A TECHNOLOGICAL AND INTELLECTUAL HISTORY

History of Software Diagnostics, Observability, and Debugging

From machine fault finding and memory dump analysis to distributed observability and AI-assisted debugging

Software execution artifacts and their analysis

Concept and direction by Dmitry Vostokov, DumpAnalysis.org

Developed with AI assistance (GPT-6 Astra Max and Fable 5.1 Extra) and checked against the sources listed in this article.

Version 10

September 2026

Historical synthesis with 135 references, chronology, glossary, and evidence taxonomy

Abstract

Software diagnostics studies signs of software structure and behavior in software execution artifacts. Such artifacts include memory dumps, traces, logs, profiles, and other data collected during or after execution. In this history, we consider how these artifacts became available, how their analysis developed, and how diagnostics became related to debugging and observability. The three activities have different starting points. Debugging commonly starts with an observed failure or a suspected defect. Diagnostics starts with a problem description and the available evidence. Observability concerns the information that a system makes available for understanding its behavior.

We follow these developments from early machine fault finding and stored-program checking routines to symbolic debuggers, memory dump analysis, static and dynamic analysis, profiling, event logging, distributed tracing, and production instrumentation. We also consider program slicing, algorithmic debugging, managed runtimes, browser measurements, database provenance, and systematic exploration of concurrent execution. Later chapters cover site reliability engineering, telemetry standards, pattern-oriented and theoretical software diagnostics, machine learning, and AI-assisted investigation. A chronology, glossary, and evidence taxonomy provide additional ways to navigate the material.

These developments introduced additional artifacts and analysis methods. They also extended existing methods to new systems and levels of abstraction. A distributed service may still require a memory dump, a packet trace, or a reduced failing input. An AI-generated explanation may still require a debugger command or another experiment to check it. Throughout this history, we therefore distinguish the observed problem, the patterns found in its artifacts, the proposed mechanism, and the action taken to resolve it. Their correspondence has to be established in each particular case.

How to read this history

This article follows diagnostic concepts, methods, and tools across different computing environments. Dates refer to documented publications, standards, releases, or incidents. They do not always identify the beginning of a practice. Similar methods often developed independently, and an existing method sometimes received a new name when it moved to another platform. We therefore use claims of priority only where the available sources support them.

The chapters consider both software tools and the analysis processes around them. For example, a memory dump requires a capture method, matching symbols, an analysis tool, and sufficient problem information. A distributed trace requires instrumentation, context propagation, collection, and interpretation. We include these surrounding activities because a history of commands or file formats alone would leave much of software diagnostics unexplained.

Neighboring practices and their primary questions

Table 1. Working distinctions used throughout this article; practices overlap in real investigations.

Practice	Primary question	Characteristic evidence
Testing	Does behavior violate an expectation?	Inputs, assertions, oracles, coverage, failures
Debugging	Where and how does execution depart from intent?	Breakpoints, stacks, variables, watchpoints, replay
Monitoring	Is a known condition or threshold occurring?	Metrics, health checks, dashboards, alerts
Diagnostics	What happened, why, and what evidence supports the explanation?	Dumps, traces, logs, metrics, models, context
Observability	What internal behavior can be inferred from available outputs?	Correlated events, traces, metrics, logs, profiles
Forensics	What past activity can be reconstructed and preserved?	Snapshots, timelines, provenance, chain of custody
Root-cause analysis	Which causal mechanism best explains the failure?	Hypotheses, counter-evidence, experiments, dependency data

SEVEN RECURRING TRANSITIONS

- Physical fault symptoms → symbolic machine and program state
- Live intervention → reproducible post-mortem evidence
- Isolated snapshots → ordered event histories
- Single-process control → distributed context and causality
- Individual failures → population-scale crash and telemetry statistics
- Tool-specific craft → reusable patterns, protocols, and semantic conventions
- Manual inspection → automated ranking, agents, and machine-assisted explanation

Contents

Seven parts · 34 chapters · three reference appendices

SECTION	PAGE
Part I · Foundations of Software Diagnostics	
1. Vocabulary and Historical Scope	6
2. Physical Diagnosis Before Stored Programs	7
3. Stored-Program Diagnosis and Postmortem State	9
4. Interactive and Symbolic Debugging	10
5. Source-Level Debuggers and Debugging Interfaces	11
Part II · Software Engineering and Diagnostic Practice	
6. The Software Crisis and Formal Analysis	14
7. Mainframes and Memory Dump Analysis	16
8. Unix Diagnostic Tools and Composition	18
9. Performance Diagnostics and Resource Attribution	19
10. Logs and Event Systems	20
11. Crash Reporting Across Software Populations	22
Part III · Methods for Observing Software Behavior	
12. Dynamic Analysis and Early Detection	23
13. Fuzzing and Automated Fault Localization	24
14. Concurrency and Record-and-Replay	25
15. Embedded and Constrained-Environment Debugging	26
Part IV · Distributed Production Systems	
16. Distributed Causality and Global State	28
17. Network and Black-Box Diagnostics	29
18. Distributed Tracing and Application Performance Management	30

SECTION	PAGE
19. Production Instrumentation	30
20. Metrics and Alerting	31
21. Site Reliability Engineering and Incident Analysis	32
Part V · Software Observability	
22. Observability in Control Theory and Software	34
23. Software Execution Artifacts and Their Relationships	35
24. OpenTelemetry and Telemetry Standardization	36
25. Continuous Profiling	37
26. Telemetry Collection and Trust	38
Part VI · Diagnostic Knowledge and Automation	
27. Pattern-Oriented and Theoretical Software Diagnostics	39
28. Model-Based and Causal Diagnosis	43
29. Log Analysis and AIOps	44
30. LLM-Assisted Debugging and Diagnostic Agents	45
31. Diagnostics of AI and AI Systems	46
Part VII · Case Studies and Diagnostic Principles	
32. Diagnostic Lessons from Software Failures	48
33. Recurring Principles of Software Diagnostics	49
34. Further Directions in Software Diagnostics	52
Conclusion	53
Appendix A · Chronology of major developments	54
Appendix B · Glossary	58
Appendix C · Evidence taxonomy and blind spots	61
References	62

PART I

Foundations of Software Diagnostics

1. Vocabulary and Historical Scope

Let us first distinguish several meanings of software diagnostics, observability, and debugging. Debugging may refer to source-level investigation, but it may also include the whole process from a reported problem to a correction. Monitoring may refer to a particular check or to an operational data collection system. Observability has a mathematical meaning in control theory and several related meanings in software practice. We use the distinctions below to explain their relationships across this history.

We consider three related activities. In debugging, we control and inspect execution, change a program or its environment, and check the result. In diagnostics, we collect artifacts, reconstruct structure and behavior, recognize patterns, and investigate possible mechanisms. In observability, we consider what can be inferred from the outputs available from a system. These activities can be composed. A metric may lead to a trace, a trace to a memory dump, and dump analysis to a debugging experiment.

The sources for early diagnostic practices include machine manuals, operator records, local utilities, and accounts by their developers. Later periods provide more research papers and product documentation, although the terminology is not always consistent. We therefore follow capabilities as well as names. A breakpoint remains a selected observation point even when its interface changes from a console command to a source editor. Similarly, an operator log and a structured event stream both preserve selected information about activity that has already occurred.

Software diagnostics shares an analysis process with medicine and other engineering disciplines. We observe signs, collect additional information, consider possible explanations, and choose further tests or interventions. However, software is also a model of a required or imagined system. A failure may arise from an incorrect instruction, an invalid state, an interface mismatch, resource exhaustion, an unexpected interaction, or an incorrect requirement. The relevant mechanism depends on the level at which we consider the problem.

A debugger provides one means of obtaining and interpreting evidence. For example, a breakpoint may show the value of a variable, but more information is needed to explain why a service missed a response-time objective. A memory dump may preserve a corrupted heap without preserving the earlier write. A

request trace may omit a resource leak that developed over several hours. An error message may describe a consequence of the original problem. The choice of artifact therefore follows the diagnostic question.

The following working definitions separate neighboring practices:

- Testing checks software against expectations using selected inputs, properties, models, or other oracles.
- Debugging investigates and corrects software defects, commonly using controlled execution and experiments.
- Monitoring collects and evaluates selected indicators of software or service behavior.
- Diagnostics analyzes evidence to identify structural and behavioral problems, their possible mechanisms, and the information needed for further investigation.
- Forensics reconstructs past activity from artifacts, with particular attention to provenance, integrity, and the requirements of the investigation.
- Observability concerns the extent to which internal behavior can be inferred from available outputs. In software practice, these outputs may include traces, metrics, logs, profiles, and contextual information [42].

The boundaries depend on how an artifact is used. A compiler warning may prevent a later failure or help explain one that has already occurred. A performance profile may explain a timeout without any crash. An incident report may become input to another analysis or a source of new checks. We therefore consider these practices together while retaining the questions that distinguish them.

We also need to specify what we mean by a root cause. A complex incident can involve a triggering event, a latent defect, an unsuccessful safeguard, and a change in the operating environment. Each may explain part of the incident. A useful diagnosis identifies the mechanism and the level of explanation, states the remaining uncertainty, and indicates what additional evidence would support or contradict it.

The location where a failure appears may differ from the location where its mechanism started. Consider a pointer overwritten in one thread and later used by another. The second thread exposes the invalid state, but its stack alone does not identify the earlier write. At a higher level, an edge service may report a timeout caused by a queue elsewhere. In both cases, we extend the analysis beyond the immediate symptom and look for the relevant relationships.

2. Physical Diagnosis Before Stored Programs

Before electronic computing, machinery was diagnosed through inspection, measurements, and controlled substitution. Telegraphy and electrical engineering added circuit diagrams, test points, and instruments for checking signals and connections. Early calculating machines required similar work. A

failed calculation could result from a contact, relay, connection, or control setting. The physical arrangement of the machine was part of the evidence.

The Harvard Mark II logbook provides a well-known example. In 1947, an insect was found in a relay and attached to the logbook with the annotation “first actual case of bug being found.” The Smithsonian record explains that the technical use of bug preceded this incident [1]. The record is therefore evidence of an actual obstruction and of the terminology already used by the computing group.

Early electronic machines exposed state through lamps, switches, oscilloscopes, registers, and printed records. An operator could halt execution, inspect memory, alter an instruction, and continue. Such observations were close to the machine representation. They showed individual values, but the analyst still had to connect those values to the intended calculation. This separation between observation and interpretation remains present in later debuggers.

Several reusable diagnostic methods can already be discerned:

- Expose state through test points, indicators, or printed values.
- Localize a problem by changing one condition or substituting a known working component.
- Record observations, attempted changes, and their results.

Similar methods appear in later software investigations. Probes expose selected state, configuration changes test hypotheses, and incident timelines record the sequence of observations and interventions. Their implementations differ, but we can recognize the same analysis operations.

Physical fault isolation also required a useful classification of problems. Technicians distinguished power from logic, components from connections, and repeatable from intermittent behavior. Schematics and measurements supported these distinctions. In software, dependency maps, interface checks, and comparisons between working and failing configurations serve related purposes.

Early computing did not provide a clean separation between hardware, programs, and operation. A relay, a misread card, an incorrect instruction, or a wrongly loaded program could produce a similar failed calculation. Later specialization created separate hardware, development, and operations roles. However, firmware, virtualization, accelerators, and distributed infrastructure still require analysis across these levels.

The Mark II record also illustrates an early diagnostic artifact collection. The page contains a time, an observation, and the physical object associated with the malfunction. Its interpretation depends on their relationship. A contemporary incident package may contain a problem description, a memory dump, a trace, and an analysis report. The general requirement is the same: preserve enough context to explain why the artifacts belong to one investigation.

3. Stored-Program Diagnosis and Postmortem State

In a stored-program computer, instructions and data share memory. A wrong operation can therefore change both the calculation and the evidence available for its analysis. Early programs were loaded and executed during limited machine sessions, often without convenient interactive access. Programmers needed checking routines and selective output that would help them reconstruct an unsuccessful run afterward.

EDSAC provides an early documented example. Stanley Gill described methods for diagnosing programming mistakes in 1951 [2]. In the same year, Maurice Wilkes, David Wheeler, and Gill included error diagnosis in The Preparation of Programs for an Electronic Digital Computer [3]. Diagnostic work was already being presented as a subject that programmers could study and practice systematically.

The methods included checking routines and selected information about memory and execution. Their design was constrained by available machine time and output facilities. Recording more detail could help an investigation but also make a run more expensive. This remains a familiar artifact acquisition problem. We must choose what to capture before knowing exactly which part will explain the failure.

Several later diagnostic concepts can be related to these practices:

- Checks and assertions test expected relationships at selected execution points.
- Selective tracing records particular values or control transfers.
- Postmortem analysis examines state preserved after a run or failure.
- Symbolic interpretation connects recorded values to program concepts.
- Reproduction and retesting check whether the same conditions produce the problem after a change.

Batch execution could make reproduction more manageable than in a distributed system, but it did not make every observation free of interference. Diagnostic output used storage and processing time and could change the conditions of execution. Later terms such as probe effect and Heisenbug describe particular manifestations of this more general difficulty. Instrumented execution and the execution we intended to observe are not automatically equivalent.

By 1963, the EDSAC 2 work described diagnostic facilities in source-language terms for Autocode programs [4]. This reduced the amount of machine-level reconstruction required from the programmer. Later symbol tables, source maps, type information, and unwind records extend this approach. They provide mappings from execution artifacts to the abstractions used during software construction.

Memory dumps supported another separation. Once the relevant state had been printed or saved, analysis could continue without occupying the original machine. Artifact acquisition and artifact analysis

became distinct activities. This arrangement made it possible to send evidence to another analyst and later to collect failures centrally from many machines.

A saved state still differs from an execution history. A dump may contain the final instruction and damaged data while omitting the earlier operation that produced them. Additional checks and trace output can reduce this gap. Throughout the history that follows, we return to the relationship between a memory snapshot and the sequence of events from which it arose.

4. Interactive and Symbolic Debugging

Interactive computing allowed the programmer to intervene during execution. We could stop at a selected point, inspect state, change a value, and continue. Consoles, displays, time-sharing systems, and symbolic assemblers made this process more convenient than repeatedly submitting a modified batch program.

Early symbolic debuggers provided operations that remain recognizable: breakpoints, single stepping, register and memory inspection, value modification, and symbolic address display. Their command languages differed, but they combined control over execution with a means of examining its current state.

Symbols allowed an address to be interpreted as a routine or variable. Interactive control also shortened the interval between a hypothesis and its test. For example, an analyst could stop before a suspected overwrite, check an invariant, move the observation point, and repeat. This progressively narrowed the part of execution requiring detailed analysis.

A debugging session could itself be preserved as an artifact. Command histories, terminal output, and scripts recorded parts of the investigation. Later facilities included conditional breakpoints, watchpoints, expression evaluation, stack unwinding, remote targets, and reverse execution. Graphical interfaces provided additional views of threads, frames, and variables without changing the need to interpret them.

Interactive investigation has several constraints:

- A reported failure may not reproduce while a debugger is attached.
- Stopping execution may change thread scheduling or interactions with external systems.
- Compiler optimization may remove or rearrange source-level objects.
- A production process may be unsuitable for suspension.
- Observed state still requires analysis to distinguish a cause from its consequences.

These constraints explain the continuing use of postmortem artifacts. Interactive debugging supports experiments with a running program. A dump preserves selected state from a particular execution, including an execution that cannot easily be repeated. We often need both forms of access during one investigation.

The Computer History Museum's PDP-1 collection documents the development of interactive programs, editors, and debugging facilities in the same computing environment [58]. Direct access to a running computation changed how programmers investigated mistakes and how quickly they could check a proposed correction.

Historical sources use several expansions of DDT. The Computer History Museum finding aid records DEC Debugging Tape [58]. DEC's PDP-8/I manual uses Dynamic Debugging Technique [59], and the CP/M manual uses Dynamic Debugging Tool [60]. These sources should be distinguished when discussing the name. The debugger operations included examining and changing locations, setting stops, and continuing execution. Symbols connected addresses to names, and selected stops allowed the programmer to inspect state along an execution path.

Interactive Lisp systems provided a related approach at a higher level of abstraction. A read-evaluate-print loop and inspectable data allowed parts of a program to be investigated within its language environment. Quam's 1969 LISP debugging system documented interactive facilities beyond assembly-language inspection [61]. The environment could retain program context while the programmer examined behavior and changed definitions.

Such interaction also changes the system being investigated. Printing consumes resources, suspension changes timing, and modifying a value changes the subsequent execution. We therefore need to record these interventions when interpreting a debugging session. Production tracing, profiling, and replay later addressed different parts of the same observation problem.

5. Source-Level Debuggers and Debugging Interfaces

A source-level debugger uses additional information to map machine execution to source files, lines, scopes, variables, and types. The mapping is partial. An optimizer may combine operations, inline a function, eliminate a variable, or move its value between storage locations. Consequently, one source line may correspond to several instruction ranges, and some values may be unavailable at a chosen stop. Analysis of optimized code must take these transformations into account.

Unix environments provided command-line debuggers such as adb and dbx. GDB extended source-level and machine-level debugging across languages and targets. Its facilities include breakpoints, watchpoints, frames, expression evaluation, core-file analysis, process attachment, remote targets, and scripting [62][63]. Commands could also be combined into repeatable analysis procedures.

Integrated development environments arranged these facilities around source editors, stack panes, variable views, and breakpoint controls. Build settings and symbol information could be shared with the

debugging session. This reduced navigation between separate tools, although interpretation still depended on the debugger engine, runtime, and compiler information.

The Debug Adapter Protocol defines messages between a development tool and a debug adapter [64]. It separates the user interface from debugger-specific operations. Remote debugging introduces another boundary, where a target or stub exposes state and execution control. Both arrangements require care with capabilities, timing, and access permissions. A uniform interface does not make every target capable of the same observations.

Debugging Information and Source Maps

Symbolic debugging depends on a correspondence between executable code and the concepts used to construct it. Debugging information can describe source locations, types, scopes, variable locations, and the information needed to reconstruct calls. DWARF provides a standardized representation of many such relationships; its fifth version was published in 2017 [122]. This information is especially significant for optimized code. A source variable may have several locations during execution, exist only as a computed value, or no longer be available. A source line may correspond to several instructions, and their execution order may differ from a simple reading of the source. We therefore need to distinguish the source representation, the generated program, and the reconstruction presented by the debugger.

Source maps provide another mapping between generated artifacts and their sources. They are useful when JavaScript has been bundled or minified, when a language has been compiled to JavaScript, and in other transformations covered by the format. Ecma standardized the source map format as ECMA-426 in December 2024, documenting practices that already existed [121]. For diagnostics, the map must correspond to the particular generated artifact. Applying a map from another build can produce a plausible but incorrect source location. The generated file, its identity, and the associated map therefore belong to the same diagnostic artifact collection.

Browser Execution and User Experience

Web applications introduced a diagnostic boundary between service execution and the user’s browser. A successful HTTP response does not establish that a page became usable. Name resolution, connection setup, redirection, document processing, script execution, and resource loading can contribute different delays. The W3C Navigation Timing Recommendation of 2012 exposed timing information for stages of document navigation through a browser interface [120]. Such measurements allowed investigation to include the client’s view of a request, instead of relying entirely on server records. They also made it necessary to identify which event a reported duration actually measured.

Consider a page whose server trace ends normally while a client-side exception prevents a control from appearing. We need the browser error, the relevant generated code and source map, the application version, and the sequence of user activity. A backend trace alone cannot supply these artifacts. Conversely, a browser timeout may be explained by a delayed dependency on the server. The analysis crosses the boundary by relating identities and intervals, while retaining the different meanings of the observations on each side.

PART II

Software Engineering and Diagnostic Practice

6. The Software Crisis and Formal Analysis

During the 1960s, operating systems, compilers, and business applications increased in size and complexity. Their development involved more people, interfaces, and changes. The 1968 NATO conference discussed the resulting difficulties under the heading of software engineering [5]. Failures were considered in relation to project organization and software construction as well as individual programming mistakes.

This wider scope affected diagnostics. Structured programming, modular design, information hiding, inspections, testing, and configuration management attempted to prevent defects and make remaining problems easier to localize. The structure of a program influenced the structure of its investigation. A well-defined interface could become a useful boundary for a check or a diagnostic experiment.

Dijkstra's notes on structured programming emphasized the limitation of testing as evidence of correctness [6]. Hoare's axiomatic approach described relationships between assertions and program operations [7]. Preconditions, postconditions, and loop invariants also have a diagnostic use. They state what should hold at particular points, so a detected violation can locate an invalid transition more closely than a later crash.

Static analysis provided another source of diagnostic information. Johnson's lint checked C programs for suspicious constructs, inconsistent types, and portability problems [10]. McCabe related control-flow structure to test and maintenance complexity [11]. These methods examined program artifacts before a production failure, although their output could also help explain an observed problem.

Dependability terminology provides a more precise distinction between fault, error, and failure [124]:

A fault is a judged or hypothesized cause of an error. An error is a part of the system state that may lead to a service failure. A failure occurs when the delivered service no longer satisfies the specified correct behavior. A code defect is one possible fault.

The boundary of the system matters. A failure of one component may become a fault affecting another component. Diagnostics commonly starts with a visible failure and investigates the states and conditions leading to it. Testing selects conditions under which failures may become visible. Formal analysis examines

the behavior permitted by a model. These activities use different starting points and can inform one another [124].

Brooks's discussion of essential and accidental complexity places these methods in perspective [55]. Better tools can remove avoidable work and expose useful evidence. They do not remove the need to understand the required behavior and the relationships among components. Diagnostic improvements therefore depend on both tools and the models used by their analysts.

Abstract interpretation provided a mathematical framework for approximating program behavior [77]. Model checking made it possible to explore finite-state models against temporal properties; Clarke and Emerson's early work addressed synchronization structures [78]. A warning or counterexample obtained this way is another analysis artifact. Its meaning depends on the model, the property, and the assumptions used to represent execution.

Static analysis also used repeated code behavior to infer likely rules. Engler and colleagues examined deviations from common patterns in systems code as possible defects [79]. This reduced dependence on a separately written specification. However, a frequently repeated practice can still be wrong, and an unusual construct can be intentional. A detected deviation provides a reason for investigation.

Static results and runtime observations support different conclusions. A runtime trace describes selected events from a concrete execution. A sound analysis may rule out a behavior within its stated model, while an approximate warning may include paths that cannot occur. We can use static warnings to choose instrumentation, runtime artifacts to refine a model, and incident findings to introduce additional properties for checking.

Program Slicing

Program slicing made the selection of relevant code an explicit analysis problem. Weiser's 1984 paper described slices that preserve the behavior relevant to selected variables at a selected program point [113]. Data dependencies relate a value to the operations that supply it, while control dependencies relate execution to decisions that govern it. A slice can therefore help us investigate an incorrect result without initially examining every statement. Static slicing considers possible execution within an analysis model; dynamic slicing restricts the question to a particular execution. Neither form automatically identifies the defective statement. The result is a smaller representation of relevant dependencies that still requires interpretation.

For example, an incorrect total may depend on an earlier conversion, a loop bound, and a branch that selected one collection of records. Looking only at the final addition can miss these relationships. The slicing question is therefore different from asking which line last wrote the variable. We ask which parts of the computation could have contributed to the value under consideration.

Algorithmic and Question-Oriented Debugging

Shapiro's Algorithmic Program Debugging, published as a book in 1983, developed diagnosis through a representation of computation and an oracle that judges whether particular results are correct [114]. In its logic-programming setting, the process can narrow the search by asking questions about subcomputations. The oracle may be a person who knows the intended behavior. This separates an important source of knowledge from the mechanics of execution: the program records what happened, while the oracle supplies judgments about what should have happened. Incorrect or unavailable judgments limit the resulting diagnosis.

The Whyline work presented in 2004 provided a debugging interface for questions about why an observed behavior occurred and why an expected behavior did not occur [115]. Its original setting was the Alice programming environment, where recorded execution and program dependencies supported the answers. This is a different way to organize access to debugging information. Instead of starting with a command for inspecting a variable or stack, the programmer starts with a question about behavior. Later conversational debugging has a related interface objective, but a language model's proposed answer still needs to be connected to actual execution evidence.

7. Mainframes and Memory Dump Analysis

Multiprogramming operating systems introduced standardized messages, termination codes, and diagnostic artifacts. In IBM mainframe environments, an abnormal end, or ABEND, could be associated with registers, program status words, storage dumps, and job information. The 1967 System/360 operator guide documents messages and abnormal-end storage dumps as part of normal system operation [8]. This made failures available for a structured support process.

The process could involve several roles and transformations:

- An operator observes a termination or another service problem.
- The system or operator captures state and identifying information.
- Support classifies the problem and checks existing knowledge.
- An analyst reconstructs relevant structures from registers, stacks, control blocks, and memory.
- Engineering investigates the mechanism and supplies a correction or requests additional artifacts.

A memory dump preserves selected state over a capture interval. Depending on its type, it can contain stacks, heaps, modules, handles, thread information, buffers, and operating-system structures. We often treat it as a snapshot, but capture is not automatically instantaneous or globally consistent. The earlier

operation that damaged a structure may also be absent. Both the contents and the acquisition method constrain subsequent analysis.

Unix core files and later desktop crash dumps extended this workflow to other platforms. A debugger could use related commands against live memory and saved state. Correct interpretation requires matching executables and debugging information. A stack resolved with the wrong symbols can look plausible while referring to another build. Symbol archives, module identities, and retained binaries therefore became part of diagnostic infrastructure.

Desktop crash collection included tools such as Dr. Watson and Windows Error Reporting. A minidump stores a selected subset of process information so that capture and transfer can be practical on an end-user machine [34]. The MiniDumpWriteDump interface allows the caller to choose categories of data to include [35]. This makes the capture policy an explicit part of the artifact.

Larger dumps may avoid a request to reproduce the problem again, but they also increase collection time, storage, and exposure of process data. Smaller dumps reduce these costs while increasing the possibility of missing relevant memory. Capture tiers, sampling, retention rules, and escalation procedures developed around this choice. The right artifact depends on the problem and the conditions under which it can be acquired.

We can classify commonly available dump information by its diagnostic use:

- Registers and exception records describe the immediate execution context.
- Stacks provide evidence of nested calls, subject to unwind and optimization constraints.
- Modules and symbols connect addresses to code, names, and builds.
- Heaps and object graphs expose current allocation and reference structures.
- Threads and synchronization objects help identify waits, deadlocks, and resource dependencies.
- Operating-system structures connect execution to files, sockets, devices, and kernel activity.
- Strings and buffers may retain earlier requests, messages, or configuration values.

From these values, the analyst reconstructs a model of the system. We identify objects and relationships, compare them with expected structures, and consider mechanisms that could explain the differences. The faulting thread is only one part of that model. Likewise, a recognizable stack or exception signature can guide the analysis without uniquely identifying its cause.

Dump formats preserve different subsets of state. Linux documents its core generation and filtering rules [65]. Windows minidumps provide selectable categories of process information [34][35]. Managed runtimes require further information about objects, generated code, and garbage collection. A file called

a dump therefore needs to be identified by its format, scope, and acquisition policy before its apparent omissions are interpreted.

8. Unix Diagnostic Tools and Composition

Unix provided a set of abstractions around which many diagnostic tools could be combined. Files, processes, pipes, signals, and command interpreters were central to the system described by Ritchie and Thompson [9]. A working investigation could use a debugger, disassembler, process listing, symbol inspection, text search, and a shell script. The tools produced artifacts that could become input to other tools.

Such composition made evidence transformations easy to repeat. Output could be filtered, sorted, compared, and summarized. A command sequence could be saved and applied to another artifact or another machine. This supported both investigation and the construction of diagnostic procedures from previously successful analysis steps.

Profiling addressed resource consumption. The gprof work combined sampled program-counter information with calling relationships [12]. A profile could therefore distinguish the cost of a function from the contexts in which it was called. A frequently executed function might require investigation of its callers even when its own implementation was efficient.

Several recurring profiling methods are visible here:

- Sampling estimates aggregate behavior without recording every event.
- Call and stack context support attribution beyond a single counter.
- Comparison across workloads, builds, or time intervals helps identify significant differences.

Unix also made diagnostic facilities available in the operating environment. An analyst could examine a core file, a system log, process statistics, and network behavior on related machines. This required movement between program code and operational evidence well before later organizational terms such as DevOps became common.

Core collection gradually acquired its own management facilities. Linux core generation depends on signals, limits, dumpability, naming rules, namespaces, and handlers [65]. A service such as systemd-coredump can associate an image with metadata and retention policies. The original artifact remains important, but its collection and later availability are also engineering concerns.

System-call tracing selects an abstraction boundary between a program and the operating system. File access, process operations, memory mappings, and network requests have recognizable interfaces even when application source is unavailable. Tools such as strace can therefore provide useful evidence without

instruction-by-instruction investigation. The choice of boundary often determines how much raw data must be interpreted.

A useful diagnostic pipeline can later become a script, a support utility, or a monitoring check. This movement from a particular investigation to a reusable procedure is also present in debugger scripts, notebooks, telemetry queries, and diagnostic agents. In each case, we need to preserve the assumptions under which the transformation produced a meaningful result.

9. Performance Diagnostics and Resource Attribution

A program may produce the expected output and still have a performance problem. It can exceed a deadline, consume too much memory, or stop making useful progress under load. Performance diagnostics examines the relationship between work and the resources used to perform it. We need to identify both what was measured and the conditions under which the measurement was obtained.

Statistical profiling estimates where execution time is spent by sampling instructions or stacks. Instrumentation records selected entries, exits, allocations, or other events. The two methods provide different kinds of evidence and impose different costs. As discussed in Chapter 8, gprof combined sampling with calling information to relate function costs to their contexts [12].

Several distinctions are important when interpreting profiles. Inclusive time includes work performed by callees; exclusive time separates it. Wall time includes periods when a thread is waiting, whereas thread CPU time measures its execution. An average does not describe the slowest requests. An aggregated stack display does not retain the order of all sampled events. The measurement must therefore be connected to a workload, a time interval, and a particular performance question.

Hardware performance counters added observations of cycles, cache behavior, branch prediction, and other processor events. Tools such as perf combined these measurements with software events and stack sampling. However, access to performance information can also expose execution context. Linux documents permissions and data exposure as part of perf security [104]. The collection method belongs in the interpretation of the profile.

A locally expensive operation is not always the operation that determines the user-visible delay. It may execute outside the relevant path or overlap other work. Conversely, a short operation repeated many times may contribute most of the total cost. We therefore relate local measurements to the organization of the computation. Distributed traces, queue measurements, and critical-path analysis extend this same requirement to multiple processes.

Tail Latency and Query Plans

Distributed performance also requires attention to the distribution of durations. Dean and Barroso's 2013 discussion of the tail at scale explained why infrequent slow operations become significant when a request depends on many components [123]. A mean can conceal this behavior. We need to know the population represented by a percentile, the interval of aggregation, and the relationship between local work and the complete request. Percentiles from separate services cannot simply be added to obtain an end-to-end percentile. A trace of a slow request and a latency distribution answer different questions: one describes a particular execution, while the other describes a selected population.

Databases provide another form of performance artifact: the query plan. PostgreSQL's EXPLAIN describes the plan selected by its optimizer, including estimated rows and costs. EXPLAIN ANALYZE executes the statement and adds observations from the actual execution [130]. Differences between estimated and observed row counts can direct attention to the assumptions behind plan selection. The analyst must retain the query, schema, data conditions, parameters, and relevant settings. The latter command is an experiment with effects: it runs the statement and can modify data when the statement does so. An estimated plan, an observed execution plan, and a production workload therefore need to be distinguished.

10. Logs and Event Systems

A running production system must preserve information for people who are not present at the time of a problem. Logging provides a way to record selected decisions, states, and events. Text messages were easy to emit and inspect, but their usefulness depended on the information chosen by developers and retained by the surrounding system.

BSD syslog separated event generation from collection and routing. RFC 3164 documented an existing practice in 2001 and described its Berkeley origins [16]. Facility and severity fields provided a small shared classification. Remote collection made records available outside the originating host, including after some host failures. It also enabled comparison of events from several machines.

Windows Event Logging provided a centralized service with application, security, and system logs; its infrastructure was redesigned in Windows Vista [17]. Additional channels and ETW providers later extended the available information. In these systems, the component producing an event and the service storing or presenting it could evolve separately.

This separation also introduced diagnostic constraints:

- Unsynchronized clocks can give a misleading event order.

- Different components can use severity levels differently.
- Missing or reused identifiers can prevent correct correlation.
- Changed message templates can disrupt comparison across versions.
- Buffering, sampling, rotation, and loss can remove relevant messages.
- Recorded values can expose sensitive information.
- Formatting and transporting messages can affect execution.
- An uninstrumented condition can remain absent from the log.

Structured logging addressed some of these constraints. Records acquired explicit timestamps, process and thread identities, service and build information, and request identifiers. Named fields supplemented message text and supported indexing and comparison. Retention and access policies became part of the design because a useful operational record could also contain customer or application data.

Crash collection followed a related development. A report could be associated with a build, resolved with retained symbols, classified by a signature, and compared with reports from other machines. This allowed an analyst to investigate relationships between failures and versions, drivers, devices, or configurations that were difficult to discern from one dump.

Such grouping requires interpretation. The same instruction can expose corruption produced by different defects. Conversely, one defect may lead to several crash locations. Signature counts therefore support prioritization and hypothesis generation. Representative artifacts and additional context are needed to determine whether a group has a shared mechanism.

Both logging and crash collection made diagnostic preparation part of product development. Error messages, event fields, identifiers, symbols, and support artifacts had to remain useful when the original developer was unavailable. The quality of later analysis depended partly on these earlier decisions.

A log records a selected history of execution. The developer chooses the observation points and the information attached to them. Collection infrastructure further determines buffering, ordering, transformation, and retention. We must therefore distinguish an event in the system from a message about that event and from the message finally available to the analyst.

RFC 5424 specified a syslog format with structured data and explicit fields for time, application, process, and message identity [70]. RFC 5848 added mechanisms for authentication, integrity, sequencing, and detection of missing messages [100]. These developments addressed the reliability of the record as well as its readability.

ETW supports dynamically enabled kernel and application events through providers, sessions, controllers, and consumers [71]. Microsoft describes the integration of event logging and ETW in the Windows Vista

eventing architecture [101]. This provided a common way to collect selected events at rates and levels of detail that would be difficult to manage through ordinary printed messages.

Apple's unified logging provides another implementation of structured collection, selective persistence, and privacy controls [99]. Such designs separate efficient event capture from later presentation. The analyst consequently depends on the relevant schemas, decoding information, and version identities to interpret the stored data correctly.

Studies of industrial logging practices examined where developers place log statements and what those statements reveal [72]. More output does not necessarily provide better evidence. Repeated messages can obscure significant transitions, while missing identifiers can prevent reconstruction. We need events that distinguish plausible explanations of the problem and remain interpretable across the components involved.

11. Crash Reporting Across Software Populations

Software distributed to many machines fails under configurations unavailable to its developers. Automated crash reporting collects artifacts from these executions so that individual failures and their distribution can be investigated together.

A reporting pipeline detects a termination, captures and transfers an artifact under the collection policy, and associates it with retained symbols. The backend reconstructs stacks and groups signatures. Incorrect build identity or symbolication can separate related failures or combine unrelated ones.

The Windows Error Reporting study describes billions of reports and comparisons across crash buckets, modules, and hardware [66]. Such comparisons show affected populations, associated attributes, and changes after an update. Detailed investigation of selected dumps supplies additional evidence about the mechanisms.

Breakpad separated capture from processing and stack reconstruction [67]. Crashpad added an external handler, a local report database, and annotations around minidump interchange [68]. Apple's reports likewise connect exceptions and stacks to device and binary context [69].

Frequency alone does not establish importance or cause. A rare failure may be severe, and one signature may contain several mechanisms. We therefore move between population comparisons and individual artifacts, retaining the collection and grouping policies needed to interpret both.

PART III

Methods for Observing Software Behavior

12. Dynamic Analysis and Early Detection

Dynamic analysis adds observations or checks while a program executes. This can reveal an invalid operation before its consequences have spread through memory or other components. The additional information is obtained at a cost in execution time, memory, or changed timing. We therefore consider both the detected condition and the conditions of observation.

Purify, described in 1992, inserted checking instructions into object code to detect memory access errors and leaks [20]. It associated ordinary execution with additional information about valid memory use. This could identify an invalid access closer to its occurrence than a later failure caused by damaged state.

Eraser instrumented binary code and checked shared-memory access and locking behavior at run time [21]. Its 1997 account illustrates a different diagnostic property. A run may return the expected result even though its synchronization permits a race. We therefore need information about relationships between accesses, not only the values that happened to result in one run.

Valgrind provided a framework for dynamic binary instrumentation, with tools such as Memcheck maintaining additional state about memory [30]. AddressSanitizer combined compiler instrumentation with runtime support for detecting important classes of memory errors [33]. These approaches made invalid accesses available as explicit diagnostic events while differing in coverage, overhead, and build requirements.

We can distinguish several ways of introducing such observations:

- Source or compiler instrumentation adds checks while source-level structure is available.
- Binary instrumentation observes or rewrites executable operations.
- Interposition observes selected library, allocation, synchronization, or I/O boundaries.
- Hardware facilities provide watchpoints, counters, trace information, or memory protection.

Each method has a particular scope. Compiler instrumentation requires a suitable build. Binary analysis must interpret the executable representation and may add substantial overhead. Interposition observes only the selected interfaces. Hardware facilities depend on the processor and have limits of their own. The choice follows the suspected problem and the environment in which it can be reproduced.

Generated inputs and reduced examples provide additional support for dynamic investigation. Fuzzing explores inputs that a developer may not have anticipated. Delta debugging searches for a smaller set of circumstances that still produces a failure [22]. A reduced input or configuration can then be analyzed with a debugger, sanitizer, or another more expensive observation method.

The interval between an invalid operation and its detection affects diagnostic complexity. During a long interval, corrupted state may propagate and the original context may disappear. Assertions, sanitizers, and invariant checks can shorten this interval by exposing the first violation they are capable of detecting. Their output connects a concrete operation to an expected property.

Static, dynamic, and postmortem analysis cover different information. Static methods consider modeled possibilities. Dynamic methods observe particular executions. Postmortem methods reconstruct state preserved by a capture process. An investigation can use all three, especially when one method identifies a question that another can answer more directly.

For example, an ordinary crash may occur inside an allocator after earlier heap corruption. A sanitizer may identify an invalid write closer to the start of the mechanism. A race detector may identify conflicting accesses before the resulting value causes a visible failure. Earlier detection changes the artifact available for analysis and can reduce the number of possible explanations.

However, an instrumented run does not cover every execution. A detector can miss relevant behavior, and a changed build or schedule can alter the problem. Findings therefore need to be related to the production configuration. Preserved inputs, build identities, and repeated checks help establish that a detected violation explains the originally reported failure.

13. Fuzzing and Automated Fault Localization

Fuzz testing systematically supplies inputs that may violate a program’s assumptions. Miller, Fredriksen, and So’s 1990 study applied random input to Unix utilities and documented failures [73]. The work showed that apparently simple input generation could expose problems missed by ordinary use and existing testing.

Coverage-guided fuzzing uses execution feedback to retain inputs that reach additional behavior. LibFuzzer combines a target harness, an input corpus, compiler feedback, and failure detectors such as sanitizers [74]. The resulting input is a diagnostic artifact. Its usefulness depends on retaining the corresponding build, invocation, and environment so that the observed failure can be investigated again.

Delta debugging reduces a failing input, change, or configuration while repeatedly checking whether the symptom remains [22]. The result is minimal relative to the reduction method and oracle, and need not

be the smallest possible example. It shows that the retained circumstances are sufficient for the observed failure under the tested conditions. The mechanism still requires analysis.

Spectrum-based localization compares test outcomes with coverage to rank program elements. Tarantula visualized these relationships [23]. Statistical bug isolation similarly examined predicates associated with failing executions [28]. Such rankings help choose where to investigate. A highly ranked element may participate in the failure without being the location of the underlying defect.

Automated repair extends the process by searching for a program change that satisfies available checks. A passing test suite does not establish that the intended behavior has been restored in all relevant cases. A proposed patch may remove a symptom or fit an incomplete oracle. We therefore retain the failing example, examine the mechanism, and test the correction against additional expectations and counterexamples.

Properties and Symbolic Execution

Property-based testing provides an oracle in the form of a property that generated inputs should satisfy. QuickCheck's 2000 work applied this approach to Haskell programs, with generators controlling the input data used in randomized tests [127]. A failed property supplies a concrete starting point for diagnosis. However, the meaning of success depends on both the property and the generated population. A property may omit a required behavior, and a generator may rarely produce the conditions that expose a defect. The test definition and the generated input are therefore part of the diagnostic evidence.

Symbolic execution explores another relationship between inputs and execution paths. KLEE, presented in 2008, used symbolic execution and constraint solving to generate inputs for complex systems programs [128]. Instead of choosing only concrete input values in advance, the analysis accumulates conditions associated with a path and seeks values that satisfy them. A generated test can then expose a particular path or failure. The approach depends on the modeled environment and available resources; the number of paths can grow rapidly. Reexecuting a generated input in the intended environment helps establish that the reported problem is not merely a consequence of the analysis model.

14. Concurrency and Record-and-Replay

Concurrent execution depends on scheduling, interrupts, I/O completion, synchronization, and memory ordering. A repeated invocation can therefore follow a different path even with apparently identical input. A breakpoint or additional logging may change the schedule enough to remove the visible problem. Reproduction becomes a diagnostic task of its own.

Record-and-replay captures information needed to reproduce an observed execution. Depending on the system, this includes input values, system-call results, signals, and scheduling decisions. The recorder must define which sources of nondeterminism it controls. A successful replay provides a stable execution against which the analyst can apply new observation points.

The rr project demonstrated deployable recording and deterministic replay for supported Linux workloads [75]. Its design uses particular hardware and execution constraints, including control over thread execution. During replay, an analyst can revisit the same state, introduce breakpoints, and investigate earlier changes. These operations concern the recorded execution; they do not demonstrate that all possible executions have been covered.

Microsoft's Time Travel Debugging records a user-mode process for later navigation in WinDbg [76]. An investigation can begin at an observed invalid state and move backward toward earlier operations affecting it. This reduces dependence on guessing the right breakpoint before a difficult reproduction attempt.

Recording has its own limits. Artifacts can become large, external services may remain outside the recorded scope, and unsupported sources of nondeterminism can prevent faithful replay. Trace data can also contain sensitive process information. We therefore need to preserve the recorder version, configuration, coverage, and integrity alongside the execution history.

Systematic Exploration of Schedules

Concurrent programs add scheduling choices to the conditions of reproduction. CHESS, described in 2008, systematically explored thread schedules and preserved the schedules that exposed failures [118]. This differs from repeatedly running the same stress test and hoping that a rare interleaving occurs again. It also differs from replaying one previously recorded execution: schedule exploration searches for other behaviors within the supported execution model. The search is bounded, and operations outside the model can restrict what is explored or reproduced. For diagnosis, the useful artifact includes the failing schedule together with the program and test conditions that give it meaning. A stack snapshot can show threads after a failure; a controlled schedule can help explain the ordering that led there.

15. Embedded and Constrained-Environment Debugging

An embedded controller, kernel, or mobile application may not support the same diagnostic methods as a desktop process. Memory and storage can be limited. A kernel failure can damage the services required for capture. A real-time system may become incorrect merely because execution is stopped. These conditions influence both artifact acquisition and the analysis that can follow.

In-circuit emulators, JTAG interfaces, breakpoints, watchpoints, and hardware trace units provide access when software cannot report its own state. Raw registers and addresses still require interpretation. Symbols, calling conventions, task information, and runtime structures connect hardware observations to software behavior.

Kernel debugging can use a separate machine and a serial or network connection. Production investigations often depend instead on crash dumps or event buffers. The capture path must work with limited assumptions because locks, allocators, drivers, or filesystems may already be involved in the failure. This illustrates a general acquisition requirement: consider the failure dependencies of the diagnostic mechanism itself.

Mobile crash and device reports include information about threads, binaries, operating-system versions, and termination conditions [69]. Watchdog or responsiveness failures may require a different interpretation from an unhandled exception. Retained symbols and device context are needed to connect a customer report to the relevant program representation.

A flight recorder keeps a bounded history of selected events, often using a circular buffer. A trigger or explicit request preserves the recent contents for analysis. This provides pre-failure information at a predictable storage cost. However, the analyst must know the recording interval, enabled event classes, and whether the relevant history was overwritten or dropped.

Managed Runtime Flight Recorders

Managed runtimes also developed event recording facilities that preserve recent execution information. JDK Flight Recorder was integrated into OpenJDK through JEP 328 for JDK 11 in 2018; the technology had existed before this integration [117]. Its event framework and recording buffers make selected JVM and application activity available for later analysis. Depending on configuration, a recording may contain information about threads, allocation, garbage collection, compilation, and other runtime events. This provides a history around a problem that a single thread dump cannot preserve. The event settings, thresholds, buffer limits, and duration still determine what is available. A flight recording, heap dump, and thread dump are related artifacts with different coverage, so collecting one does not establish that the others are unnecessary.

PART IV

Distributed Production Systems

16. Distributed Causality and Global State

A distributed request can pass through services, queues, databases, caches, and external dependencies. Components have separate state and can fail, retry, or continue independently. No individual thread stack contains all of these relationships. We need artifacts that preserve identity and ordering across the relevant boundaries.

Lamport's 1978 paper described the happened-before relation and logical clocks [13]. Chandy and Lamport later showed how to record a consistent distributed snapshot under specified communication assumptions [14]. Fidge's vector timestamps provided additional information for distinguishing causal order from concurrency [80]. These results help explain why a global diagnostic view requires more than a timestamp-sorted collection of local observations.

Manual log correlation initially relied on host names, clock times, and message content. As request paths became longer and more concurrent, these fields were often insufficient. Propagated context made it possible to associate observations with the work that connected them. Its presence or absence became a property of the instrumented execution.

Pinpoint used request tracking and statistical analysis to investigate failures in Internet services [24]. The 2003 Magpie paper proposed online modeling from correlated events across machines [25]. The 2004 paper developed request extraction and workload modeling in greater detail [129]. Related work by Aguilera and colleagues examined performance diagnosis from communication between systems treated as black boxes [26]. These approaches show several ways to reconstruct behavior beyond one process.

X-Trace proposed tracing across layers and administrative boundaries [29]. Dapper described Google's production tracing infrastructure and its collection constraints [31]. The later common span model can be described through the following elements:

- A trace groups observations of related work.
- A span describes an operation over a time interval, with attributes and status information.
- Parent relationships and additional links connect operations.
- Context propagation carries identifiers across execution boundaries.
- Sampling and collection policies determine which observations are retained.

- A backend assembles and queries the available records.

This model supports investigation of request paths, queueing, retries, fan-out, and dependency delays. It also has boundaries. Sampling may omit a failure. Asynchronous work can lose context. Batch processing can relate one operation to several earlier operations. Resource exhaustion may develop outside any one request. We therefore combine traces with other artifacts when the mechanism crosses the recorded scope.

Clock time and causal order must be interpreted separately. Wall-clock timestamps relate activity to external events, but clock errors can reverse their apparent order. Monotonic clocks help measure local durations. Parent relationships, message identities, and span links provide other ordering evidence. A merged timeline should preserve these distinctions instead of presenting every event as part of one fully known sequence.

Context propagation is itself software behavior. Missing propagation separates related work, and incorrect reuse can combine unrelated requests. The existence of a trace identifier consequently does not prove that the reconstructed graph is correct. Context capture and propagation need to be checked when the graph contains gaps or unexpected relationships.

17. Network and Black-Box Diagnostics

Network diagnostics often concerns devices whose implementation is unavailable to the analyst. Packet captures expose exchanges at a communication boundary. Counters and management objects expose selected state. Active probes test reachability or response time. Routing and flow information provide further context for interpreting application symptoms.

SNMP first appeared as RFC 1067 in August 1988; RFC 1157 followed in May 1990 [134][81]. The protocol defined communication between management applications and agents exposing managed objects. This enabled common inspection across different devices. However, a counter or status value usually identifies only part of the problem. The analyst still needs to connect it to traffic, configuration, topology, and the affected service.

Packet traces can preserve handshakes, retransmissions, resets, and timing relationships. Their interpretation depends on capture location, filters, loss, and clock behavior. Encryption may hide payloads while leaving some boundary information available. A missing packet in one capture does not by itself prove that the packet was never sent or received elsewhere.

Black-box performance analysis uses external observations to constrain possible internal explanations. Aguilera and colleagues examined relationships among message traces to infer request flows and delays

[26]. Such analysis can be useful without source instrumentation, although its conclusions depend on the assumptions used to associate messages and components.

An application timeout may be related to DNS, retransmission, a route change, queueing, or a load balancer. Network analysis therefore requires correlation with other evidence. In cloud systems, virtual networks, service meshes, managed storage, and control planes add further boundaries that may be relevant to the same symptom.

18. Distributed Tracing and Application Performance Management

Distributed tracing associates records with logical operations across process boundaries. A trace view can show where recorded time accumulated and which services participated. Its identifiers, relationships, attributes, and collection policy determine which conclusions can be drawn from that view. A long interval requires further interpretation before it can be assigned to CPU work, waiting, or unobserved activity.

Open-source systems made these methods available beyond the organizations that developed early tracing infrastructure. Twitter released Zipkin in 2012 [83]. Uber's account of Jaeger describes the difficulty of tracing across heterogeneous languages and communication frameworks [84]. Jaeger's CNCF graduation in 2019 provides a documented milestone in the operational adoption of distributed tracing [85].

Traces support investigation of correctness as well as performance. Unexpected repeated operations may indicate retries or duplicate work. Missing operations may indicate a branch or an instrumentation gap. Resource and deployment attributes connect the request to its environment. Metric exemplars can provide links from aggregate observations to particular traces. Each relationship becomes useful when its meaning is preserved.

A trace is a partial execution artifact. Head sampling can omit a request before its outcome is known. Tail sampling requires buffering and may still receive an incomplete trace. Instrumentation does not automatically cover every queue or background task. These limits should remain visible when a waterfall or service graph is used to support a diagnostic explanation.

19. Production Instrumentation

Production instrumentation provides selected observations while a workload continues to run. Dynamic probes allow an analyst to ask additional questions without rebuilding every component. Collection and aggregation replace some operations that would otherwise require process suspension. Their value depends on obtaining useful information with acceptable disturbance.

DTrace's 2004 design described safe dynamic instrumentation and in-kernel aggregation for production systems [27]. ETW uses providers, sessions, buffers, and consumers to control collection [71]. Both distinguish the availability of observation points from the decision to enable them. This permits more detail to be collected when a particular investigation requires it.

Linux provides several related facilities. ftrace supports function and event tracing as well as latency investigation [102]. perf combines hardware and software events with sampling. BPF provides facilities for programmable execution in the kernel [56]. Tools such as BCC and bpftrace apply these facilities to observation [103]. The available interfaces differ, but they allow observations to be selected closer to the relevant execution boundary.

For example, we may need to identify allocation stacks, accumulated waiting time, system-call errors, or a latency distribution at one tracepoint. Filtering and aggregation near the source can reduce the amount exported for later analysis. User-space probes and stack capture add context beyond kernel counters. The resulting artifact should retain which event and population were measured.

Instrumentation can add overhead, change behavior, or expose sensitive state. Verifiers and permission boundaries constrain some operations, but do not eliminate every failure mode. Buffer loss, probe errors, and collection gaps also need to be recorded. Otherwise, apparently quiet output may indicate a failed observation path rather than an absence of relevant activity.

20. Metrics and Alerting

Metrics represent selected properties as numerical observations, commonly stored over time. Aggregation makes them suitable for trends, capacity questions, and alerting. It also removes information about individual events. A metric can identify an interval or population requiring investigation while leaving its mechanism to another artifact.

Host statistics and network counters preceded later time-series platforms. Graphing tools and bounded storage made longer operational histories practical. Systems such as Borgmon and Prometheus supported service monitoring at larger scales. Prometheus combines labeled series, queries, service discovery, and a pull-oriented collection model [36]. Labels determine the populations that can be compared.

The SRE literature uses latency, traffic, errors, and saturation as four starting signals for service monitoring [38]. Service-level indicators and objectives relate such observations to expected user experience. They also make it possible to discuss an error budget over an explicit interval. These concepts organize monitoring, but an individual incident may require additional measurements.

An alert connects a detected condition to a response. Its value depends on whether the condition requires action and whether the associated information helps start an investigation. Repeated alerts for one incident can obscure that information. A silent collection failure can be equally misleading. We therefore distinguish a measured zero from a period for which no valid observations were obtained.

OpenMetrics describes an interoperable exposition format for metric data, including labels, types, and exemplars [91]. OpenTelemetry places metrics alongside other signals [42]. The choice of labels remains an engineering decision. Unbounded request or user identifiers can create excessive series, while removing every detailed identifier prevents useful correlation. Exemplars and links to event records provide one way to connect these different levels of detail.

21. Site Reliability Engineering and Incident Analysis

Large Internet services made production a necessary source of diagnostic evidence. Real workloads, machine diversity, accumulated state, and dependencies could be difficult to reproduce elsewhere. Observation mechanisms had to remain usable under these conditions, including during overload and partial failure.

Site reliability engineering brought monitoring, automation, capacity, releases, incident response, and postmortem analysis into a shared engineering practice. A service investigation could therefore lead to changes in software, deployment procedures, or operational controls. The relevant unit was the service and the organization maintaining it.

A typical incident process confirms the impact, stabilizes the service, collects evidence, compares affected and unaffected conditions, and tests possible explanations. Restoration and diagnosis may proceed on different timescales. A rollback can restore service without identifying the defect. We need to record what changed and preserve artifacts that may still be required for a more detailed analysis.

Postmortem reports preserve knowledge that would otherwise remain with the responders. The Google SRE account emphasizes learning from incidents without treating blame as an explanation [39]. An effective report relates impact, detection, timeline, contributing conditions, interventions, and follow-up work. An attribution to human error leaves further questions about why an action was possible and why its effects were not contained.

The same incident may require immediate alerts, short-term triage, later reconstruction, and longer-term preventive work. Collection and retention policies need to support these different intervals. NIST’s 2025 incident-response guidance uses the CSF 2.0 Functions: Govern, Identify, and Protect support preparation, while Detect, Respond, and Recover organize incident handling. Improvement is included within Identify and informs the wider risk-management process [44].

Ownership, escalation, change records, training, and time for corrective work affect whether an observation becomes an explanation and then an effective change. These organizational relationships are therefore part of software diagnostics. A technically detailed artifact can remain unusable when no one can identify its owner, build, or operational context.

Fault Injection and Diagnostic Readiness

Fault injection can test both software behavior and the means available for diagnosing it. Yuan and colleagues analyzed 198 reported failures in five distributed data-intensive systems in their 2014 study [119]. Their findings emphasized error handling and showed how relatively simple tests could expose many serious failures in that sample. The scope of the sample matters; it does not establish the same proportion for every class of software. Historically, this work connects production incident evidence with tests that exercise exceptional paths during development.

We can extend such an experiment to diagnostic readiness. When a dependency fails, does the application preserve the original error? Can we relate retries to the initiating operation? Are the required dumps or event records collected before the process disappears? Does a collection queue lose precisely the interval of highest load? The test then produces two sets of observations: the behavior of the target system and the behavior of its diagnostic facilities. A recovery mechanism may work while leaving insufficient evidence to explain why it was needed.

PART V

Software Observability

22. Observability in Control Theory and Software

In control theory, observability concerns whether internal state can be distinguished from output observations over time, given the system model and known inputs. Kalman's work on filtering and state-space systems provides part of this mathematical background [57]. His 1963 treatment explicitly develops the relationship between state representations, controllability, and observability [125]. Software practice borrows the idea of inferring hidden behavior, although a changing software system rarely has the fixed and tractable model assumed by a classical observability test.

Operational monitoring commonly starts with selected conditions and measurements. The later software observability movement emphasized investigation of questions that were not all known when instrumentation was introduced. Honeycomb's early writing discussed unknown problems and richly described events [88]. Sridharan's 2018 account examined these ideas in distributed systems [89]. Their emphasis was on what an analyst could investigate using the available data.

The grouping of metrics, logs, and traces became a common way to describe observability tools. However, signal types and observability are different concepts. A memory dump, packet capture, profile, or deployment record may contain the information needed for a particular question. Conversely, a system may emit all three commonly named signals while omitting the identity or relationship needed to distinguish two possible mechanisms.

A dashboard displays selected views of known measurements. Exploratory investigation also needs a way to change the grouping, select another population, inspect particular records, and compare them with changes in the system. These operations resemble some activities of an interactive debugger, but they act on recorded information from a system that cannot generally be suspended as one computation.

By the early 2020s, observability was used for a broad range of monitoring and analysis products. Practitioners also reconsidered how the term had developed [90]. For this history, we distinguish collection, storage, querying, presentation, alerting, and interpretation. All can contribute to observability, but the availability of a backend does not by itself provide a diagnosis.

23. Software Execution Artifacts and Their Relationships

Each artifact preserves selected information and omits other information. A dump provides detailed state within its capture scope. A trace records selected execution relationships. A log reflects the observation points chosen by its producers. A metric summarizes a population, and a profile attributes sampled or instrumented events. We combine these representations when one artifact cannot answer the whole diagnostic question.

Snapshots help reconstruct structures such as objects, threads, modules, and ownership. Histories help reconstruct transitions and their order. Aggregates help compare populations and intervals. Configuration and deployment records identify which system produced these observations. The required combination depends on whether we are investigating state, behavior, change, or a difference between populations.

Correlation requires identifiers with a known scope. Trace identifiers relate spans and associated logs. Build identities relate addresses to symbols. Deployment identities relate behavior to changes. Process and boot identities help distinguish reused numerical identifiers. Incorrect joins can produce a coherent-looking explanation assembled from unrelated evidence.

Artifact completeness also requires investigation. Buffers can overflow, agents can restart, clocks can change, symbols can be unavailable, and retention can remove earlier records. Loss counts, sequence numbers, schema versions, and collection metadata help identify these conditions. Without such information, the analyst may interpret an acquisition problem as software behavior.

Collection has practical limits. More detailed artifacts consume resources and may contain sensitive data. One useful design is to retain broad observations continuously and support more detailed acquisition for a selected problem, population, or interval. The escalation path must be prepared before the original evidence disappears.

Database Provenance and Transaction Histories

Data provenance provides another history of explanation from artifacts. Buneman, Khanna, and Tan distinguished why-provenance and where-provenance in 2001 [116]. The former concerns the source data contributing to a result; the latter concerns the origin of values appearing in it. This is different from recording who captured a dump or transformed a log. Both uses concern origin, but they operate at different levels. When a report contains an incorrect value, we may need the query transformations and contributing records, even if every service completed successfully and every request trace appears normal.

Distributed databases also require evidence about transactions and their relationships. Elle, described by Kingsbury and Alvaro in 2020, inferred isolation anomalies from experimental observations by

constructing and analyzing dependencies between transactions [126]. Its conclusions depend on the available histories and on the information retained by the tested operations. A client-observed history can therefore support a correctness question that ordinary latency metrics do not answer. Finding an anomaly demonstrates a violation under the observed conditions; failing to find one does not prove correctness for all executions.

These examples widen our collection of software execution artifacts. A query result with its derivation and a transaction history with its dependencies can both be diagnostic inputs. We should select them according to the question being asked. A slow operation, an incorrect value, and an invalid ordering may require different artifacts even when they occur in the same database service.

24. OpenTelemetry and Telemetry Standardization

Distributed tracing introduced different APIs, propagation formats, attributes, and exporters. Instrumentation could consequently become tied to one implementation or backend. Standardization addressed the relationships between components that produce, process, and consume telemetry.

OpenTracing and OpenCensus merged into OpenTelemetry in 2019 [40]. The specification's v1.0.0 release on 10 February 2021 marked the Tracing API and SDK as stable [135]; Google Cloud described the milestone in May [41]. CNCF announced OpenTelemetry's graduation in May 2026 [43]. The project provides APIs, SDKs, protocols, collectors, and semantic conventions for telemetry [42]. Project graduation is a governance and maturity milestone; individual signals and implementations can still have different stability levels.

Several architectural changes follow from these shared interfaces:

- Application instrumentation can be separated from the eventual analysis backend.
- Context can be propagated across different languages and frameworks.
- Common attribute definitions can support reusable comparisons and queries.
- Collectors can apply filtering, enrichment, sampling, transformation, and routing policies.
- Producers and consumers can evolve against explicit protocols and schemas.

A conforming record is not automatically informative. A span may have the wrong parent, an unstable name, or insufficient error context. Automatic instrumentation may also omit the application decision that matters to the investigation. Semantic conventions provide shared definitions, while diagnostic requirements determine which additional observations are needed.

Profiling and generative-AI instrumentation extend the range of information considered by the project. The GenAI work described model identity, token use, interactions, and tool activity [53][54]. These records

need their own semantics and capture policies. Model content and tool results may improve reconstruction while also containing sensitive data.

W3C Trace Context defines HTTP headers for interoperable trace identity and trace state [82]. This addresses one boundary in distributed correlation. Correct instrumentation must still propagate that context across asynchronous work and other interfaces, and must distinguish a parent relationship from a more general link. The first W3C Recommendation appeared in February 2020; the November 2021 Recommendation incorporated editorial updates [82].

Semantic conventions define names, units, operation kinds, and attribute meanings [92]. Their stability affects whether telemetry from different services and versions can be compared. A field that changes meaning without changing its name can be more misleading than a missing field. Schema evolution therefore belongs in diagnostic architecture.

Implementation differences remain. Language support can vary, automatic instrumentation has limited knowledge of application intent, and collectors can lose or transform records. OpenTelemetry makes many of these interfaces explicit. The analyst must still identify which implementation and configuration produced the evidence used in a particular case.

Interoperability also supports later analysis. A trace without its context, a dump without matching symbols, or an event without its decoding schema can lose much of its value. Preserving the information needed to interpret an artifact is part of preserving the artifact itself.

25. Continuous Profiling

Continuous profiling collects resource-attribution information over extended production intervals. A profile can then be selected after a problem is reported, even when no analyst was present to start a dedicated profiling session. The recorded workload and capture policy determine which earlier behavior remains available.

Google-Wide Profiling described collection of stack, processor, heap, contention, and kernel information across data-center machines [86]. Profiles could be compared across applications and platforms. This extended performance investigation from a single program run to populations of executions and helped identify recurring costs and regressions.

A trace may identify a slow operation without showing which code consumed its CPU time. A profile can provide that additional view if its workload and time interval are relevant. Conversely, a CPU profile alone does not account for all waiting time. Combining artifacts requires compatible scope and a clear statement of the resource measured.

OpenTelemetry announced support for profiling in March 2024 [131] and a public alpha for the Profiles signal in March 2026 [87]. This is a dated stage in standardization. It should not be interpreted as a claim that every profiling implementation or integration had reached general availability at that point.

Profiles also depend on retained symbols and application identity. Stack names, loaded modules, and allocation information may be sensitive. Sampling rate, event type, access, retention, and tenant separation all affect the artifact's later use. Continuous collection increases the importance of making these choices explicit.

26. Telemetry Collection and Trust

A telemetry pipeline is another software system. It has queues, batches, retries, transformations, storage policies, and resource limits. An investigation can fail because useful evidence arrives too late, is dropped, or cannot be queried at the required detail. We therefore need diagnostics and observability for the collection system as well as for its sources.

Cardinality has different consequences in different representations. A request identifier can help relate detailed event records while creating excessive series if used as a metric label. Removing that identifier entirely can prevent analysis of an individual request. The design must specify where detailed identity is retained and how it is connected to aggregate measurements.

Sampling selects which future investigations remain possible. Head sampling decides before the outcome of a trace is known. Tail sampling uses later information but requires buffering and depends on receiving enough of the trace. Adaptive policies can change the represented population over time. Counts and comparisons must therefore be interpreted with their sampling and loss policies.

Diagnostic artifacts can contain credentials, paths, database statements, customer data, prompts, and arbitrary process memory. Collection policies should specify the information required for analysis, permitted transformations, access, and retention. Redaction also changes evidence. Its location and effect need to be understood when a missing or altered value affects a hypothesis.

Provenance records the origin and transformations of an artifact. Signed syslog addresses some integrity and continuity requirements for message records [100]. More general pipelines also need producer identity, schema and clock information, transformation history, and evidence of loss. These details help an analyst decide how much confidence to place in a derived observation.

PART VI

Diagnostic Knowledge and Automation

27. Pattern-Oriented and Theoretical Software Diagnostics

Diagnostic knowledge can remain tied to particular commands, products, and individual experience. A pattern language provides another representation. We give recurring structures and analysis methods names, describe their contexts, and explain how they can be used together. This allows a diagnostic procedure to be considered above the level of one tool implementation.

Our crash-dump analysis pattern work began in 2006 with the aim of providing a shared vocabulary [46]. The patterns described recurring configurations in memory and methods for interpreting them. Trace analysis patterns followed in 2009, drawing on different tracing environments [47]. The resulting catalogs developed around software execution artifacts and their analysis.

Consider a collection of thread stacks. Different debuggers require different commands to obtain it, but the analyst may look for the same relationships: repeated stacks, waits, exceptions, and dependencies between threads. A named analysis pattern describes what to examine and how to organize it. A problem pattern describes a recurring structural or behavioral condition found through that analysis.

The pattern-oriented approach supports several related activities:

- Naming provides a vocabulary for observations and analysis methods.
- Generalization allows methods to be adapted to other artifacts and platforms.
- Composition relates patterns within a larger analysis process.
- Education connects commands and examples to the diagnostic questions they address.

The Memory Dump Analysis Anthology developed into seventeen volumes in nineteen books through 2025 [48]. It combines case studies, analysis patterns, and related articles across Windows and selected Linux and macOS environments. The trace and log reference organizes a related pattern language for other software execution artifacts [49]. These catalogs provide documented examples from which more general concepts can be developed.

Patterns distinguish kinds of observations that a signature alone may combine. An abnormal value, an unexpected relationship, and an unusual sequence require different analysis. A pattern also suggests what to obtain next. For example, a wait chain can prompt examination of an owner thread, its stack, and the resource preventing its progress.

The same process can influence instrumentation. If an analysis repeatedly requires information that is difficult to obtain, we can introduce an event, identifier, snapshot, or other acquisition method. Diagnostic experience then affects software construction. The new instrumentation still needs to preserve the conditions under which the pattern is meaningful.

The distinction between problem patterns and analysis patterns is important. Recognizing a blocked thread is different from collecting and comparing thread stacks. Several observed patterns may participate in one mechanism, and one pattern may have several possible explanations. Pattern sequences guide investigation without removing the need to establish the mechanism in the particular case.

Advanced Software Diagnostics and Debugging Reference is an umbrella title used for the Memory Dump Analysis Anthology, whose volumes bring together catalogs, examples, and related methods [48][105]. Such references support communication across support and development, provide material for education, and identify operations that may be automated. Their organization can change as new artifacts and analysis requirements appear, so catalog counts should be associated with an edition or dated description.

Python Debugging for AI, Machine Learning, and Cloud Computing applies a pattern-oriented approach to Python and related runtime environments [106]. It distinguishes elementary diagnostic patterns and categories such as analysis, implementation, presentation, architecture, design, and usage. This relates a concrete debugging action to its role in the wider process, including investigations that cross Python and native-code boundaries.

The first edition of Theoretical Software Diagnostics was published in 2016 [132]. The accompanying discussion of diagnostic education and platform-independent principles explains the naming of the program [112]. The current web page is undated, but its text refers to the first edition as forthcoming in September 2016. Its underlying work began earlier. The fourth-edition collection includes articles developed from 2006 onward and groups them into threads of thinking [107]. These cover memory and trace models, software narratology, geometrical and topological approaches, categorical thinking, and the organization of diagnostic analysis. We should therefore distinguish the naming of the program from the earlier development of its concepts.

Theoretical software diagnostics considers what can be generalized across artifacts, tools, and platforms. Filtering a log, grouping a collection of stacks, and comparing memory regions are concrete operations that can also be studied as transformations of software data. Such abstractions help relate methods and identify their assumptions. Some formulations use precise mathematical structures; others use metaphors to suggest questions and methods. The status of each formulation must be stated rather than inferred from its mathematical terminology.

Memory Models and Software Narratives

Let us consider some of the earlier threads of thinking collected in Theoretical Software Diagnostics. A memory dump can be treated as a model of a system whose components and relationships have to be reconstructed. Bytes alone do not identify these relationships. Symbols, types, object layouts, pointers, handles, and ownership information allow us to move to other levels of description. The book also distinguishes an ideal snapshot at an instant from an artifact acquired over an interval [107] (discussions of memory dump categories and configurations). In the latter case, the acquisition process may affect which relationships can be interpreted as simultaneous.

Traces and logs provide sequences of selected observations. Software narratology considers how these sequences describe software activity, including their omissions, ordering, repeated structures, and points of view [107] (discussions of software narratives and higher-order narratives). A trace is not the complete execution that produced it. It has been constructed through instrumentation, filtering, buffering, and presentation. Two traces of the same activity may therefore tell different software stories. This perspective directs attention to the construction of the artifact as well as to the behavior described by its messages.

Categorical and geometrical formulations provide further ways to organize these relationships. We may consider artifacts as objects and selected analysis operations as transformations between them. We may also consider spaces in which processes, runtimes, or diagnostic actions are related [107]. The choice of objects and transformations is part of the model. Calling a construction categorical does not by itself establish the required mathematical properties, and a spatial analogy does not make memory a literal geometrical surface. The historical contribution lies in making such modeling questions explicit and in relating operations that previously appeared as separate platform-specific techniques.

Diagnostic Analysis as an Artifact

In 2011, we introduced the iterative and incremental A.C.P. root cause analysis methodology. Its name refers to artifacts, checklists, and patterns [107] (“A.C.P. Root Cause Analysis Methodology” and its later revision). We obtain software execution artifacts, use checklists to guide analysis, and recognize patterns. The recognized patterns may require another checklist, another analysis operation, or additional artifacts. This makes the investigation a process with feedback. It also explains why a fixed sequence of commands may be insufficient when the available evidence changes the diagnostic question.

The analysis itself produces another artifact. Commands, selected records, interpretations, rejected hypotheses, and requests for additional information form an analysis narrative. Such a narrative can be examined in turn, giving higher-order analysis narratives in the terminology of the book [107] (“Higher-Order Pattern Narratives”). For example, a review may discover that a filter removed the first relevant

event or that a proposed cause was copied from an earlier incident without checking the current artifacts. The subject of this review is the analysis process and its evidence, not only the original software execution.

Diagnostics During Construction and Cloud Analysis

Diagnostics-driven development brings artifact analysis into software construction. The book describes periodic memory dump analysis during development and the construction of trace statements with later pattern-oriented analysis in mind [107] (“Diagnostics-Driven Development”). This can expose structural and behavioral problems before a reported production failure. It can also reveal that the current instrumentation cannot answer a recurring question. Diagnostic requirements then become inputs to implementation: what information should be retained, where it should be captured, and how it will be related to other artifacts.

The proposed System of Cloud Analysis Patterns, or CAPS, extends this thinking to cloud abstractions [107] (“Introducing Methodology and System of Cloud Analysis Patterns (CAPS)”). Nodes, pods, containers, and control components can be considered through spaces of relationships, while memory analysis remains available at lower levels. The correspondences with traditional operating-system concepts are modeling choices. A pod does not become an operating-system process merely because both can be considered resource containers. Keeping the levels distinct allows an investigation to move from a cluster relationship to the processes and memory artifacts that implement it.

Analysis Pattern Networks

Analysis Pattern Networks provide an architectural proposal for composing concrete analysis operations [107] (“General Architecture of Analysis Pattern Networks”). An operation receives an artifact or collection of artifacts and may produce a transformation, a derived artifact, or diagnostic indicators. Subsequent iterations work with the enriched collection. Connections between operations can express known pattern sequences or relationships discovered during analysis. Implementations may use rules, machine learning, human analysis, or transformations. The neural-network terminology is an architectural analogy here; it does not imply that every operation is a learned numerical neuron.

For example, one operation may partition a trace into activities, another may identify repeated failures, and a later operation may compare the relevant activity regions. Retaining the intermediate artifacts makes the resulting explanation inspectable. This also gives a connection to contemporary diagnostic agents: both require a way to organize analysis steps and preserve their results. We should distinguish the earlier proposal from any claim that a particular later agent implements it. Similar organization alone does not establish a direct historical dependency.

Debugging Spaces

The associated 3D and 4D debugging descriptions organize several groups of concepts, including kernel, user, and managed spaces; binary, source, and metacode; and different categories of debugging patterns. Their geometry is conceptual. Debugging Paradigm³ relates live debugging and memory dump analysis, while Debugging Paradigm⁴ incorporates time-travel debugging. Memory-spacetime diagrams place the corresponding spaces at different times [110][111].

Accelerated Windows Debugging⁴ applies these relationships to WinDbg investigations [108]. Accelerated Linux Debugging⁴ includes WinDbg, GDB, LLDB, rr, KDB, and KGDB in its exercises and cross-platform scope [109]. The framework allows live state, saved state, and recorded execution to be considered together. It does not imply that every listed tool provides all three forms of access to every memory space.

A pattern name does not establish a root cause. A familiar configuration may suggest the wrong mechanism, especially when acquisition is incomplete. We therefore need alternative explanations, additional artifacts, and checks of the expected relationships. Theoretical models and spatial metaphors are useful when they improve this process. They are not substitutes for evidence or a claim of universal adoption across software engineering.

28. Model-Based and Causal Diagnosis

Automated diagnosis has a history preceding contemporary machine learning. Model-based approaches represent expected behavior, observations, and assumptions about components. They then identify assumptions whose rejection makes the observations consistent with the model. Reiter's 1987 theory formalized this approach [15]. Its results depend on the supplied model and on which possible faults it represents.

Statistical debugging compares features of successful and failing executions. Predicates, coverage, stacks, or component presence can be used to rank candidates for investigation [28]. Crash populations provide related evidence at a larger scale [66]. Such methods reduce the search space, but an association can result from a shared cause, a downstream effect, or a selection in the collected population.

Causal investigation asks what mechanism could produce the observed relationships. Controlled interventions, rollout comparisons, and fault injection can help test it. Production constraints may prevent some experiments, so we also compare versions, topology, timing, and unaffected populations. The strength of the conclusion should reflect which alternatives these observations can actually exclude.

Bayesian methods represent uncertainty over hypotheses and update it with additional evidence. Dependency graphs constrain possible propagation paths. Change-point methods identify intervals where

behavior changes, and differential analysis compares selected populations. These methods answer related but distinct questions. A confidence value is meaningful only in relation to the model and data used to obtain it.

A practical analysis can combine artifact facts, statistical rankings, known patterns, and experiments. The purpose of automation is to reduce unnecessary search and make relevant relationships available for examination. We still need a traceable connection between an observation, the hypothesis it supports, and the test used to distinguish that hypothesis from alternatives.

29. Log Analysis and AIOps

Operational logs provide another source of data for automated diagnosis. Their volume makes manual inspection difficult, while their messages often preserve information about decisions and execution stages. Automated methods therefore transform and organize logs before attempting to identify significant deviations.

Template extraction separates stable message structure from variable fields. Lou and colleagues used relationships among log parameters and message groups to infer invariants [32]. DeepLog used sequence learning for log anomaly detection and diagnosis [45]. Related methods cluster events, detect changes, and associate observations with dependencies or previous incidents.

AIOps became a general name for several forms of operational automation. We can distinguish the following tasks:

- Reducing duplicate or irrelevant signals.
- Detecting deviations from a baseline.
- Relating events and alerts.
- Comparing observations with changes.
- Classifying failures.
- Ranking possible fault locations.
- Constructing or evaluating causal models.
- Suggesting or executing corrective actions.
- Summarizing the available evidence.

DARPA’s 2016 Cyber Grand Challenge combined automated vulnerability discovery, analysis, and defensive patching within a controlled competition [37]. It demonstrated a form of automated problem-solving loop under defined rules. Applying similar loops in production introduces additional questions about system intent, incomplete evidence, and the consequences of a change.

Machine learning allows some rules to be inferred from observed populations. A model may learn message sequences, incident groupings, metric behavior, or relationships among components. This can reduce manual rule construction. The training data and its labels become part of the diagnostic system and require their own analysis.

Software data changes with versions, workloads, maintenance, and instrumentation. A rare valid operation may appear anomalous, while a repeated failure may become part of a learned baseline. Template extraction can remove a significant parameter or separate equivalent messages. We therefore need to distinguish a change in the system from a change in the model's representation of it.

Reducing repeated triage is a useful result, but performance on historical incident labels does not automatically extend to new mechanisms. Novel incidents may have different evidence or incomplete labels. An evaluation should identify which populations were tested and which parts of the diagnostic process were assessed.

An automated suggestion needs an inspectable basis. The analyst should be able to examine the records, time intervals, changes, and dependencies used to produce it. Observation, inference, and suggested action should remain distinct. Feedback can improve a model, but it should not silently replace the record of what was known during the original investigation.

30. LLM-Assisted Debugging and Diagnostic Agents

Large language models provide an interface to heterogeneous diagnostic information. They can help explain stack output, summarize logs, locate relevant code, propose commands, and relate an incident description to previous material. These capabilities reduce some translation and navigation work. Their output must still be checked against the artifacts and sources used in the investigation.

RCACopilot examined the use of incident information and language models for cause categories and explanations [50]. Roy and colleagues studied agents that could collect additional diagnostic information through tools [51]. TRIANGLE explored multi-agent incident triage [93]. These studies address different stages of investigation and should be evaluated within their stated datasets, tool access, and task definitions.

A diagnostic agent can choose a query, obtain an observation, and use that observation to decide what to inspect next. For the result to be reviewable, the system must retain queries, parameters, returned records, and their relation to its claims. Missing telemetry cannot be supplied by a plausible narrative. A proposed corrective action also requires its own authority and validation.

Several requirements follow from treating the agent's work as a diagnostic analysis artifact:

- Claims refer to available dumps, traces, logs, code, changes, or documentation.
- Observations are distinguished from interpretations and assumptions.
- Commands, filters, time ranges, and returned results are retained.
- Alternative hypotheses receive tests that could reject the leading explanation.
- Missing data, sampling, clock uncertainty, and symbol mismatches remain explicit.
- Actions stay within the permitted scope and account for their possible effects.
- People responsible for the system can review consequential conclusions and changes.

A single expected root-cause label may be inadequate for evaluation. An incident can have several contributing mechanisms, and the available evidence may not support all of them equally. We can also evaluate whether the agent obtains relevant artifacts, identifies gaps, rejects misleading associations, and produces a reproducible investigation. These results are more informative than a fluent final explanation alone.

Conversational debugging brings similar methods into an interactive development environment. Microsoft's ROBIN study used an investigate-and-respond process that collected IDE context before answering [52]. This relates a natural-language question to actual debugging state. It also continues earlier work on question-oriented interfaces, although the method of generating and checking an answer differs.

The role of the model depends on the task. It may organize evidence, suggest a pattern, generate a query, or propose a test. Confidence in the result comes from the resulting evidence and checks. The language model is one component of the diagnostic process and can itself produce errors requiring investigation.

31. Diagnostics of AI and AI Systems

AI systems also require diagnosis. Training can involve accelerators, networks, storage, schedulers, and checkpoints. Inference adds routing, batching, retrieval, caches, and tool use. We distinguish problems in this execution infrastructure from problems in the model's outputs and behavior. A healthy service can still return an incorrect answer, and a correct model can be hosted by a failing service.

Training failures can arise from devices, communication, numerical behavior, data, software, or workload imbalance. Aegis examined fault diagnosis in a production AI training service [94]. Rapidly locating a suspected device can support recovery while the more detailed mechanism remains under investigation. Isolation and explanation are related stages with different information requirements.

Generative-AI telemetry includes model and provider identity, token usage, timing, tool interactions, and selected content or evaluation information [53][54]. These fields help relate a result to the conditions

under which it was produced. Content collection requires a deliberate policy because prompts, responses, and retrieved documents may contain sensitive information.

Reproduction requires more than saving one prompt. Relevant context may include the model version, instructions, retrieval corpus, tool definitions, sampling parameters, and external service state. A random seed is useful only where the implementation provides the corresponding guarantees. Even a carefully preserved interaction may support comparison without guaranteeing an identical rerun.

Agentic behavior adds sequences of decisions, tool results, retries, and changes to stored context. We can analyze this as a software narrative while distinguishing recorded actions from later explanations of those actions. The relevant artifact records what was requested, what information was returned, and what changed in the environment. A generated rationale is another claim to examine rather than direct evidence of a hidden mechanism.

PART VII

Case Studies and Diagnostic Principles

32. Diagnostic Lessons from Software Failures

The following cases illustrate how a failure can involve several levels of a system. We consider the available reports and the diagnostic relationships they describe. A short case description cannot replace the full inquiry, but it can identify a recurring problem in evidence collection, interpretation, or containment.

Therac-25. Leveson and Turner documented six known accidents between 1985 and 1987 and examined software, interfaces, concurrency, safeguards, and investigation [18]. The diagnosis cannot be reduced to one race condition. Error displays, assumptions inherited from earlier machines, and the handling of reports affected how hazardous behavior was recognized and understood. The case relates technical evidence to the surrounding safety and support process.

Ariane 5 Flight 501. The inquiry into the 1996 loss described a numerical conversion exception, reused software operating under a different flight profile, shutdown of redundant units, and diagnostic data interpreted as flight information [19]. The failure mechanism crossed code, exception handling, redundancy, and interface semantics. Identical redundant components did not protect against a shared software assumption.

Mars Climate Orbiter. NASA's Phase I report identified a units mismatch and contributing verification and organizational problems [95]. The interface defect explains an important part of the mechanism. Its diagnosis also required examination of navigation data, anomalous trends, team communication, and the handling of earlier evidence. A familiar summary of the units error should not remove these contributing relationships.

Knight Capital. The SEC order describes the 2012 deployment and control failures that generated unintended orders and losses exceeding $460 million [96]. Inconsistent deployment, obsolete behavior, and inadequate risk controls interacted. Version checks and bounded operational controls are therefore relevant diagnostic and containment mechanisms, alongside analysis of the defective code.

Amazon S3 in February 2017. While debugging slow progress in the S3 billing system, an authorized team member supplied a command input that removed more server capacity than intended [97]. Recovery required subsystem restarts. The post-event changes included constraints on the command and recovery

improvements. The incident shows why the effects of a diagnostic intervention must be considered as part of its design.

Cloudflare in July 2020. An erroneous configuration caused an Atlanta router to advertise routes with increased preference, attracting backbone traffic and overloading the location [98]. The main traffic outage lasted 27 minutes. Subsequent congestion also caused some log loss before full recovery of the telemetry path. The case therefore concerns both a routing mechanism and the integrity of the evidence collected during recovery.

These cases require more than identifying the nearest error message or latest change. We need to relate conditions, state transitions, interfaces, and containment. Reports also show that a diagnostic mechanism can fail or contribute to an incident. Its commands, outputs, and assumptions belong within the scope of analysis.

33. Recurring Principles of Software Diagnostics

Different computing environments require different tools, but several analysis and acquisition requirements recur. We collect them here as principles that can be applied to the artifacts and methods discussed in the preceding chapters.

Capture before interpretation

Execution evidence can disappear when a process exits, a buffer wraps, a container is removed, or a deployment replaces a binary. Acquisition policies must account for this lifetime. We can improve an interpretation later if its evidence survives. We cannot assume that the original state will remain available for another request.

Preserve identity

An artifact needs sufficient identity to associate it with a build, symbols, configuration, workload, and collection method. Numerical precision does not compensate for the wrong identity. An address from one build or a timestamp from an uncertain clock can support an incorrect reconstruction if its context is omitted.

Correlation is not causation

Time proximity and statistical association help select hypotheses. A causal explanation also needs a mechanism and evidence that distinguishes it from alternatives. The transformations used to obtain a ranking or correlation should remain available for inspection.

Shorten the defect-to-detection interval

Checks near invalid transitions can provide more useful evidence than their later consequences. Assertions, sanitizers, schema checks, and resource limits expose particular violations. Their value depends on the property checked and the relationship between that violation and the reported problem.

Combine snapshots and histories

Dumps provide selected state, traces and logs provide selected history, and metrics and profiles provide aggregate views. Source and configuration help interpret the intended computation. We combine these representations through explicit relationships instead of assuming that one can answer every diagnostic question.

Design for unanticipated questions

A fixed monitor checks conditions anticipated during its design. Further investigation may require another grouping, comparison, or level of detail. Context, retained records, and suitable query facilities allow such questions to be asked after a problem appears.

Treat the organization as part of the system

Ownership, escalation, deployment, review, and training influence both failures and their investigation. A diagnosis may therefore lead to changes in code and in the process around it. The analysis should identify the particular relationship involved rather than stopping at a general attribution to an individual or team.

Govern diagnostic data

The information required for diagnosis may also include confidential or personal data. Capture, access, transformation, retention, and deletion policies should reflect this fact. Incident-response guidance similarly relates preparation and technical response to wider risk management [44]. Data protection and evidence preservation need to be considered together.

Keep the reasoning inspectable

An analysis should record its artifacts, transformations, hypotheses, and checks. This permits another analyst to reproduce useful steps and challenge unsupported conclusions. Automation increases the need for this record when a ranking or summary hides many intermediate operations.

Build a diagnostic vocabulary

Named patterns, event definitions, and problem classifications make experience easier to communicate and reuse. Their meaning must remain connected to examples and context. A shared name is useful when analysts can determine what evidence it requires and which questions it leaves open.

Separate observation, inference, and action

A stack contains an observed frame. The claim that this function corrupted memory is an interpretation. Disabling the corresponding feature is an intervention. Keeping these categories separate makes it possible to review the evidence, explanation, and effect of the action independently.

Make missingness explicit

Loss, sampling, unavailable symbols, expired records, and absent fields are part of the available information. We preserve these conditions explicitly. An unknown value should not silently become a zero, a negative result, or evidence of normal behavior.

Design containment as a diagnostic capability

Canaries, limits, circuit breakers, safe modes, and bounded commands reduce the effects of failure. They can also create comparative evidence about the conditions under which behavior changes. Such experiments require interpretation because a successful mitigation may leave the original mechanism unresolved.

Keep raw evidence beneath summaries

A bucket, dashboard, score, or generated narrative directs attention to selected information. The analyst also needs representative records and their provenance. A summary should preserve a path back to the evidence used to construct it.

Treat diagnosability as an architectural quality

Event semantics, identifiers, dump policies, symbol retention, and collection-health measurements can be designed before deployment. These choices affect which later problems can be analyzed. Diagnosability is therefore a property of the software and its supporting process, not only a skill applied after failure.

34. Further Directions in Software Diagnostics

One direction is closer integration of existing artifact types. An investigation may move from a metric to a trace, then to logs, profiles, a memory dump, and deployment records. The integration is useful when the relationships are explicit and can be checked. It should not conceal differences in time, scope, or capture policy.

Recorded execution, processor traces, snapshots, and provenance can make more of the past available for queries. Their coverage and cost differ. Adaptive capture may preserve greater detail around a selected condition, but it cannot recover information that was never recorded. Pre-event history remains an important acquisition problem.

Evidence graphs can represent artifacts, events, states, versions, hypotheses, and interventions. Edges may describe ordering, reference, dependency, or a proposed causal relationship. These meanings should be distinguished. A recorded parent link and an inferred causal edge do not carry the same evidential status, even when a visualization draws them similarly.

Diagnostic agents can operate on these artifacts through queries, comparisons, and targeted acquisition. More capable agents need correspondingly clear action boundaries and records of their work. Reproduction of an investigation requires the data and tool results available at the time, as well as the final explanation.

Instrumentation may also be assisted by compilers, frameworks, and previous incident information. Such assistance can identify missing checks or suggest capture points. Application meaning still determines which distinctions matter. An automatically recorded function boundary may omit the business decision or external condition needed to explain a failure.

Evaluation needs to examine diagnostic results as well as speed. Time to resolution depends on incident mix and organizational conditions. Data volume does not establish observability. A correct incident label does not demonstrate a safe intervention. We can instead examine access to discriminating evidence, reproducibility of analysis, correctness of proposed mechanisms, and the effect of resulting changes.

Conclusion

We have followed the development of software diagnostics through its artifacts and analysis methods. Early checking routines preserved selected execution information. Symbolic debuggers related machine values to program names. Memory dumps separated acquisition from later analysis. Traces, logs, and profiles added views of behavior and resource use. Distributed methods introduced explicit relationships across processes. Formal analysis, patterns, statistical methods, and diagnostic agents provided further ways to organize the investigation.

These methods remain applicable at different levels of contemporary systems. A service with extensive telemetry may still require a memory dump or a packet capture. An automated explanation may require a reduced test case to establish its mechanism. New representations extend the available analysis; they also introduce acquisition, interpretation, and trust requirements of their own.

The recurring process is to obtain relevant artifacts, reconstruct their structures and relationships, consider possible mechanisms, and test the resulting explanations. Missing information and uncertainty remain part of that process. A successful investigation can then contribute a case study, an analysis pattern, an improved check, or a change in the system’s diagnostic architecture.

Software diagnostics, observability, and debugging can therefore be considered as related activities throughout software construction and post-construction. Their history shows how knowledge moves from particular failures to reusable methods. Further development depends on keeping this knowledge connected to the artifacts, assumptions, and experiments from which it was obtained.

Appendix A · Chronology of major developments

The chronology lists selected publications, standards, systems, and incidents. Dates identify the stated milestone and do not imply exclusive priority for the underlying idea.

Table 2. Selected publications, systems, standards, and incidents, with the diagnostic capability each changed.

PERIOD	DEVELOPMENT	DIAGNOSTIC SIGNIFICANCE
Before 1940	Electrical and electromechanical fault isolation	Schematics, test points, substitution, logs, and physical inspection established a cross-layer fault-finding method.
1947	Harvard Mark II moth logged	Iconic physical evidence attached to an operator record; the technical use of bug predated the event [1].
1951	EDSAC program-diagnosis methods published	Checking routines, selective output, test data, and post-mortem reasoning became teachable practice [2][3].
1960–1963	Kalman's state-space work	Filtering, state representation, and observability developed within linear-system theory [57][125].
Early 1960s	PDP-1 interactive culture and DDT variants	Direct inspection, alteration, breakpoints, and rapid reruns shortened the diagnostic loop [58].
1963	EDSAC 2 diagnostic techniques	Diagnostics moved toward source-language interpretation [4].
1967	IBM System/360 operator guidance	Messages, abnormal ends, and dumps became institutional operational evidence [8].
1968	NATO Software Engineering Conference	Failure was framed as an engineering and organizational problem of scale [5].
1969	Interactive LISP debugging system documented	Debugging integrated break packages, inspection, and higher-level program structures [61].
1969	Hoare logic and structured-programming work	Reasoning about correctness complemented empirical debugging [6][7].
1974	Unix system described publicly	Composable tools and process abstractions shaped a diagnostic culture [9].
1976	McCabe complexity measure	Structural complexity became a measurable maintenance and risk signal [11].
1977	Abstract interpretation	Static analysis gained a general theory for safe semantic approximation [77].
1978	Lamport logical clocks; lint report	Distributed ordering and practical static checking advanced in parallel [13][10].
1981	Model-checking workshop formulation	Temporal properties of concurrent control could be checked mechanically; proceedings published in 1982 [78].

PERIOD	DEVELOPMENT	DIAGNOSTIC SIGNIFICANCE
1982	gprof	Sampling and call-graph attribution joined performance diagnosis [12].
1983	Algorithmic Program Debugging	Oracle judgments about computations guided diagnosis in a logic-programming framework [114].
1984	Program slicing	Dependencies were used to select code relevant to variables at a chosen point [113].
1985	Distributed snapshots	Consistent global state could be captured without halting the whole system [14].
1986	GNU Project develops GDB lineage	Portable source-level debugging became a major free-software capability [133].
1987	Reiter's theory of diagnosis	Model-based diagnosis formalized explanations from components and observations [15].
1988	Vector-clock work	Causality and concurrency could be distinguished more precisely [80].
1988	First SNMP specification	RFC 1067 defined communication between management applications and agents [134].
1990	SNMP revision and fuzz testing publications	Network manageability and automated hostile-input discovery broadened diagnostics [81][73].
1992	Purify	Shadow-state instrumentation detected memory errors closer to their origin [20].
1993	Therac-25 analysis	Software safety, UI, concurrency, and organizational controls were analyzed as one system [18].
1996	Ariane 5 Flight 501 report	Reuse, exception policy, common-mode redundancy, and diagnostic data formed a causal chain [19].
1997	Eraser	Dynamic race detection made concurrency discipline observable [21].
1999	Mars Climate Orbiter report	Interface units, verification, trending, and organizational communication became an emblematic systems lesson [95].
2000	QuickCheck	Generated inputs tested executable properties and supplied examples for diagnosis [127].
2001	RFC 3164; statistical inference from code	Syslog practice was documented while code populations became sources of inferred rules [16][79].
2001	Why- and where-provenance	Query results could be explained through contributing data and value origins [116].
2002	Delta debugging, Tarantula, Pinpoint	Failure minimization, ranked localization, and large-service problem determination advanced [22][23][24].

PERIOD	DEVELOPMENT	DIAGNOSTIC SIGNIFICANCE
2003	Magpie and black-box distributed diagnosis	Request reconstruction and boundary evidence addressed multi-service behavior [25][26].
2004	DTrace	Safe, dynamic production instrumentation supported new questions without restarts [27].
2004	Whyline; Magpie request extraction	Behavioral questions and reconstructed requests provided additional ways to organize execution evidence [115][129].
2005	Scalable statistical bug isolation	Deployed executions could rank predicates associated with failure [28].
2006	Crash-dump analysis patterns published	Recurring diagnostic structures began to be codified in a pattern language [46].
2007	Valgrind framework and X-Trace	Dynamic binary instrumentation and pervasive cross-layer tracing matured [30][29].
2008	CHESS and KLEE	Controlled schedules and generated path inputs supported discovery and reproduction of failures [118][128].
2009	RFC 5424; trace-analysis patterns; WER at scale	Structured logging, pattern languages, and fleet-scale crash statistics converged [70][47][66].
2010	Dapper; log-invariant mining; Google-Wide Profiling	Distributed traces, machine log analysis, and continuous profiles became production-scale evidence [31][32][86].
2011	A.C.P. methodology	Artifacts, checklists, and patterns were organized as an iterative diagnostic process [107].
2012	AddressSanitizer and Zipkin	Fast compiler instrumentation and open-source distributed tracing broadened adoption [33][83].
2012	Knight Capital incident	Deployment consistency and automated risk containment became central diagnostic lessons [96].
2012	W3C Navigation Timing	Standardized browser timing exposed stages of the client's navigation experience [120].
2013	The tail at scale	Rare delays became significant in requests depending on many components [123].
2014	Production failure and error-handling study	Analysis of 198 failures in five systems connected incident evidence to exceptional-path tests [119].
2015	Jaeger created at Uber	Microservice scale made end-to-end tracing operational infrastructure [85].
2016	Google SRE book; modern observability movement	Golden signals, SLOs, postmortems, and exploratory production inquiry entered mainstream practice [38][39][88].

PERIOD	DEVELOPMENT	DIAGNOSTIC SIGNIFICANCE
2016	Theoretical Software Diagnostics named	A platform-independent program connected diagnostic patterns with systems thinking and concepts from mathematics and the humanities [112][132].
2017	DeepLog, rr, AWS S3 disruption	Learned log sequences, deployable replay, and safe diagnostic operations highlighted three frontiers [45][75][97].
2017	DWARF Version 5	Debugging information continued to standardize mappings from executable artifacts to program abstractions [122].
2018	Distributed Systems Observability	Monitoring and observability distinctions were synthesized for practitioners [89].
2018	Flight Recorder in OpenJDK 11	JVM event recording became available through the OpenJDK integration specified by JEP 328 [117].
2019	OpenTelemetry formed; Jaeger graduated	Instrumentation APIs began consolidating while tracing reached cloud-native maturity [40][85].
2020	Cloudflare backbone outage postmortem	Topology, configuration change, and network evidence illustrated cross-layer observability [98].
2020	W3C Trace Context; Elle	Propagation headers became a W3C Recommendation; transaction histories supported isolation-anomaly inference [82][126].
2021	Trace Context update and OTel Trace 1.0	A revised W3C Recommendation and stable tracing specification supported interoperable instrumentation [82][135].
2024	Pattern-oriented Python debugging and 4D Windows/Linux debugging	Pattern categories were applied to Python, AI, and cloud workloads; conceptual spaces related live, dump, and time-travel investigation [106][108][109][110][111].
2024	OpenTelemetry GenAI and profiling initiatives; LLM RCA research	Telemetry expanded toward AI semantics, profiles, and tool-using diagnostic agents [53][131][51].
2024	Theoretical Software Diagnostics, fourth edition; ECMA-426	A collected theoretical program and a source map standard documented two different levels of diagnostic representation [107][121].
2025	NIST incident-response revision; AI-training diagnosis; multi-agent triage	Risk management, accelerator-scale isolation, and agentic investigation continued converging [44][94][93].
2026	OTel Profiles alpha and CNCF graduation	Continuous profiles entered the shared signal model as OpenTelemetry reached project maturity [87][43].

Appendix B · Glossary

Abnormal end (ABEND). Termination classified by an operating system or runtime as unsuccessful, often accompanied by codes and dump evidence.

A.C.P. methodology. An iterative diagnostic process organized around artifacts, checklists, and patterns; observations can lead to further analysis or artifact acquisition [107].

Address space. The memory locations and mappings visible to a process or execution context.

Agent. Software that collects, processes, exports, or reasons over diagnostic data; in AI contexts, a model-directed tool user.

Alert. A notification produced when evidence satisfies a rule or learned condition requiring attention.

Algorithmic debugging. Diagnosis organized around a computation model and judgments about the correctness of its parts [114].

Analysis Pattern Network. A proposed architecture that composes analysis operations over artifacts and their successive enrichment [107].

Anomaly. An observation that differs from a baseline or expectation; not automatically a defect or cause.

Application performance management (APM). Tools and practices for measuring and diagnosing application performance and availability.

Baggage. Contextual key-value data propagated across service boundaries alongside trace context.

Breakpoint. A condition that suspends execution at a selected location or event.

Bucketing. Grouping crash reports or incidents by a normalized signature.

Call graph. A representation of calling relationships among functions or procedures.

Cardinality. The number of distinct values or label combinations in a dataset.

Causal inference. Reasoning about whether and how a factor produces an outcome rather than merely correlates with it.

Client-side diagnostics. Analysis of execution and user-visible behavior in a browser or client, using artifacts from that environment.

Context propagation. Passing identifiers and metadata across threads, processes, services, or messages.

Core dump / core image. A preserved subset or image of process memory and execution state, usually captured at failure.

Counterfactual. A claim or test about what would happen if a condition or action were different.

Crash signature. A normalized representation, often based on stack frames and exception data, used to group failures.

Data provenance. Relationships between a result, contributing source data, and the origins of particular values [116].

Debugger. A tool for controlling and inspecting program execution or analyzing preserved execution state.

Debugging paradigm³ / paradigm⁴. Conceptual pattern-oriented frameworks relating debugging spaces and methods; the fourth-dimensional formulation incorporates time-travel debugging.

Delta debugging. Systematic minimization of a failure-inducing input, change, or configuration.

Diagnosability. The degree to which a system's failures can be detected, distinguished, and explained.

Diagnostics. The study and analysis of signs of system structure and behavior; a diagnosis is a conclusion reached for a particular problem and evidence set.

Diagnostics-driven development. The use of diagnostic artifact analysis and its requirements during software construction [107].

Distributed trace. A correlated record of one logical operation across process and service boundaries.

Dump. A captured representation of selected memory or runtime state. Its consistency and completeness depend on how it was acquired.

Dynamic analysis. Analysis based on observations made while a program executes.

Dynamic instrumentation. Attaching or enabling runtime probes without permanently instrumenting every path.

eBPF. A verified execution mechanism in the Linux kernel used for networking, security, tracing, and observability.

Event. A timestamped record that an identified activity or state transition occurred.

Exemplar. A link from an aggregate metric observation to a representative detailed record such as a trace.

Fault. A condition capable of causing an error; terminology varies across dependability traditions.

Fault localization. Ranking or identifying program elements likely related to a failure.

Flame graph. An aggregated stack visualization commonly used for profile analysis.

Flight recorder. A facility that retains selected execution events, often in bounded buffers, for later analysis.

Forensics. Reconstruction and preservation of past activity from artifacts, often with evidentiary controls.

Fuzzing. Automated generation or mutation of inputs to discover failures and invariant violations.

Happened-before. A partial order based on local event order and communication; unrelated events in this order are concurrent within the model.

Heisenbug. A failure whose manifestation changes or disappears when observed or when timing changes.

High cardinality. A field or dimension with many distinct values, useful for discrimination but costly in some stores.

Higher-order analysis narrative. A record produced by examining an earlier diagnostic analysis and the evidence or decisions it contains [107].

Incident. An event or period of degraded service, risk, or operational impact requiring coordinated response.

Instrumentation. Code or mechanisms that expose runtime measurements or events.

Invariant. A property expected to remain true over a defined scope or state transition.

Log. A stored or streamed sequence of events or messages emitted by software or infrastructure.

Memory spacetime. A conceptual representation of kernel, user, and managed memory spaces across time, connecting live state, captured state, and recorded execution.

Metric. A numerical observation or aggregation of a measured property, often recorded as a labeled time series.

Minidump. A compact, selectable subset of crash-dump information designed for efficient capture and transport.

Model checking. Automated exploration of a model's state space against formal properties.

Monitoring. Ongoing measurement and evaluation of known conditions, thresholds, and service health.

Observability. The ability to infer internal behavior from externally available outputs and context.

Observer effect. A change in system behavior caused by the act of measurement or debugging.

OpenTelemetry (OTel). A vendor-neutral project for telemetry APIs, SDKs, protocols, collectors, and semantic conventions.

Pattern-oriented software diagnostics. A method that names recurring problem and analysis structures so diagnostic knowledge can be reused across artifacts, tools, and platforms.

Postmortem. A structured analysis of an incident after stabilization, intended to explain and reduce recurrence.

Profile. A sampled or instrumented attribution of resource use or events to code paths.

Program slice. A representation of program elements relevant to a selected analysis criterion, such as a variable at a program point [113].

Property-based testing. Testing with generated inputs against explicitly stated executable properties [127].

Provenance. Information about the origin and transformations of an artifact or data value; the relevant definition depends on the analysis domain.

Query plan. A database optimizer's representation of how a query will be executed; an analyzed execution can add observed behavior [130].

Record-and-replay. Capturing nondeterministic inputs or execution so an observed run can be reproduced.

Root cause. A mechanism or contributing condition selected as an explanatory stopping point within a stated system boundary and investigation scope.

Sampling. Selecting a subset of events, traces, stacks, or observations according to a policy.

Sanitizer. Compiler- or runtime-assisted dynamic checking for classes of unsafe behavior.

Semantic convention. A shared definition of names, attributes, units, and meanings for telemetry.

Service-level indicator (SLI). A measured quantity representing an aspect of service behavior experienced by users.

Service-level objective (SLO). A target range or threshold for an SLI over a defined period.

Source map. A mapping from positions in a generated artifact to corresponding source positions [121].

Span. A timed operation within a distributed trace, carrying identity, parentage, attributes, and status.

Static analysis. Analysis of program artifacts without executing the target program.

Structured logging. Logging in stable fields or schemas rather than relying only on free-form text.

Symbolication. Mapping runtime addresses to functions, files, lines, types, and other symbolic information.

Symbolic execution. Analysis that represents inputs symbolically and accumulates path conditions from which concrete inputs may be generated [128].

Systematic schedule exploration. Controlled investigation of alternative concurrent execution orders within a supported model and search bound [118].

Telemetry. Measurements and events exported from a system for remote collection or analysis.

Theoretical Software Diagnostics. A research and educational program studying general concepts, models, and methods of software artifact analysis across platforms.

Time series. Observations indexed by time, often numeric and labeled.

Trace. A temporally ordered record of selected execution activity; in distributed systems, a correlated operation tree or graph.

Transaction history. Recorded operations and outcomes used to analyze relationships between transactions, including possible isolation violations [126].

Vector clock. A logical timestamp structure that can distinguish causal order from concurrency.

Watchpoint. A debugger condition that suspends or reports execution when selected data is accessed or changed.

Appendix C · Evidence taxonomy and blind spots

The following matrix relates diagnostic artifacts to the questions they can help answer, their characteristic omissions, and the context needed for interpretation.

Table 3. Complementary diagnostic evidence, characteristic blind spots, and identity requirements.

Evidence	Strongest questions	Characteristic blind spots	Identity / trust requirements
Memory dump / core image	Deep state at one capture point	Prior transitions; omitted pages; external dependencies	Exact binaries, symbols, module list, capture policy
Debugger session	Interactive control and local state	Timing may change; session may be irreproducible	Commands, breakpoints, build, environment, replay trace
Log / structured event	Selected decisions and temporal narrative	Uninstrumented paths; loss; ambiguous wording	Timestamp basis, schema, producer, sequence/loss data
Distributed trace	Recorded request path and latency allocation	Unsampled work; broken propagation; hidden queues	Trace/span IDs, sampling policy, resource and deployment
Metric / time series	Continuous aggregates and trends	Individual context; distribution detail after aggregation	Labels, units, aggregation, scrape/collection health
Profile	Sampled resource attribution to stacks	Intent and non-sampled behavior	Workload window, sampling rate, symbols, event type
Packet / flow record	Boundary-level communication behavior	Host internals; encrypted payload; off-path traffic	Capture point, interface, clocks, filters, topology
Change and deployment record	Version and configuration transitions	Runtime mechanism and hidden manual changes	Commit/build IDs, rollout cohort, actor, timestamps
User or operator report	Impact and experiential context	Precise internal state; selection and recall bias	Environment, time window, workflow, correlation identifiers
Model or anomaly score	Ranked hypotheses or unusual cohorts	Causal proof; behavior outside training assumptions	Model version, features, baseline, confidence, explanation
Browser timing / error record	Client navigation stages and failed client execution	Unrecorded interaction; backend internals; generated code without maps	Client and build identity, navigation context, source maps, timing basis
Query plan / data provenance	Execution choices; contributing records and value origins	Unrecorded workload conditions; incomplete derivation	Query, schema, parameters, data and statistics versions, execution mode
Transaction history / failing schedule	Observed ordering and concurrency anomalies	Unexplored executions; gaps in the observation model	Program and test version, operation semantics, history completeness, schedule
Analysis narrative	How a conclusion was reached and what was checked	Unrecorded decisions; mistaken interpretation of earlier artifacts	Original artifacts, commands, filters, intermediate results, analyst or agent version

References

[1] National Museum of American History. “Log Book With Computer Bug.” Smithsonian Institution. https://americanhistory.si.edu/collections/object/nmah_334663

[2] Gill, S. “The Diagnosis of Mistakes in Programmes on the EDSAC.” Proceedings of the Royal Society A, vol. 206, no. 1087, 1951, pp. 538-554. https://doi.org/10.1098/rspa.1951.0087

[3] Wilkes, M. V., D. J. Wheeler, and S. Gill. The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, 1951. https://archive.org/details/programsforelect00wilk

[4] Barron, D. W., and D. F. Hartley. “Techniques for Program Error Diagnosis on EDSAC 2.” The Computer Journal, vol. 6, no. 1, 1963, pp. 44-49. https://doi.org/10.1093/comjnl/6.1.44

[5] Naur, P., and B. Randell, editors. Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels, 1969. Brian Randell’s report archive. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

[6] Dijkstra, E. W. “Notes on Structured Programming.” EWD249, 1969. E. W. Dijkstra Archive, University of Texas at Austin. https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD249/EWD249.html

[7] Hoare, C. A. R. “An Axiomatic Basis for Computer Programming.” Communications of the ACM, vol. 12, no. 10, 1969, pp. 576-580, 583. https://doi.org/10.1145/363235.363259

[8] IBM. IBM System/360 Operating System: Operator’s Guide. Form C28-6540-5, sixth edition, September 1967. https://bitsavers.trailing-edge.com/pdf/ibm/360/operatingGuide/C28-6540-5_360_operGuide.pdf

[9] Ritchie, D. M., and K. Thompson. “The UNIX Time-Sharing System.” Communications of the ACM, vol. 17, no. 7, 1974, pp. 365-375. https://doi.org/10.1145/361011.361061

[10] Johnson, S. C. “Lint, a C Program Checker.” Bell Laboratories, 26 July 1978. Scanned copy. https://www.silicon-russia.com/wp-content/uploads/2019/02/lint_1978_stephen_johnson.pdf

[11] McCabe, T. J. “A Complexity Measure.” IEEE Transactions on Software Engineering, SE-2, no. 4, 1976, pp. 308-320. https://doi.org/10.1109/TSE.1976.233837

[12] Graham, S. L., P. B. Kessler, and M. K. McKusick. “gprof: A Call Graph Execution Profiler.” Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pp. 120–126. https://doi.org/10.1145/800230.806987

[13] Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System.” Communications of the ACM, vol. 21, no. 7, 1978, pp. 558-565. https://lamport.azurewebsites.net/pubs/time-clocks.pdf

[14] Chandy, K. M., and L. Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems.” ACM Transactions on Computer Systems, vol. 3, no. 1, 1985, pp. 63-75. https://lamport.azurewebsites.net/pubs/chandy.pdf

[15] Reiter, R. “A Theory of Diagnosis from First Principles.” Artificial Intelligence, vol. 32, no. 1, 1987, pp. 57-95. https://doi.org/10.1016/0004-3702(87)90062-2

[16] Lonvick, C. “The BSD syslog Protocol.” RFC 3164, IETF, August 2001. https://www.rfc-editor.org/rfc/rfc3164

[17] Microsoft. “Event Logging” and “Eventlog Key.” Microsoft Learn, accessed 13 September 2026. https://learn.microsoft.com/en-us/windows/win32/eventlog/event-logging; https://learn.microsoft.com/en-us/windows/win32/eventlog/eventlog-key

[18] Leveson, N. G., and C. S. Turner. “An Investigation of the Therac-25 Accidents.” Computer, vol. 26, no. 7, 1993, pp. 18-41. https://doi.org/10.1109/MC.1993.274940; Open copy: https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/therac25.pdf

[19] Ariane 5 Inquiry Board, chaired by J.-L. Lions. Ariane 5 Flight 501 Failure: Report by the Inquiry Board. Paris, 19 July 1996. MIT OpenCourseWare copy. https://ocw.mit.edu/courses/16-355j-software-engineering-concepts-fall-2005/91f1e550b30b00ad797293f430220f18_ari5fail_ful_rep.pdf

[20] Hastings, R., and B. Joyce. “Purify: Fast Detection of Memory Leaks and Access Errors.” Winter USENIX Conference, 1992, pp. 125–136. https://web.stanford.edu/class/cs343/resources/purify.pdf

[21] Savage, S., M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs.” ACM Transactions on Computer Systems, vol. 15, no. 4, 1997, pp. 391–411. https://doi.org/10.1145/265924.265927

[22] Zeller, A., and R. Hildebrandt. “Simplifying and Isolating Failure-Inducing Input.” IEEE Transactions on Software Engineering, vol. 28, no. 2, 2002, pp. 183-200. https://doi.org/10.1109/32.988498; Open copy: https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/delta-debugging.pdf

[23] Jones, J. A., M. J. Harrold, and J. Stasko. “Visualization of Test Information to Assist Fault Localization.” Proceedings of ICSE 2002, pp. 467-477. https://doi.org/10.1145/581339.581397; Open copy: https://faculty.cc.gatech.edu/~stasko/papers/icse02.pdf

[24] Chen, M. Y., E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. “Pinpoint: Problem Determination in Large, Dynamic Internet Services.” Proceedings of DSN 2002, pp. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[25] Barham, P., R. Isaacs, R. Mortier, and D. Narayanan. “Magpie: Online Modelling and Performance-Aware Systems.” 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003. https://www.microsoft.com/en-us/research/wp-content/uploads/2003/05/magpiehotos03.pdf

[26] Aguilera, M. K., J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. “Performance Debugging for Distributed Systems of Black Boxes.” Proceedings of SOSP 2003, pp. 74–89. https://doi.org/10.1145/945445.945454

[27] Cantrill, B. M., M. W. Shapiro, and A. H. Leventhal. “Dynamic Instrumentation of Production Systems.” USENIX Annual Technical Conference, 2004, pp. 15–28. https://www.usenix.org/conference/2004-usenix-annual-technical-conference/dynamic-instrumentation-production-systems

[28] Liblit, B., M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. “Scalable Statistical Bug Isolation.” Proceedings of PLDI 2005. https://doi.org/10.1145/1065010.1065014; Open copy: https://theory.stanford.edu/~aiken/publications/papers/pldi05.pdf

[29] Fonseca, R., G. Porter, R. H. Katz, S. Shenker, and I. Stoica. “X-Trace: A Pervasive Network Tracing Framework.” Proceedings of NSDI 2007, pp. 271–284. https://www.usenix.org/legacy/event/nsdi07/tech/full_papers/fonseca/fonseca.pdf

[30] Nethercote, N., and J. Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.” Proceedings of PLDI 2007, pp. 89–100. https://doi.org/10.1145/1250734.1250746; https://valgrind.org/docs/valgrind2007.pdf

[31] Sigelman, B. H., et al. “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.” Google technical report, 2010. https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/

[32] Lou, J.-G., Q. Fu, S. Yang, Y. Xu, and J. Li. “Mining Invariants from Console Logs for System Problem Detection.” USENIX Annual Technical Conference, 2010. https://www.usenix.org/legacy/event/atc10/tech/full_papers/Lou.pdf

[33] Serebryany, K., D. Bruening, A. Potapenko, and D. Vyukov. “AddressSanitizer: A Fast Address Sanity Checker.” USENIX Annual Technical Conference, 2012, pp. 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[34] Microsoft. “Minidump Files.” Microsoft Learn. Accessed 13 September 2026. https://learn.microsoft.com/en-us/windows/win32/debug/minidump-files

[35] Microsoft. “MiniDumpWriteDump function.” Microsoft Learn. Accessed 13 September 2026. https://learn.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump

[36] Prometheus Authors. “Overview.” Prometheus documentation. Accessed 13 September 2026. https://prometheus.io/docs/introduction/overview/

[37] Defense Advanced Research Projects Agency. “Cyber Grand Challenge.” Accessed 13 September 2026. https://www.darpa.mil/research/programs/cyber-grand-challenge

[38] Ewaschuk, R. “Monitoring Distributed Systems.” Edited by B. Beyer. In B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, editors, Site Reliability Engineering: How Google Runs Production Systems. O’Reilly, 2016. https://sre.google/sre-book/monitoring-distributed-systems/

[39] Lunney, J., and S. Lueder. “Postmortem Culture: Learning from Failure.” Site Reliability Engineering, O’Reilly / Google, 2016. https://sre.google/sre-book/postmortem-culture/

[40] Sigelman, B., and M. McLean. “OpenTelemetry: The Merger of OpenCensus and OpenTracing.” Google Open Source Blog, 21 May 2019. https://opensource.googleblog.com/2019/05/opentelemetry-merger-of-opencensus-and.html

[41] Brier, J., and E. Ita. “OpenTelemetry Trace 1.0 Is Now Available.” Google Cloud Blog, 3 May 2021. https://cloud.google.com/blog/products/operations/opentelemetry-specification-enables-standardized-tracing

[42] OpenTelemetry Authors. “What Is OpenTelemetry?” and “Signals.” OpenTelemetry documentation. Accessed 13 September 2026. https://opentelemetry.io/docs/what-is-opentelemetry/; https://opentelemetry.io/docs/concepts/signals/

[43] Cloud Native Computing Foundation. “OpenTelemetry’s Graduation.” 21 May 2026. https://www.cncf.io/announcements/2026/05/21/cloud-native-computing-foundation-announces-opentelemetrys-graduation-solidifying-status-as-the-de-facto-observability-standard/

[44] Nelson, A., S. Rekhi, M. Souppaya, and K. Scarfone. Incident Response Recommendations and Considerations for Cybersecurity Risk Management: A CSF 2.0 Community Profile. NIST SP 800-61 Rev. 3, April 2025. https://doi.org/10.6028/NIST.SP.800-61r3

[45] Du, M., F. Li, G. Zheng, and V. Srikumar. “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning.” Proceedings of CCS 2017. https://doi.org/10.1145/3133956.3134015; Open copy: https://users.cs.utah.edu/~lifeifei/papers/deeplog.pdf

[46] Vostokov, D. “Crash Dump Analysis Patterns (Part 1).” Software Diagnostics Library, 30 October 2006. https://www.dumpanalysis.org/blog/index.php/2006/10/30/crash-dump-analysis-patterns-part-1/

[47] Vostokov, D. “Trace Analysis Patterns (Part 1).” Software Diagnostics Library, 28 April 2009. https://www.dumpanalysis.org/blog/index.php/2009/04/28/trace-analysis-patterns-part-1/

[48] Vostokov, D. Memory Dump Analysis Anthology, 17 volumes in 19 books. OpenTask, 2008–2025. Volume-set listing, accessed 13 September 2026; also described under the umbrella title in [105]. https://www.patterndiagnostics.com/mdaa-volumes

[49] Vostokov, D. Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection. 5th ed., OpenTask, 2023. https://www.dumpanalysis.org/trace-log-analysis-pattern-reference

[50] Chen, Y., et al. “Automatic Root Cause Analysis via Large Language Models for Cloud Incidents.” EuroSys 2024, pp. 674–688. https://doi.org/10.1145/3627703.3629553

[51] Roy, D., et al. “Exploring LLM-based Agents for Root Cause Analysis.” arXiv:2403.04123, 2024. https://arxiv.org/abs/2403.04123

[52] Bajpai, Y., B. Chopra, P. Biyani, C. Aslan, S. Gulwani, D. Coleman, C. Parnin, A. Radhakrishna, and G. Soares. “Let’s Fix this Together: Conversational Debugging with GitHub Copilot.” IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), August 2024. https://www.microsoft.com/en-us/research/publication/lets-fix-this-together-conversational-debugging-with-github-copilot/

[53] Robbins, D., and L. Molkova. “OpenTelemetry for Generative AI.” OpenTelemetry Blog, 5 December 2024. https://opentelemetry.io/blog/2024/otel-generative-ai/

[54] Newton-King, J. “Inside the LLM Call: GenAI Observability with OpenTelemetry.” OpenTelemetry Blog, 14 May 2026. https://opentelemetry.io/blog/2026/genai-observability/

[55] Brooks, F. P., Jr. “No Silver Bullet: Essence and Accidents of Software Engineering.” Technical Report TR86-020, Department of Computer Science, University of North Carolina at Chapel Hill, September 1986. https://www.cs.unc.edu/techreports/86-020.pdf

[56] Linux Kernel Project. “BPF Documentation.” Accessed 13 September 2026. https://docs.kernel.org/bpf/

[57] Kalman, R. E. “A New Approach to Linear Filtering and Prediction Problems.” Journal of Basic Engineering, vol. 82, no. 1, 1960, pp. 35-45. https://doi.org/10.1115/1.3662552

[58] Computer History Museum. “Guide to the Collection of Digital Equipment Corporation PDP-1 Computer Materials.” Collection X3602.2006. https://www.computerhistory.org/pdp-1/_media/pdf/102660913.PDP_1.pdf

[59] Digital Equipment Corporation. PDP-8/I DDT: Dynamic Debugging Technique, Programmer’s Reference Manual. DEC-08-CDDB-D, fourth printing, August 1969. https://bitsavers.org/pdf/dec/pdp8/software/DEC-08-CDDB-D_DDT_RefMan.pdf

[60] Digital Research. CP/M Operating System Manual. Digital Research, July 1982. Chapter 4, “CP/M Dynamic Debugging Tool.” https://www.bitsavers.org/pdf/digitalResearch/cpm/CPM_Operating_System_Manual_Jul82.pdf

[61] Quam, L. H. “A LISP Debugging System.” Stanford Artificial Intelligence Laboratory, 18 March 1969. https://softwarepreservation.computerhistory.org/LISP/stanford/Quam-Debugger-19690318.pdf

[62] GNU Project. “GDB: The GNU Project Debugger.” Accessed 13 September 2026. https://www.gnu.org/software/gdb/

[63] Stallman, R. M., R. Pesch, S. Shebs, and contributors. Debugging with GDB: The GNU Source-Level Debugger. Tenth edition, GNU Project / Free Software Foundation, online manual accessed 13 September 2026. https://sourceware.org/gdb/current/onlinedocs/gdb.html/

[64] Microsoft. “Debug Adapter Protocol.” Accessed 13 September 2026. https://microsoft.github.io/debug-adapter-protocol/

[65] Linux man-pages contributors. “core(5) — core dump file.” Linux man-pages, accessed 13 September 2026. https://man7.org/linux/man-pages/man5/core.5.html

[66] Glerum, K., et al. “Debugging in the (Very) Large: Ten Years of Implementation and Experience.” Proceedings of SOSP 2009, pp. 103–116. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/sosp153-glerum-web.pdf

[67] Breakpad contributors. “Breakpad Processor Library.” Repository revision a9df3bd99cf8, accessed 13 September 2026. https://chromium.googlesource.com/breakpad/breakpad/+/a9df3bd99cf8b863a92502ef906b5ddc399c2c9a/docs/processor_design.md

[68] Crashpad contributors. “Crashpad Overview Design.” Repository revision db44314646cb, accessed 13 September 2026. https://chromium.googlesource.com/crashpad/crashpad/+/db44314646cbd0825a73b58dd2b7b5f4faca64a7/doc/overview_design.md

[69] Apple. “Examining the fields in a crash report.” Apple Developer Documentation, accessed 13 September 2026. https://developer.apple.com/documentation/xcode/examining-the-fields-in-a-crash-report

[70] Gerhards, R. “The Syslog Protocol.” RFC 5424, IETF, 2009. https://www.rfc-editor.org/rfc/rfc5424

[71] Microsoft. “About Event Tracing.” Microsoft Learn. Accessed 13 September 2026. https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing

[72] Fu, Q., J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. “Where Do Developers Log? An Empirical Study on Logging Practices in Industry.” Companion Proceedings of ICSE 2014, SEIP track, pp. 24–33. https://doi.org/10.1145/2591062.2591175

[73] Miller, B. P., L. Fredriksen, and B. So. “An Empirical Study of the Reliability of UNIX Utilities.” Communications of the ACM, vol. 33, no. 12, 1990, pp. 32-44. https://doi.org/10.1145/96267.96279

[74] LLVM Project. “LibFuzzer — a library for coverage-guided fuzz testing.” Accessed 13 September 2026. https://llvm.org/docs/LibFuzzer.html

[75] O’Callahan, R., C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. “Engineering Record and Replay for Deployability.” USENIX ATC 2017, pp. 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[76] Microsoft. “Time Travel Debugging Overview.” Microsoft Learn. Accessed 13 September 2026. https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview

[77] Cousot, P., and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” POPL 1977, pp. 238-252. https://doi.org/10.1145/512950.512973

[78] Clarke, E. M., and E. A. Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic.” In D. Kozen, editor, Logics of Programs, workshop held in 1981. Lecture Notes in Computer Science, vol. 131, Springer, 1982, pp. 52–71. https://doi.org/10.1007/BFb0025774

[79] Engler, D., D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. “Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code.” SOSP 2001, pp. 57–72. https://doi.org/10.1145/502034.502041

[80] Fidge, C. J. “Timestamps in Message-Passing Systems That Preserve the Partial Ordering.” Proceedings of the 11th Australian Computer Science Conference, 1988, pp. 56-66. Course-hosted scan, accessed 13 September 2026. https://classpages.cselabs.umn.edu/Spring-2018/csci8980/Papers/Foundations/fidge.pdf

[81] Case, J., M. Fedor, M. Schoffstall, and J. Davin. “A Simple Network Management Protocol (SNMP).” RFC 1157, IETF, May 1990. https://www.rfc-editor.org/rfc/rfc1157

[82] W3C Distributed Tracing Working Group. “Trace Context.” W3C Recommendation, 6 February 2020; updated Recommendation, 23 November 2021. https://www.w3.org/TR/2021/REC-trace-context-1-20211123/

[83] Aniszczyk, C. “Distributed Systems Tracing with Zipkin.” Twitter Engineering, 7 June 2012. https://blog.x.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin

[84] Shkuro, Y. “Evolving Distributed Tracing at Uber Engineering.” Uber Engineering, 2 February 2017. https://www.uber.com/us/en/blog/distributed-tracing/

[85] Cloud Native Computing Foundation. “Cloud Native Computing Foundation Announces Jaeger Graduation.” 31 October 2019. https://www.cncf.io/announcements/2019/10/31/cloud-native-computing-foundation-announces-jaeger-graduation/

[86] Ren, G., E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. “Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers.” IEEE Micro, vol. 30, no. 4, 2010, pp. 65–79. https://doi.org/10.1109/MM.2010.68; Open copy: https://research.google/pubs/google-wide-profiling-a-continuous-profiling-infrastructure-for-data-centers/

[87] Alexandrov, A., I. Anjo, F. Geisendörfer, C. Kalkanis, F. Lehner, and D. Mathieu. “OpenTelemetry Profiles Enters Public Alpha.” OpenTelemetry Blog, 26 March 2026. https://opentelemetry.io/blog/2026/profiles-alpha/

[88] Majors, C. “Why Honeycomb? Black Swans, Unknown-Unknowns, and the Glorious Future of Doom.” Honeycomb, October 2016; updated 1 April 2022. https://www.honeycomb.io/blog/why-honeycomb-black-swans-unknown-unknowns-and-the-glorious-future-of-doom

[89] Sridharan, C. Distributed Systems Observability: A Guide to Building Robust Systems. O’Reilly Media, 2018. https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/

[90] Majors, C. “Observability: The 5-Year Retrospective.” The New Stack, 31 August 2021; republished by Honeycomb, 23 September 2021. https://thenewstack.io/observability-the-5-year-retrospective/; https://www.honeycomb.io/blog/observability-5-year-retrospective

[91] OpenMetrics contributors. “OpenMetrics, a Cloud-Native, Highly Scalable Metrics Protocol.” Version 1.0.0, repository tag v1.0.0. https://github.com/prometheus/OpenMetrics/blob/v1.0.0/specification/OpenMetrics.md

[92] OpenTelemetry Authors. “Semantic Conventions.” OpenTelemetry documentation, accessed 12 September 2026. https://opentelemetry.io/docs/specs/semconv/

[93] Yu, Z., et al. “Triangle: Empowering Incident Triage with Multi-Agent.” Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2025, pp. 674–686. https://doi.org/10.1109/ASE63991.2025.00062; https://www.microsoft.com/en-us/research/wp-content/uploads/2025/02/TRIANGLE_ASE25.pdf

[94] Dong, J., et al. “Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production.” NSDI 2025. https://www.usenix.org/system/files/nsdi25-dong.pdf

[95] Mars Climate Orbiter Mishap Investigation Board. Phase I Report. NASA, 10 November 1999. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[96] U.S. Securities and Exchange Commission. In the Matter of Knight Capital Americas LLC, Release No. 70694, 16 October 2013. https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf

[97] Amazon Web Services. “Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.” 28 February 2017. https://aws.amazon.com/message/41926/

[98] Graham-Cumming, J. “Cloudflare Outage on July 17, 2020.” Cloudflare Blog, 18 July 2020. https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/

[99] Apple. “Generating Log Messages from Your Code.” Apple Developer Documentation, accessed 13 September 2026. https://developer.apple.com/documentation/os/generating-log-messages-from-your-code

[100] Kelsey, J., J. Callas, and A. Clemm. “Signed Syslog Messages.” RFC 5848, IETF, 2010. https://www.rfc-editor.org/rfc/rfc5848

[101] Microsoft. “Instrumenting Your Code with ETW.” Microsoft Learn. Accessed 13 September 2026. https://learn.microsoft.com/en-us/windows-hardware/test/weg/instrumenting-your-code-with-etw

[102] Rostedt, S. “ftrace — Function Tracer.” Linux kernel documentation, accessed 13 September 2026. https://docs.kernel.org/trace/ftrace.html

[103] Gregg, B. “BPF Performance Tools.” USENIX LISA 2019. https://www.usenix.org/conference/lisa19/presentation/gregg-bpf

[104] Linux Kernel Project. “Perf events and tool security.” Accessed 13 September 2026. https://docs.kernel.org/admin-guide/perf-security.html

[105] Software Diagnostics and Observability Institute. “Advanced Software Diagnostics and Debugging Reference.” Description of the Memory Dump Analysis Anthology (Diagnomicon), listed in [48]. Accessed 13 September 2026. https://www.dumpanalysis.org/advanced-software-debugging-reference

[106] Vostokov, D. Python Debugging for AI, Machine Learning, and Cloud Computing: A Pattern-Oriented Approach. Apress, 2024. https://doi.org/10.1007/978-1-4842-9745-2

[107] Vostokov, D. Theoretical Software Diagnostics: Collected Articles. Fourth edition. OpenTask, 2024. https://www.dumpanalysis.org/theoretical-software-diagnostics-book

[108] Vostokov, D., and Software Diagnostics Services. Accelerated Windows Debugging[4]: Training Course Transcript and WinDbg Practice Exercises. Fourth edition. OpenTask, February 2024. https://www.patterndiagnostics.com/accelerated-windows-debugging-book

[109] Vostokov, D., and Software Diagnostics Services. Accelerated Linux Debugging[4]: Training Course Transcript with WinDbg, GDB, LLDB, rr, KDB, KGDB Practice Exercises. OpenTask, July 2024. https://www.patterndiagnostics.com/accelerated-linux-debugging-book

[110] Vostokov, D., and Software Diagnostics Services. “Accelerated Windows Debugging 4D Slides.” 2024. https://www.patterndiagnostics.com/Training/Accelerated-Windows-Debugging-4D-Slides.pdf

[111] Vostokov, D., and Software Diagnostics Services. “Accelerated Linux Debugging 4D Slides.” 2024. https://www.patterndiagnostics.com/Training/Accelerated-Linux-Debugging-4D-Slides.pdf

[112] Software Diagnostics and Observability Institute. “Theoretical Software Diagnostics and Education.” Undated page referring to the first edition as forthcoming in September 2016; accessed 13 September 2026. https://www.dumpanalysis.org/theoretical-software-diagnostics

[113] Weiser, M. “Program Slicing.” IEEE Transactions on Software Engineering, vol. SE-10, no. 4, 1984, pp. 352–357. https://doi.org/10.1109/TSE.1984.5010248

[114] Shapiro, E. Y. Algorithmic Program Debugging. ACM Distinguished Dissertations series. MIT Press, 14 April 1983. ISBN 978-0-262-19218-7. https://mitpress.mit.edu/9780262192187/algorithmic-program-debugging/; Bibliographic record: https://books.google.com/books?id=Xm-HQgAACAAJ

[115] Ko, A. J., and B. A. Myers. “Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior.” Proceedings of CHI 2004, pp. 151–158. https://doi.org/10.1145/985692.985712; Open copy: https://faculty.washington.edu/ajko/papers/Ko2004Whyline.pdf

[116] Buneman, P., S. Khanna, and W.-C. Tan. “Why and Where: A Characterization of Data Provenance.” Database Theory — ICDT 2001, pp. 316–330. https://doi.org/10.1007/3-540-44503-X_20

[117] OpenJDK. “JEP 328: Flight Recorder.” Delivered in JDK 11, 2018. https://openjdk.org/jeps/328

[118] Musuvathi, M., S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. “Finding and Reproducing Heisenbugs in Concurrent Programs.” Proceedings of OSDI 2008, pp. 267–280. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

[119] Yuan, D., et al. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.” Proceedings of OSDI 2014, pp. 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[120] W3C Web Performance Working Group. “Navigation Timing.” W3C Recommendation, 17 December 2012. https://www.w3.org/TR/2012/REC-navigation-timing-20121217/

[121] Ecma International. Source Map Format Specification. ECMA-426, first edition, December 2024. https://ecma-international.org/publications-and-standards/standards/ecma-426/

[122] DWARF Debugging Information Format Committee. DWARF Debugging Information Format, Version 5. 13 February 2017. https://dwarfstd.org/doc/DWARF5.pdf

[123] Dean, J., and L. A. Barroso. “The Tail at Scale.” Communications of the ACM, vol. 56, no. 2, 2013, pp. 74–80. https://doi.org/10.1145/2408776.2408794; Open copy: https://research.google/pubs/the-tail-at-scale/

[124] Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing.” IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004, pp. 11–33. https://doi.org/10.1109/TDSC.2004.2

[125] Kalman, R. E. “Mathematical Description of Linear Dynamical Systems.” Journal of the Society for Industrial and Applied Mathematics, Series A: Control, vol. 1, no. 2, 1963, pp. 152–192. https://doi.org/10.1137/0301010

[126] Kingsbury, K., and P. Alvaro. “Elle: Inferring Isolation Anomalies from Experimental Observations.” Proceedings of the VLDB Endowment, vol. 14, no. 3, 2020, pp. 268–280. https://doi.org/10.14778/3430915.3430918; Author preprint: https://arxiv.org/abs/2003.10554

[127] Claessen, K., and J. Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.” Proceedings of ICFP 2000, pp. 268–279. https://doi.org/10.1145/351240.351266

[128] Cadar, C., D. Dunbar, and D. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” Proceedings of OSDI 2008, pp. 209–224. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf

[129] Barham, P., A. Donnelly, R. Isaacs, and R. Mortier. “Using Magpie for Request Extraction and Workload Modelling.” Proceedings of OSDI 2004, pp. 259–272. https://www.usenix.org/legacy/events/osdi04/tech/full_papers/barham/barham.pdf

[130] PostgreSQL Global Development Group. “Using EXPLAIN.” PostgreSQL 18 Documentation, section 14.1, accessed 12 September 2026. https://www.postgresql.org/docs/18/using-explain.html

[131] Parker, A. “OpenTelemetry Announces Support for Profiling.” OpenTelemetry Blog, 19 March 2024. https://opentelemetry.io/blog/2024/profiling/

[132] Vostokov, D., and Software Diagnostics Institute. Theoretical Software Diagnostics: Collected Articles. First edition, OpenTask, 2016. ISBN 978-1-908043-98-6. Bibliographic record: https://books.google.com/books?id=RJEMMQAACAAJ

[133] Free Software Foundation. “GNU Software Available Now: GDB.” GNU’s Bulletin, vol. 1, no. 2, January 1987, p. 6. https://www.gnu.org/bulletins/bull2.txt

[134] Case, J., M. Fedor, M. Schoffstall, and J. Davin. “A Simple Network Management Protocol.” RFC 1067, IETF, August 1988. https://www.rfc-editor.org/rfc/rfc1067

[135] OpenTelemetry contributors. “Release v1.0.0.” OpenTelemetry Specification, 10 February 2021. https://github.com/open-telemetry/opentelemetry-specification/releases/tag/v1.0.0

Version 10: reference corrections and supporting sources checked 13 September 2026. Living documentation carries an access date or a fixed revision; dated publications retain their publication details. Earlier source checks were made on 30 July and 12 September 2026.