crash-dump-analysis-patterns-part-113/

R636 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 116) [Fat Process Dump].*
https://www.dumpanalysis.org/blog/index.php/2010/11/20/crash-dump-analysis-patterns-part-116/

R637 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 118) [String Parameter].*
https://www.dumpanalysis.org/blog/index.php/2010/12/02/crash-dump-analysis-patterns-part-118/

R638 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 121) [Hooking Level].*
https://www.dumpanalysis.org/blog/index.php/2010/12/11/crash-dump-analysis-patterns-part-121/

R639 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 124) [Environment Hint].*
https://www.dumpanalysis.org/blog/index.php/2010/12/24/crash-dump-analysis-patterns-part-124/

R640 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 127) [Technology-Specific Subtrace (COM interface invocation)].* https://www.dumpanalysis.org/blog/index.php/2011/01/15/crash-dump-analysis-patterns-part-127/

R641 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 127d) [Technology-Specific Subtrace (COM client call)].* https://www.dumpanalysis.org/blog/index.php/2015/05/10/crash-dump-analysis-patterns-part-127d/

R642 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 130a) [Semantic Structure (PID.TID)].* https://www.dumpanalysis.org/blog/index.php/2011/02/19/crash-dump-analysis-patterns-part-130a/

R643 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 133) [Truncated Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2011/03/20/crash-dump-analysis-patterns-part-133/

R644 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 135) [Module Hint].* https://www.dumpanalysis.org/blog/index.php/2011/04/24/crash-dump-analysis-patterns-part-135/

R645 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 138) [No Data Types].* https://www.dumpanalysis.org/blog/index.php/2011/06/12/crash-dump-analysis-patterns-part-138/

R646 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 142) [Unsynchronized Dumps].* https://www.dumpanalysis.org/blog/index.php/2011/06/24/crash-dump-analysis-patterns-part-142/

R647 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 145) [Blocking File].* https://www.dumpanalysis.org/blog/index.php/2011/06/25/crash-dump-analysis-patterns-part-145/

R648 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 148) [Stack Trace Set].* https://www.dumpanalysis.org/blog/index.php/2011/09/07/crash-dump-analysis-patterns-part-148/

R649 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 151) [Annotated Disassembly (JIT .NET code)].* https://www.dumpanalysis.org/blog/index.php/2011/10/13/crash-dump-analysis-patterns-part-151/

R650 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 152b) [Handled Exception (.NET CLR)].* https://www.dumpanalysis.org/blog/index.php/2011/10/17/crash-dump-analysis-patterns-part-152b/

R651 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 154a) [Special Thread (.NET CLR)].* https://www.dumpanalysis.org/blog/index.php/2011/10/27/crash-dump-analysis-patterns-part-154a/

R652 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 157) [Module Variable].* https://www.dumpanalysis.org/blog/index.php/2011/12/03/crash-dump-analysis-patterns-part-157/

R653 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 160) [Debugger Bug].* https://www.dumpanalysis.org/blog/index.php/2011/12/11/crash-dump-analysis-patterns-part-160/

R654 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 163) [Disconnected Network Adapter].* https://www.dumpanalysis.org/blog/index.php/2011/12/25/crash-dump-analysis-patterns-part-163/

R655 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 166) [Translated Exception].* https://www.dumpanalysis.org/blog/index.php/2012/02/12/crash-dump-analysis-patterns-part-166/

R656 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 169) [Blocked DPC].* https://www.dumpanalysis.org/blog/index.php/2012/03/22/crash-dump-analysis-patterns-part-169/

R657 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 172) [No Current Thread].* https://www.dumpanalysis.org/blog/index.php/2012/05/19/crash-dump-analysis-patterns-part-172/

R658 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 175) [Stored Exception].*
https://www.dumpanalysis.org/blog/index.php/2012/05/23/crash-dump-analysis-patterns-part-175/

R659 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 178) [Unloaded Module].*
https://www.dumpanalysis.org/blog/index.php/2012/06/27/crash-dump-analysis-patterns-part-178/

R660 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 181) [Incomplete Session].*
https://www.dumpanalysis.org/blog/index.php/2012/10/01/crash-dump-analysis-patterns-part-181/

R661 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 184) [Frozen Process].*
https://www.dumpanalysis.org/blog/index.php/2012/10/31/crash-dump-analysis-patterns-part-184/

R662 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 187, Mac OS X) [Active Thread (Mac OS X)].*
https://www.dumpanalysis.org/blog/index.php/2012/11/17/crash-dump-analysis-patterns-part-187-mac-os-x/

R663 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 190) [Module Collection].*
https://www.dumpanalysis.org/blog/index.php/2012/12/24/crash-dump-analysis-patterns-part-190/

R664 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 192) [Step Dumps].*
https://www.dumpanalysis.org/blog/index.php/2013/01/04/crash-dump-analysis-patterns-part-192/

R665 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 195) [Glued Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2013/02/26/crash-dump-analysis-patterns-part-195/

R666 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 198) [Distributed Wait Chain].*
https://www.dumpanalysis.org/blog/index.php/2013/05/26/crash-dump-analysis-patterns-part-198/

R667 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 201) [Crash Signature Invariant].*
https://www.dumpanalysis.org/blog/index.php/2013/11/04/crash-dump-analysis-patterns-part-201/

R668 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 204) [Thread Cluster].*
https://www.dumpanalysis.org/blog/index.php/2014/04/18/crash-dump-analysis-patterns-part-204/

R669 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 207) [Design Value].*
https://www.dumpanalysis.org/blog/index.php/2014/06/21/crash-dump-analysis-patterns-part-207/

R670 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 211) [Memory Fluctuation (process heap)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/04/crash-dump-analysis-patterns-part-211/

R671 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 213b) [Rough Stack Trace (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2021/04/09/crash-dump-analysis-patterns-part-213b/

R672 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 216) [Dry Weight].*
https://www.dumpanalysis.org/blog/index.php/2014/11/19/crash-dump-analysis-patterns-part-216/

R673 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 219) [Origin Module].*
https://www.dumpanalysis.org/blog/index.php/2015/02/07/crash-dump-analysis-patterns-part-219/

R674 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 222) [Software Exception].*
https://www.dumpanalysis.org/blog/index.php/2015/02/28/crash-dump-analysis-patterns-part-222/

R675 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 225) [User Space Evidence].*
https://www.dumpanalysis.org/blog/index.php/2015/05/09/crash-dump-analysis-patterns-part-225/

R676 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 228) [Thread Poset].* https://www.dumpanalysis.org/blog/index.php/2015/06/28/crash-dump-analysis-patterns-part-228/

R677 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 231) [Evental Dumps].* https://www.dumpanalysis.org/blog/index.php/2015/10/26/crash-dump-analysis-patterns-part-231/

R678 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 234) [Parameter Flow].* https://www.dumpanalysis.org/blog/index.php/2015/11/30/crash-dump-analysis-patterns-part-234/

R679 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 237) [Constant Subtrace].* https://www.dumpanalysis.org/blog/index.php/2016/03/17/crash-dump-analysis-patterns-part-237/

R680 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 241) [Place Trace].* https://www.dumpanalysis.org/blog/index.php/2016/05/29/crash-dump-analysis-patterns-part-241/

R681 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 244) [Quotient Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2016/10/10/crash-dump-analysis-patterns-part-244/

R682 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 247) [Unified Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2017/02/05/crash-dump-analysis-patterns-part-247/

R683 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 250) [Aggregated Frames].* https://www.dumpanalysis.org/blog/index.php/2017/04/03/crash-dump-analysis-patterns-part-250/

R684 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 253) [System Call].* https://www.dumpanalysis.org/blog/index.php/2017/11/25/crash-dump-analysis-patterns-part-253/

R685 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 256) [Disassembly Ambiguity].* https://www.dumpanalysis.org/blog/index.php/2018/08/04/crash-dump-analysis-patterns-part-256/

R686 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 259) [Subsystem Modules].* https://www.dumpanalysis.org/blog/index.php/2019/09/01/crash-dump-analysis-patterns-part-259/

R687 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 262) [Source Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2019/11/30/crash-dump-analysis-patterns-part-262/

R688 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 266) [False Memory].* https://www.dumpanalysis.org/blog/index.php/2020/05/02/crash-dump-analysis-patterns-part-266/

R689 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 269) [Context Pointer].* https://www.dumpanalysis.org/blog/index.php/2020/06/15/crash-dump-analysis-patterns-part-269/

R690 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 272) [Procedure Call Chain].* https://www.dumpanalysis.org/blog/index.php/2020/09/18/crash-dump-analysis-patterns-part-272/

R691 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 275) [Structure Sheaf].* https://www.dumpanalysis.org/blog/index.php/2021/03/03/crash-dump-analysis-patterns-part-275/

R692 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 278) [Spiking Interrupts].* https://www.dumpanalysis.org/blog/index.php/2021/11/22/crash-dump-analysis-patterns-part-278/

R693 Dmitry Vostokov. 2022. *Crash Dump Analysis Patterns (Part 281) [Rough Stack Trace Collection (unmanaged space)].* https://www.dumpanalysis.org/blog/index.php/2022/06/08/crash-dump-analysis-patterns-part-281/

R694 Dmitry Vostokov. 2023. *Crash Dump Analysis Patterns (Part 284) [Exception Collection].*
https://www.dumpanalysis.org/blog/index.php/2023/07/01/crash-dump-analysis-patterns-part-284/

R695 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 287) [Near Exception].*
https://www.dumpanalysis.org/blog/index.php/2024/02/11/crash-dump-analysis-patterns-part-287/

R696 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 290) [Foreign Stack].*
https://www.dumpanalysis.org/blog/index.php/2024/03/13/crash-dump-analysis-patterns-part-290/

R697 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 293) [Region Summary].*
https://www.dumpanalysis.org/blog/index.php/2024/03/18/crash-dump-analysis-patterns-part-293/

R698 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 296) [Normalized Region].*
https://www.dumpanalysis.org/blog/index.php/2024/03/31/crash-dump-analysis-patterns-part-296/

R699 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 300) [DPC Stack Collection].*
https://www.dumpanalysis.org/blog/index.php/2025/09/20/crash-dump-analysis-patterns-part-300/

R700 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 303) [Encoded Pointer].*
https://www.dumpanalysis.org/blog/index.php/2025/11/30/crash-dump-analysis-patterns-part-303/

Memory analysis catalog additions: third group

R701 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 1c) [Multiple Exceptions (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2011/06/28/crash-dump-analysis-patterns-part-1c/

R702 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 2b) [Dynamic Memory Corruption (kernel pool)].*
https://www.dumpanalysis.org/blog/index.php/2008/03/13/crash-dump-analysis-patterns-part-2b/

R703 Dmitry Vostokov. 2006. *Crash Dump Analysis Patterns (Part 5) [Optimized Code (function parameter reuse)].*
https://www.dumpanalysis.org/blog/index.php/2006/12/15/crash-dump-analysis-patterns-part-5/

R704 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 6a) [NULL Pointer (code)].*
https://www.dumpanalysis.org/blog/index.php/2008/04/28/crash-dump-analysis-patterns-part-6a/

R705 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 8c) [Hidden Exception (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2018/08/11/crash-dump-analysis-patterns-part-8c/

R706 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 9c) [Deadlock (mixed objects, user space)].*
https://www.dumpanalysis.org/blog/index.php/2007/07/28/crash-dump-analysis-patterns-part-9c/

R707 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 9f) [Deadlock (self)].*
https://www.dumpanalysis.org/blog/index.php/2010/10/27/crash-dump-analysis-patterns-part-9f/

R708 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 5b) [OMAP Code Optimization].*
https://www.dumpanalysis.org/blog/index.php/2007/04/20/crash-dump-analysis-patterns-part-5b/

R709 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 13c) [Insufficient Memory (kernel pool)].*
https://www.dumpanalysis.org/blog/index.php/2007/11/02/crash-dump-analysis-patterns-part-13c/

R710 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 13f) [Insufficient Memory (physical memory)].*
https://www.dumpanalysis.org/blog/index.php/2009/01/05/crash-dump-analysis-patterns-part-13f/

R711 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 13i) [Insufficient Memory (session pool)].*
https://www.dumpanalysis.org/blog/index.php/2013/01/05/crash-dump-analysis-patterns-part-13i/

R712 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 13l) [Insufficient Memory (stack)].*
https://www.dumpanalysis.org/blog/index.php/2018/10/06/crash-dump-analysis-patterns-part-13l/

R713 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 16c) [Stack Overflow (software implementation)].*
https://www.dumpanalysis.org/blog/index.php/2011/03/01/crash-dump-analysis-patterns-part-16c/

R714 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 17) [Managed Code Exception].*
https://www.dumpanalysis.org/blog/index.php/2007/07/20/crash-dump-analysis-patterns-part-17/

R715 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 19b) [Waiting Thread Time (user dumps)].*
https://www.dumpanalysis.org/blog/index.php/2008/07/10/crash-dump-analysis-patterns-part-19b/

R716 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 20c) [Memory Leak (page tables)].*
https://www.dumpanalysis.org/blog/index.php/2011/12/26/crash-dump-analysis-patterns-part-20c/

R717 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 298) [Missing Thread (kernel space)].*
https://www.dumpanalysis.org/blog/index.php/2024/09/14/crash-dump-analysis-patterns-part-298/

R718 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 23b) [Double Free (kernel pool)].*
https://www.dumpanalysis.org/blog/index.php/2007/08/25/crash-dump-analysis-patterns-part-23b/

R719 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 25c) [Stack Trace (file system filters)].*
https://www.dumpanalysis.org/blog/index.php/2014/10/12/crash-dump-analysis-patterns-part-25c/

R720 Dmitry Vostokov. 2007. *Crash Dump Analysis Patterns (Part 26) [Virtualized Process (WOW64)].*
https://www.dumpanalysis.org/blog/index.php/2007/09/11/crash-dump-analysis-patterns-part-26/

R721 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 27c) [Stack Trace Collection (predicate)].*
https://www.dumpanalysis.org/blog/index.php/2011/12/03/crash-dump-analysis-patterns-part-27c/

R722 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 28b) [Coupled Processes (weak)].*
https://www.dumpanalysis.org/blog/index.php/2010/04/07/crash-dump-analysis-patterns-part-28b/

R723 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 29c) [High Contention (processors)].*
https://www.dumpanalysis.org/blog/index.php/2010/07/21/crash-dump-analysis-patterns-part-29c/

R724 Dmitry Vostokov. 2022. *Crash Dump Analysis Patterns (Part 29f) [High Contention (sockets)].*
https://www.dumpanalysis.org/blog/index.php/2022/11/27/crash-dump-analysis-patterns-part-29f/

R725 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 36b) [Local Buffer Overflow (kernel space)].*
https://www.dumpanalysis.org/blog/index.php/2021/10/05/crash-dump-analysis-patterns-part-36b/

R726 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 39b) [Custom Exception Handler (kernel space)].*
https://www.dumpanalysis.org/blog/index.php/2011/04/05/crash-dump-analysis-patterns-part-39b/

R727 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 42c) [Wait Chain (executive resources)].*
https://www.dumpanalysis.org/blog/index.php/2008/01/22/crash-dump-analysis-patterns-part-42c/

R728 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 42f) [Wait Chain (process objects)].*
https://www.dumpanalysis.org/blog/index.php/2010/02/16/crash-dump-analysis-patterns-part-42f/

R729 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 42i) [Wait Chain (named pipes)].*
https://www.dumpanalysis.org/blog/index.php/2011/01/03/crash-dump-analysis-patterns-part-42i/

R730 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 42l) [Wait Chain (CLR monitors)].*
https://www.dumpanalysis.org/blog/index.php/2014/04/09/crash-dump-analysis-patterns-part-42l/

R731 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 238) [Wait Chain (nonstandard synchronization)].*
https://www.dumpanalysis.org/blog/index.php/2016/03/17/crash-dump-analysis-patterns-part-238/

R732 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 51) [Message Box].*
https://www.dumpanalysis.org/blog/index.php/2008/02/19/crash-dump-analysis-patterns-part-51/

R733 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 54) [Zombie Processes].*
https://www.dumpanalysis.org/blog/index.php/2008/02/28/crash-dump-analysis-patterns-part-54/

R734 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 57) [Hardware Error].*
https://www.dumpanalysis.org/blog/index.php/2008/04/03/crash-dump-analysis-patterns-part-57/

R735 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 59) [Missing Component (general)].*
https://www.dumpanalysis.org/blog/index.php/2008/04/22/crash-dump-analysis-patterns-part-59/

R736 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 60b) [Execution Residue (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2011/10/17/crash-dump-analysis-patterns-part-60b/

R737 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 61b) [Invalid Handle (managed space)].*
https://www.dumpanalysis.org/blog/index.php/2014/11/15/crash-dump-analysis-patterns-part-61b/

R738 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 102) [Thread Starvation (normal priority)].*
https://www.dumpanalysis.org/blog/index.php/2010/07/29/crash-dump-analysis-patterns-part-102/

R739 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 66) [Data Contents Locality; Code Path Locality].*
https://www.dumpanalysis.org/blog/index.php/2008/06/20/crash-dump-analysis-patterns-part-66/

R740 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 68) [Affine Thread].*
https://www.dumpanalysis.org/blog/index.php/2008/06/27/crash-dump-analysis-patterns-part-68/

R741 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 69c) [Self-Diagnosis (registry)].*
https://www.dumpanalysis.org/blog/index.php/2011/12/04/crash-dump-analysis-patterns-part-69c/

R742 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 72) [Lost Opportunity].*
https://www.dumpanalysis.org/blog/index.php/2008/07/26/crash-dump-analysis-patterns-part-72/

R743 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 75) [Hidden Module].*
https://www.dumpanalysis.org/blog/index.php/2008/08/07/crash-dump-analysis-patterns-part-75/

R744 Dmitry Vostokov. 2008. *Crash Dump Analysis Patterns (Part 78a) [Divide by Zero (user mode)].*
https://www.dumpanalysis.org/blog/index.php/2008/12/01/crash-dump-analysis-patterns-part-78a/

R745 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 80) [Process Factory].*
https://www.dumpanalysis.org/blog/index.php/2009/02/13/crash-dump-analysis-patterns-part-80/

R746 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 83) [Pass Through Function].*
https://www.dumpanalysis.org/blog/index.php/2009/04/14/crash-dump-analysis-patterns-part-83/

R747 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 85) [Ubiquitous Component (user space)].*
https://www.dumpanalysis.org/blog/index.php/2009/06/23/crash-dump-analysis-patterns-part-85/

R748 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 87) [Virtualized System].*
https://www.dumpanalysis.org/blog/index.php/2009/07/10/crash-dump-analysis-patterns-part-87/

R749 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 90) [Mixed Exception].*
https://www.dumpanalysis.org/blog/index.php/2009/10/28/crash-dump-analysis-patterns-part-90/

R750 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 93) [Platform-Specific Debugger].*
https://www.dumpanalysis.org/blog/index.php/2009/11/24/crash-dump-analysis-patterns-part-93/

R751 Dmitry Vostokov. 2009. *Crash Dump Analysis Patterns (Part 95) [Runtime Thread (CLR)].*
https://www.dumpanalysis.org/blog/index.php/2009/12/07/crash-dump-analysis-patterns-part-95/

R752 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 97) [Fault Context].*
https://www.dumpanalysis.org/blog/index.php/2010/03/16/crash-dump-analysis-patterns-part-97/

R753 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 100) [Message Hooks].*
https://www.dumpanalysis.org/blog/index.php/2010/07/06/crash-dump-analysis-patterns-part-100/

R754 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 105) [Exception Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2010/08/05/crash-dump-analysis-patterns-part-105/

R755 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 108) [Template Module].*
https://www.dumpanalysis.org/blog/index.php/2010/10/04/crash-dump-analysis-patterns-part-108/

R756 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 111) [Pervasive System].*
https://www.dumpanalysis.org/blog/index.php/2010/10/26/crash-dump-analysis-patterns-part-111/

R757 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 114) [Crash Signature].*
https://www.dumpanalysis.org/blog/index.php/2010/11/09/crash-dump-analysis-patterns-part-114/

R758 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 117) [Invalid Parameter (process heap)].*
https://www.dumpanalysis.org/blog/index.php/2010/11/29/crash-dump-analysis-patterns-part-117/

R759 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 119) [Well-Tested Module].*
https://www.dumpanalysis.org/blog/index.php/2010/12/06/crash-dump-analysis-patterns-part-119/

R760 Dmitry Vostokov. 2010. *Crash Dump Analysis Patterns (Part 122) [Blocking Module].*
https://www.dumpanalysis.org/blog/index.php/2010/12/21/crash-dump-analysis-patterns-part-122/

R761 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 125) [Top Module].*
https://www.dumpanalysis.org/blog/index.php/2011/01/02/crash-dump-analysis-patterns-part-125/

R762 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 127b) [Technology-Specific Subtrace (dynamic memory)].*
https://www.dumpanalysis.org/blog/index.php/2011/04/25/crash-dump-analysis-patterns-part-127b/

R763 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 128) [Dialog Box].*
https://www.dumpanalysis.org/blog/index.php/2011/01/29/crash-dump-analysis-patterns-part-128/

R764 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 131) [Directing Module].*
https://www.dumpanalysis.org/blog/index.php/2011/03/07/crash-dump-analysis-patterns-part-131/

R765 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 134a) [Data Correlation (function parameters)].* https://www.dumpanalysis.org/blog/index.php/2011/03/28/crash-dump-analysis-patterns-part-134a/

R766 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 136) [Version-Specific Extension].* https://www.dumpanalysis.org/blog/index.php/2011/06/01/crash-dump-analysis-patterns-part-136/

R767 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 139) [Managed Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2011/06/17/crash-dump-analysis-patterns-part-139/

R768 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 140) [Coupled Modules].* https://www.dumpanalysis.org/blog/index.php/2011/06/20/crash-dump-analysis-patterns-part-140/

R769 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 143) [Pleiades].* https://www.dumpanalysis.org/blog/index.php/2011/06/25/crash-dump-analysis-patterns-part-143/

R770 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 146) [Problem Vocabulary].* https://www.dumpanalysis.org/blog/index.php/2011/07/15/crash-dump-analysis-patterns-part-146/

R771 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 149) [Double IRP Completion].* https://www.dumpanalysis.org/blog/index.php/2011/10/07/crash-dump-analysis-patterns-part-149/

R772 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 151b) [Annotated Disassembly (unmanaged code)].* https://www.dumpanalysis.org/blog/index.php/2024/01/28/crash-dump-analysis-patterns-part-151b/

R773 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 152c) [Handled Exception (kernel space)].* https://www.dumpanalysis.org/blog/index.php/2012/08/31/crash-dump-analysis-patterns-part-152c/

R774 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 155) [Hidden Parameter].* https://www.dumpanalysis.org/blog/index.php/2011/11/10/crash-dump-analysis-patterns-part-155/

R775 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 158) [System Object].* https://www.dumpanalysis.org/blog/index.php/2011/12/04/crash-dump-analysis-patterns-part-158/

R776 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 161) [Empty Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2011/12/12/crash-dump-analysis-patterns-part-161/

R777 Dmitry Vostokov. 2011. *Crash Dump Analysis Patterns (Part 164) [Network Packet Buildup].* https://www.dumpanalysis.org/blog/index.php/2011/12/25/crash-dump-analysis-patterns-part-164/

R778 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 167) [Regular Data].* https://www.dumpanalysis.org/blog/index.php/2012/02/12/crash-dump-analysis-patterns-part-167/

R779 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 170) [Coincidental Error Code].* https://www.dumpanalysis.org/blog/index.php/2012/04/05/crash-dump-analysis-patterns-part-170/

R780 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 173) [Value Adding Process].* https://www.dumpanalysis.org/blog/index.php/2012/05/20/crash-dump-analysis-patterns-part-173/

R781 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 176) [Spike Interval].* https://www.dumpanalysis.org/blog/index.php/2012/06/06/crash-dump-analysis-patterns-part-176/

R782 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 179) [Deviant Module].*
https://www.dumpanalysis.org/blog/index.php/2012/07/15/crash-dump-analysis-patterns-part-179/

R783 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 182) [Error Reporting Fault].*
https://www.dumpanalysis.org/blog/index.php/2012/10/09/crash-dump-analysis-patterns-part-182/

R784 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 185) [Disk Packet Buildup].*
https://www.dumpanalysis.org/blog/index.php/2012/11/05/crash-dump-analysis-patterns-part-185/

R785 Dmitry Vostokov. 2012. *Crash Dump Analysis Patterns (Part 188) [Critical Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2012/12/23/crash-dump-analysis-patterns-part-188/

R786 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 190b) [Module Collection (predicate)].*
https://www.dumpanalysis.org/blog/index.php/2014/04/19/crash-dump-analysis-patterns-part-190b/

R787 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 193) [Broken Link].*
https://www.dumpanalysis.org/blog/index.php/2013/01/09/crash-dump-analysis-patterns-part-193/

R788 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 196) [Reduced Symbolic Information].*
https://www.dumpanalysis.org/blog/index.php/2013/02/26/crash-dump-analysis-patterns-part-196/

R789 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 199) [One-Thread Process].*
https://www.dumpanalysis.org/blog/index.php/2013/05/28/crash-dump-analysis-patterns-part-199/

R790 Dmitry Vostokov. 2013. *Crash Dump Analysis Patterns (Part 202) [Small Value].*
https://www.dumpanalysis.org/blog/index.php/2013/11/09/crash-dump-analysis-patterns-part-202/

R791 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 205) [False Effective Address].*
https://www.dumpanalysis.org/blog/index.php/2014/04/26/crash-dump-analysis-patterns-part-205/

R792 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 208) [Hidden IRP].*
https://www.dumpanalysis.org/blog/index.php/2014/06/23/crash-dump-analysis-patterns-part-208/

R793 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 212) [Last Object].*
https://www.dumpanalysis.org/blog/index.php/2014/10/05/crash-dump-analysis-patterns-part-212/

R794 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 214) [Past Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2014/10/08/crash-dump-analysis-patterns-part-214/

R795 Dmitry Vostokov. 2014. *Crash Dump Analysis Patterns (Part 217) [Exception Module].*
https://www.dumpanalysis.org/blog/index.php/2014/11/29/crash-dump-analysis-patterns-part-217/

R796 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 220) [Hidden Call].*
https://www.dumpanalysis.org/blog/index.php/2015/02/17/crash-dump-analysis-patterns-part-220/

R797 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 223) [Crashed Process].*
https://www.dumpanalysis.org/blog/index.php/2015/03/07/crash-dump-analysis-patterns-part-223/

R798 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 226) [Internal Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2015/05/13/crash-dump-analysis-patterns-part-226/

R799 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 229) [Stack Trace Surface].*
https://www.dumpanalysis.org/blog/index.php/2015/09/13/crash-dump-analysis-patterns-part-229/

R800 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 232) [Active Thread (Windows)].*
https://www.dumpanalysis.org/blog/index.php/2015/10/31/crash-dump-analysis-patterns-part-232/

R801 Dmitry Vostokov. 2015. *Crash Dump Analysis Patterns (Part 235, Linux) [Critical Region].*
https://www.dumpanalysis.org/blog/index.php/2015/12/18/crash-dump-analysis-patterns-part-235-linux/

R802 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 239) [Not My Thread].*
https://www.dumpanalysis.org/blog/index.php/2016/05/02/crash-dump-analysis-patterns-part-239/

R803 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 242) [Stack Trace Signature].*
https://www.dumpanalysis.org/blog/index.php/2016/09/13/crash-dump-analysis-patterns-part-242/

R804 Dmitry Vostokov. 2016. *Crash Dump Analysis Patterns (Part 245) [Module Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2016/12/01/crash-dump-analysis-patterns-part-245/

R805 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 248) [Mirror Dump Set].*
https://www.dumpanalysis.org/blog/index.php/2017/02/19/crash-dump-analysis-patterns-part-248/

R806 Dmitry Vostokov. 2017. *Crash Dump Analysis Patterns (Part 251) [Frame Regularity].*
https://www.dumpanalysis.org/blog/index.php/2017/05/12/crash-dump-analysis-patterns-part-251/

R807 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 254) [Stack Trace Race].*
https://www.dumpanalysis.org/blog/index.php/2018/01/31/crash-dump-analysis-patterns-part-254/

R808 Dmitry Vostokov. 2018. *Crash Dump Analysis Patterns (Part 257) [Exception Reporting Thread].*
https://www.dumpanalysis.org/blog/index.php/2018/11/20/crash-dump-analysis-patterns-part-257/

R809 Dmitry Vostokov. 2019. *Crash Dump Analysis Patterns (Part 260) [Region Profile].*
https://www.dumpanalysis.org/blog/index.php/2019/09/27/crash-dump-analysis-patterns-part-260/

R810 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 263) [Hidden Stack].*
https://www.dumpanalysis.org/blog/index.php/2020/02/21/crash-dump-analysis-patterns-part-263/

R811 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 267) [Frame Trace].*
https://www.dumpanalysis.org/blog/index.php/2020/06/13/crash-dump-analysis-patterns-part-267/

R812 Dmitry Vostokov. 2020. *Crash Dump Analysis Patterns (Part 270) [Pointer Class].*
https://www.dumpanalysis.org/blog/index.php/2020/06/17/crash-dump-analysis-patterns-part-270/

R813 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 273) [C++ Object].*
https://www.dumpanalysis.org/blog/index.php/2021/01/25/crash-dump-analysis-patterns-part-273/

R814 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 276) [Saved Exception Context (.NET)].*
https://www.dumpanalysis.org/blog/index.php/2021/04/03/crash-dump-analysis-patterns-part-276/

R815 Dmitry Vostokov. 2021. *Crash Dump Analysis Patterns (Part 279) [Structure Field Collection].*
https://www.dumpanalysis.org/blog/index.php/2021/11/23/crash-dump-analysis-patterns-part-279/

R816 Dmitry Vostokov. 2022. *Crash Dump Analysis Patterns (Part 282) [COM Object].*
https://www.dumpanalysis.org/blog/index.php/2022/11/27/crash-dump-analysis-patterns-part-282/

R817 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 285) [Dereference Nearpoint].*
https://www.dumpanalysis.org/blog/index.php/2024/01/21/crash-dump-analysis-patterns-part-285/

R818 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 288) [Shadow Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2024/02/23/crash-dump-analysis-patterns-part-288/

R819 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 291) [Annotated Stack Trace].*
https://www.dumpanalysis.org/blog/index.php/2024/03/14/crash-dump-analysis-patterns-part-291/

R820 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 294) [Analysis Summary].*
https://www.dumpanalysis.org/blog/index.php/2024/03/18/crash-dump-analysis-patterns-part-294/

R821 Dmitry Vostokov. 2024. *Crash Dump Analysis Patterns (Part 297, Linux) [Function Pointer].*
https://www.dumpanalysis.org/blog/index.php/2024/06/15/crash-dump-analysis-patterns-part-297-linux/

R822 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 301) [Dump Context].*
https://www.dumpanalysis.org/blog/index.php/2025/10/21/crash-dump-analysis-patterns-part-301/

R823 Dmitry Vostokov. 2025. *Crash Dump Analysis Patterns (Part 304) [Latent Structure].*
https://www.dumpanalysis.org/blog/index.php/2025/12/07/crash-dump-analysis-patterns-part-304/

Related pattern catalogs and diagnostic methods

R824 Dmitry Vostokov. 2011. *User Interface Problem Analysis Patterns (Part 1) [Error Message Box].*
https://www.dumpanalysis.org/blog/index.php/2011/07/14/user-interface-problem-analysis-patterns-part-1/

R825 Dmitry Vostokov. 2012. *User Interface Problem Analysis Patterns (Part 2) [Unresponsive Window].*
https://www.dumpanalysis.org/blog/index.php/2012/09/09/user-interface-problem-analysis-patterns-part-2/

R826 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 1) [Memory Snapshot].*
https://www.dumpanalysis.org/blog/index.php/2010/09/24/structural-memory-patterns-part-1/

R827 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 2) [Aggregate Snapshot].*
https://www.dumpanalysis.org/blog/index.php/2010/09/26/structural-memory-patterns-part-2/

R828 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 3) [Snapshot Collection].*
https://www.dumpanalysis.org/blog/index.php/2010/09/26/structural-memory-patterns-part-3/

R829 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 4) [Memory Region].*
https://www.dumpanalysis.org/blog/index.php/2010/10/01/structural-memory-patterns-part-4/

R830 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 5) [Region Boundary].*
https://www.dumpanalysis.org/blog/index.php/2010/10/01/structural-memory-patterns-part-5/

R831 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 6) [Memory Hierarchy (General)].*
https://www.dumpanalysis.org/blog/index.php/2010/10/13/structural-memory-patterns-part-6/

R832 Dmitry Vostokov. 2010. *Structural Memory Patterns (Part 7) [Anchor Region].*
https://www.dumpanalysis.org/blog/index.php/2010/10/18/structural-memory-patterns-part-7/

R833 Dmitry Vostokov. 2016. *Structural Memory Patterns (Part 8) [Region Strata].*
https://www.dumpanalysis.org/blog/index.php/2016/08/08/structural-memory-patterns-part-8/

R834 Dmitry Vostokov. 2012. *Malware Analysis Patterns (Part 2) [Fake Module].*
https://www.dumpanalysis.org/blog/index.php/2012/12/29/malware-analysis-patterns-part-2/

R835 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 3) [Packed Code].*
https://www.dumpanalysis.org/blog/index.php/2013/01/19/malware-analysis-patterns-part-3/

R836 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 4) [Pre-Obfuscation Residue].*
https://www.dumpanalysis.org/blog/index.php/2013/01/19/malware-analysis-patterns-part-4/

R837 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 10) [Driver Device Collection].* https://www.dumpanalysis.org/blog/index.php/2013/01/20/malware-analysis-patterns-part-10/

R838 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 11) [RIP Stack Trace].* https://www.dumpanalysis.org/blog/index.php/2013/01/20/malware-analysis-patterns-part-11/

R839 Dmitry Vostokov. 2008. *Hooksware.* https://www.dumpanalysis.org/blog/index.php/2008/08/10/hooksware/

R840 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 18) [String Hint].* https://www.dumpanalysis.org/blog/index.php/2013/02/01/malware-analysis-patterns-part-18/

R841 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 20) [Namespace].* https://www.dumpanalysis.org/blog/index.php/2013/02/05/malware-analysis-patterns-part-20/

R842 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 21) [Patched Code].* https://www.dumpanalysis.org/blog/index.php/2013/02/09/malware-analysis-patterns-part-21/

R843 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 22) [Raw Pointer].* https://www.dumpanalysis.org/blog/index.php/2013/02/09/malware-analysis-patterns-part-22/

R844 Dmitry Vostokov. 2013. *Malware Analysis Patterns (Part 23) [Out-of-Module Pointer].* https://www.dumpanalysis.org/blog/index.php/2013/02/10/malware-analysis-patterns-part-23/

R845 Dmitry Vostokov. 2010. *Workaround Patterns (Part 1) [Hidden Output].* https://www.dumpanalysis.org/blog/index.php/2010/01/24/workaround-patterns/

R846 Dmitry Vostokov. 2010. *Workaround Patterns (Part 2) [Frozen Process].* https://www.dumpanalysis.org/blog/index.php/2010/01/25/workaround-patterns-part-2/

R847 Dmitry Vostokov. 2010. *Workaround Patterns (Part 3) [Axed Code].* https://www.dumpanalysis.org/blog/index.php/2010/01/26/workaround-patterns-part-3/

R848 Dmitry Vostokov. 2011. *Workaround Patterns (Part 4) [Fake API].* https://www.dumpanalysis.org/blog/index.php/2011/06/26/workaround-patterns-part-4/

R849 Dmitry Vostokov. 2015. *Workaround Patterns (Part 5) [Axed Data].* https://www.dumpanalysis.org/blog/index.php/2015/09/26/workaround-patterns-part-5/

R850 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 1) [API Query].* https://www.dumpanalysis.org/blog/index.php/2008/07/19/debugware-patterns-part-1/

R851 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 2) [Tool Façade].* https://www.dumpanalysis.org/blog/index.php/2008/07/20/debugware-patterns-part-2/

R852 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 3) [Configuration Wrapper].* https://www.dumpanalysis.org/blog/index.php/2008/08/04/debugware-patterns-part-3/

R853 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 4) [Dual Interface].* https://www.dumpanalysis.org/blog/index.php/2008/08/10/debugware-patterns-part-4/

R854 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 5) [Tool Chain].* https://www.dumpanalysis.org/blog/index.php/2008/09/12/debugware-patterns-part-5/

R855 Dmitry Vostokov. 2008. *DebugWare Patterns (Part 6) [Tool Box].* https://www.dumpanalysis.org/blog/index.php/2008/09/15/debugware-patterns-part-6/

R856 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 7) [Trace Expert].* https://www.dumpanalysis.org/blog/index.php/2009/09/10/debugware-patterns-part-7/

R857 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 8) [Troubleshooting Unit of Work].*
https://www.dumpanalysis.org/blog/index.php/2009/09/21/debugware-patterns-part-8/

R858 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 9) [Checklist].*
https://www.dumpanalysis.org/blog/index.php/2009/09/24/debugware-patterns-part-9/

R859 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 10) [Supporting Module].*
https://www.dumpanalysis.org/blog/index.php/2009/10/20/debugware-patterns-part-10/

R860 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 11) [Span Differentiator].*
https://www.dumpanalysis.org/blog/index.php/2009/10/21/debugware-patterns-part-11/

R861 Dmitry Vostokov. 2009. *DebugWare Patterns (Part 12) [Self Extractor].*
https://www.dumpanalysis.org/blog/index.php/2009/10/28/debugware-patterns-part-12/

R862 Dmitry Vostokov. 2010. *DebugWare Patterns (Part 13) [System Description Snapshot].*
https://www.dumpanalysis.org/blog/index.php/2010/11/25/debugware-patterns-part-13/

R863 Dmitry Vostokov. 2006. *Crash Dump Analysis AntiPatterns (Part 1) [Alien Component].*
https://www.dumpanalysis.org/blog/index.php/2006/11/01/crash-dump-analysis-antipatterns-part-1/

R864 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 2) [Zippocricy].*
https://www.dumpanalysis.org/blog/index.php/2007/02/28/crash-dump-analysis-antipatterns-part-2/

R865 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 3) [Word of Mouth].*
https://www.dumpanalysis.org/blog/index.php/2007/02/28/crash-dump-analysis-antipatterns-part-3/

R866 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 4) [Wrong Dump].*
https://www.dumpanalysis.org/blog/index.php/2007/03/01/crash-dump-analysis-antipatterns-part-4/

R867 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 5) [Fooled by Description].*
https://www.dumpanalysis.org/blog/index.php/2007/11/01/crash-dump-analysis-antipatterns-part-5/

R868 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 6) [Need the crash dump. Period.].*
https://www.dumpanalysis.org/blog/index.php/2007/11/22/crash-dump-analysis-antipatterns-part-6/

R869 Dmitry Vostokov. 2007. *Crash Dump Analysis AntiPatterns (Part 7) [Be Language].*
https://www.dumpanalysis.org/blog/index.php/2007/12/03/crash-dump-analysis-antipatterns-part-7/

R870 Dmitry Vostokov. 2008. *Crash Dump Analysis AntiPatterns (Part 8) [Fooled by Abbreviation].*
https://www.dumpanalysis.org/blog/index.php/2008/01/10/crash-dump-analysis-antipatterns-part-8/

R871 Dmitry Vostokov. 2008. *Crash Dump Analysis AntiPatterns (Part 9) [Symbolless or Imageless Analysis].*
https://www.dumpanalysis.org/blog/index.php/2008/03/27/crash-dump-analysis-antipatterns-part-9/

R872 Dmitry Vostokov. 2008. *Crash Dump Analysis AntiPatterns (Part 10) [Myopic Troubleshooting and Debugging].*
https://www.dumpanalysis.org/blog/index.php/2008/07/09/crash-dump-analysis-antipatterns-part-10/

R873 Dmitry Vostokov. 2008. *Crash Dump Analysis AntiPatterns (Part 11) [No Question].*
https://www.dumpanalysis.org/blog/index.php/2008/10/16/crash-dump-analysis-antipatterns-part-11/

R874 Dmitry Vostokov. 2008. *Crash Dump Analysis AntiPatterns (Part 12) [Missing Space].*
https://www.dumpanalysis.org/blog/index.php/2008/12/16/crash-dump-analysis-antipatterns-part-12/

R875 Dmitry Vostokov. 2009. *Crash Dump Analysis AntiPatterns (Part 13) [Habitual Reply].*
https://www.dumpanalysis.org/blog/index.php/2009/07/09/crash-dump-analysis-antipatterns-part-13/

R876 Dmitry Vostokov. 2010. *Crash Dump Analysis AntiPatterns (Part 14) [Wild Explanation].*
https://www.dumpanalysis.org/blog/index.php/2010/06/04/crash-dump-analysis-antipatterns-part-14/

R877 Dmitry Vostokov. 2013. *Static Code Analysis Patterns (Part 1).*
https://www.dumpanalysis.org/blog/index.php/2013/01/11/static-code-analysis-patterns-part-1/

R878 Dmitry Vostokov. 2010. *Analysis, Architectural, Design, Implementation and Usage Debugging Patterns (Part 0).*
https://www.dumpanalysis.org/blog/index.php/2010/12/03/analysis-architectural-design-implementation-and-usage-debugging-patterns-part-0/

R879 Dmitry Vostokov. 2012. *Analysis, Architectural, Design, Implementation and Usage Debugging Patterns (Part 1).*
https://www.dumpanalysis.org/blog/index.php/2012/01/07/analysis-architectural-design-implementation-and-usage-debugging-patterns-part-1/

R880 Dmitry Vostokov. *Diagnostic Analysis Patterns as MCP Tools for Agentic Diagnostics.*
https://www.dumpanalysis.org/diagnostic-analysis-pattern-mcp-tools

Additional patterns found through article cross-references

R881 Dmitry Vostokov. 2009. *Counterfactual Debugging: Dereference Fixpoints.*
https://www.dumpanalysis.org/blog/index.php/2009/09/11/counterfactual-debugging-dereference-fixpoints/

Additional primary references

R882 International Software Testing Qualifications Board. 2024. *Certified Tester Foundation Level Syllabus, v4.0.1.*
https://www.istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf

R883 Linux man-pages project. *intro(2) — Introduction to System Calls.*
https://man7.org/linux/man-pages/man2/intro.2.html

R884 Dmitry Vostokov. 2020. Software Trace Analysis, Revised Version, Part 1: Fundamentals and Basic Patterns. Public training slides. PDF p. 63, Glued Activity.
https://www.dumpanalysis.org/ASTA/Accelerated-Software-Trace-Analysis-Part1-Public.pdf