

Forms of Computational Death



Based on Dmitry Vostokov's ideas
in pattern-oriented diagnostics

DumpAnalysis.org

Forms of Computational Death

Idea 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 4 • October 2026

Abstract

A process may terminate while its service continues. A service may stop doing useful work while all its processes remain present. A computation may be recoverable after a crash, and its history may become unrecoverable after an apparently successful restart. We consider these situations as different forms of computational death. Starting from the distinction between crashes and hangs, we extend the analysis to progress, deadlines, state, identity, dependencies, and diagnostic evidence. We propose a Computational Death Profile and a candidate Death Boundary analysis pattern. Both connect termination and recovery decisions to memory dump analysis patterns and trace and log analysis patterns. We also revisit the four causes of crash dumps to distinguish the original failure, the terminating action, and the purpose of collecting diagnostic evidence. The contribution is a diagnostic synthesis with explicit conditions for its use and subsequent validation.

1 Computation and its endings

We often say that a process died, a service is dead, or a machine is no longer alive. Such descriptions are convenient when reporting a problem. They become less convenient when we start analyzing it. The process may still be in the process list. Its threads may consume CPU time. Its log may continue to grow. Nevertheless, the operation for which the process exists is no longer performed.

Ricard Solé’s *Endless Endings: The Many Forms of Death and What They Teach Us About Life* provides a useful starting point [1]. Its published description and contents extend the discussion to machines, languages, and information. In an accompanying publisher interview, Solé describes how complex systems can lose the organization that sustained them [2]. This suggests a question for software diagnostics: what, precisely, has ended when we say that a computation has died?

In our earlier discussion of crashes and hangs, we considered both as manifestations of functional failure and distinguished system failures from component failures [3, “Crashes and Hangs Differentiated”]. We also emphasized the observer’s perspective. A component may disappear from one view and remain visible in another. Here we develop that distinction further. We separate the ending of an execution from the loss of useful progress, the loss of an acceptable continuation, and the loss of evidence about what happened.

The word death is used as a technical metaphor. It does not require a computer to be an organism or an AI system to have subjective experience. Its value depends on the distinctions it helps us make. We retain a distinction only if it changes what we inspect, what we infer, or what we

preserve before intervention. This follows our use of metaphorical bijectionism as a method of inquiry [3, “Metaphorical Bijectionism”]. The mapping is a source of diagnostic questions; the evidence still has to answer them.

An ordinary successful completion is also an ending. A batch process that produces its required result and exits has finished its work. We should therefore distinguish an execution ending from a failure to fulfill an outstanding obligation. The taxonomy of faults, errors, and failures remains applicable: a fault can produce an erroneous internal state, and that state may lead to a deviation in delivered service [5]. Computational death adds a viewpoint on continuation and its loss. It does not supply a new cause for every existing failure.

2 A Computational Death Profile

Before classifying a death, we identify what is supposed to continue. A thread, a process incarnation, a user session, a transaction, an agent task, and a replicated service have different continuation requirements. A restarted process is a new process incarnation even when it loads the same executable. A service may nevertheless preserve its identity across that replacement. Conversely, the original process may remain alive after it has irretrievably lost a particular request.

We propose to attach five qualifiers to a computational death judgment. The entity specifies the scope. The obligation specifies the work or property that must continue. The horizon specifies the relevant deadline or observation interval. The continuity criterion states what may change while we still accept the result as a continuation. The recovery assumptions identify available checkpoints, replicas, inputs, dependencies, and permitted interventions.

For example, “the service is dead” becomes “the current worker cannot complete accepted request 417 within its five-second deadline; restarting the worker can restore service for later requests, but no permitted recovery can satisfy that expired deadline.” A second judgment may concern the same worker’s state: its queue is still inspectable in a memory dump. A third may concern the service as a whole: another worker is completing requests. These judgments are compatible because they concern different entities and obligations.

Under a specified continuation contract, we call a loss irreversible when no recovery path admitted by that contract can restore an acceptable continuation. This is a model-relative definition. Changing the deadline, obtaining a missing key, finding a replica, or permitting reconstruction changes the recovery assumptions. An investigation that has not established either a recovery path or its impossibility should record recoverability as unknown. Failure to find a backup is weaker evidence than a verified inventory showing that every usable copy has been lost.

The Computational Death Profile records the following dimensions separately. It is a proposed diagnostic representation, not a numerical severity score.

Table 1. Dimensions of the proposed Computational Death Profile

Dimension	Question	Evidence to examine
Execution	Does this incarnation still execute, or has it terminated?	Exit records, thread states, scheduler observations, runtime state
Progress	Does execution advance the specified obligation?	Completion counters, queue movement, state transitions, repeated dumps
Service	Is the required result available and valid for the intended consumer?	End-to-end outcomes, response contents, deadlines, client evidence
Continuity	Can an acceptable continuation be obtained under the stated recovery assumptions?	Checkpoints, replicas, replay inputs, dependency and identity requirements
Substrate	Is the required execution environment usable or replaceable?	Hardware state, runtime availability, device-specific dependencies
Evidence	Can the relevant history and state still be reconstructed?	Dumps, retained logs, symbols, schemas, configuration and provenance

For each dimension, we record a finding, its time interval, its evidence, and its uncertainty. We may use terms such as present, impaired, ended, recoverable, unrecoverable, and unknown, as appropriate to the dimension. Adding these values together would hide the distinctions the profile was introduced to preserve.

A heartbeat supports a claim about the code path that produces the heartbeat. It does not establish progress of an unrelated request. Similarly, a CPU counter establishes that some execution consumed time; the relationship to useful progress must be demonstrated. The formal distinction between safety and liveness provides a background for this reasoning [6]. A finite interval without progress does not, by itself, establish permanent nonprogress. A violated finite deadline, however, can establish the loss of that particular timely outcome.

3 Forms of computational death

The forms below are overlapping diagnostic viewpoints. Some describe an ending, some describe mechanisms that prevent continuation, and others describe what is lost. A resource leak may eventually produce a crash, a restart may restore the service, and the same restart may destroy the remaining evidence of the leak. We should be able to describe that sequence without forcing it into one category.

3.1 Completion and deliberate termination

A program can terminate because its work is complete. It can also be terminated to replace an instance, cancel obsolete work, remove a compromised component, or restore a larger system. The death of one incarnation may be part of the continuing operation of another entity.

Crash-only software makes termination and restart part of the design of recoverable services [8]. Software rejuvenation uses proactive restart to counter problems associated with software aging

[9]. These are established precedents for treating local endings as part of continued service. Our diagnostic question is whether the required state and obligations survived the particular termination. A successful restart command does not answer that question.

In a trace, we look for the relationship between the termination request, work draining or cancellation, persistence, exit, and subsequent admission of work. Last Activity and Incomplete History help organize this evidence [4]. An exit code may be consistent with a planned shutdown while an accepted request remains unfinished. The process-level ending can be expected, and the request-level loss can still be a failure.

3.2 Abrupt termination

A process may end after an unhandled exception, an explicit termination request, or an action by a supervising environment. A system may stop after a fatal kernel failure or a loss of power. These events do not have identical diagnostic consequences. In particular, the event that ends execution may occur after the original corruption or resource failure.

Invalid Pointer, Dynamic Memory Corruption, and Lateral Damage are relevant memory dump analysis patterns when the observed failure is separated from the earlier defect [16]. Last Activity identifies a relevant neighborhood in a trace, but proximity to termination does not establish causation. We need an explanation connecting the earlier state, the failing operation, and the termination mechanism.

A watchdog termination introduces an additional causal layer. First, the workload stops satisfying a monitored condition. Later, the watchdog acts. The terminating agent explains why execution ended at that time; it may not explain why useful work stopped earlier. Recording both events prevents the final action from displacing the original problem.

3.3 Deadlock and blocked continuation

In a deadlock, the relevant activities cannot make progress because their dependencies cannot be satisfied under the modeled execution rules. For a simple case involving exclusive locks, a cycle of ownership and waiting can establish this condition. More general waits require their own semantics. A list of waiting threads is not sufficient evidence of deadlock.

Our Deadlock and Wait Chain patterns provide the structural starting points [17, 18]. We inspect what each blocked activity needs, who can supply it, and whether that supplier can proceed. A wait chain may end at a slow but functioning component. A cycle may be broken by a timeout or cancellation. These possibilities belong in the diagnosis before we declare the continuation closed.

The scope also matters. Two deadlocked workers can lose their requests while an unrelated health endpoint continues to respond. A manually collected dump can preserve this condition while the process remains present. In the trace, Incomplete History identifies outstanding requests, and Discontinuity identifies a gap in an activity. Neither trace pattern alone proves a deadlock [4].

3.4 Livelock and unproductive repetition

In livelock, activities continue changing state but fail to advance the required work. Our Livelock pattern includes an example of threads repeatedly acquiring and releasing a resource without

progressing [19]. High CPU use is only a possible accompanying observation. A slow retry loop can also consume the entire useful lifetime of a request.

We therefore compare activity with an independent measure of progress. Periodic Error may reveal repeated unsuccessful attempts, while Counter Value or State Dump can show whether the queue head, completed operation count, or application state changes [4]. Repetition becomes diagnostically relevant when its effect on the obligation is known. Periodic polling may be correct behavior; repeated retries may eventually succeed.

For an agentic workflow, the corresponding constructed example is repeated planning and tool invocation without an accepted task result. Tool-call count and generated text measure activity. They do not establish task completion. The same pattern-oriented distinction applies without attributing intentions, awareness, or biological properties to the agent.

3.5 Exhaustion and starvation

A computation may lose the resources needed for its next useful transition. The relevant resource can be memory, a handle, a runnable execution slot, a connection, or storage capacity. This form often develops over an interval. The service initially works, then slows, then rejects or abandons work, and may finally terminate.

Memory Leak and Insufficient Memory are relevant when retained allocation and a limiting capacity are involved. Thread Starvation is relevant when required activity is not scheduled sufficiently [16]. The diagnostic task is to establish the connection between a resource condition and the missing progress. A large memory footprint by itself does not prove a leak. A leak by itself does not establish that the observed request failed because of exhaustion.

The Counter Value trace pattern can connect resource growth to declining completions [4]. A series of dumps can help distinguish a stable large working set from continuing accumulation. Restart may temporarily restore capacity. If the triggering behavior remains, repeated replacement produces a sequence of short-lived incarnations rather than a durable recovery. This is where a death profile becomes a history of changing conditions.

3.6 Expiry of a computational opportunity

Some computations become useless before they stop executing. Suppose that a controller must provide a result before a physical action, or that a request must complete before its contractually defined deadline. After the relevant boundary, the same value may no longer satisfy the obligation.

We can call this the death of a computational opportunity. The name is proposed here for a diagnostic viewpoint on an already familiar deadline failure. It does not imply that every slow request is permanently lost. The obligation must actually require the missed time bound, and the observation must establish that the bound was crossed.

This form explains why a later successful completion can coexist with an earlier failure. The value was eventually computed, but the timely result no longer exists as an available outcome. Cancellation may reduce wasted work after that point. It cannot retroactively satisfy the missed obligation. Diagnostic records should retain both the deadline and the eventual completion time.

3.7 Partial and relational failure

A component can be usable by one consumer and unusable by another. A partition, a failing dependency path, or a selective malfunction can preserve local execution while preventing a particular service relationship. Declaring the whole machine dead hides the location of the lost capability.

Gray failure research describes differential observability, including situations in which applications experience a problem that system failure detectors do not recognize [10]. Failure-detector theory separately explains why suspicions of crashed processes must be distinguished from reliable knowledge under specified distributed-system assumptions [7]. Silence can result from delayed communication as well as termination.

We consequently attach the consumer and communication path to the service dimension of the profile. We correlate client-side and server-side evidence, preserving uncertainty about clocks and message delivery. Adjoint Thread of Activity is useful for following a selected request or another logical grouping across execution threads; Inter-Correlation relates evidence from different traces [4]. A process identifier alone is usually too narrow for this task.

3.8 Loss of valid results

A computation can continue producing outputs after those outputs cease to satisfy its intended meaning. Consider a service that acknowledges a write but fails to make the promised state change, or a model-serving pipeline that silently applies the wrong preprocessing. These are constructed examples of failures of the delivered function. Process presence and response counts can remain normal.

We may describe this as functional or semantic death only after specifying the lost function. The phrase should not turn every incorrect value into a total system failure. A single erroneous result, a sustained failure of one operation, and the loss of all useful service are different claims.

The appropriate evidence includes the expected state transition, the observed result, and an independent check of the postcondition. Abnormal Value, Message Invariant, and State Dump may help organize trace evidence [4]. A pattern match locates an anomaly; the application model determines whether that anomaly violates the obligation. This is consistent with our earlier discussion of model-dependent judgments of software defects [3, “On Subjectivity of Software Defects”].

3.9 Loss of recoverable state

Execution and state have different endings. A process may terminate while a usable checkpoint and replay inputs preserve its work. A process may also continue after unique state required by one task has been overwritten. The first situation can permit recovery; the second can prevent it.

Rollback-recovery research distinguishes checkpoint-based and log-based approaches to state restoration [11]. For diagnostics, this directs attention to the actual recovery materials. A file named checkpoint is insufficient evidence. We need a usable state, compatible interpretation, and the inputs or recorded events required by the recovery method. Ordinary application logs need not contain enough information for replay.

Deletion of one copy does not establish information loss if another usable copy exists. Loss of the only usable decryption key can make retained bytes operationally unrecoverable under the available recovery methods. Missing schemas, executable dependencies, or nondeterministic inputs can also prevent an acceptable continuation even when some bytes survive. Each conclusion must state the relevant recovery assumptions.

This form is particularly important before restart. A memory dump may preserve diagnostically useful state that would otherwise disappear. Whether that dump can support execution recovery is a separate question. Inspection and resumption require different kinds of completeness.

3.10 End of an incarnation and continuity of identity

When a supervisor starts another process, the replacement can inherit a service name without inheriting all the previous work. Similarly, reloading model weights does not automatically reconstruct an agent's pending tool operations, external commitments, or task-specific state. We should identify which continuity was restored.

We distinguish continuation of the same execution, recovery of an accepted logical state, and replacement by an equivalent service. These distinctions are proposed as entries in the continuation contract. They are not claims about personal identity. A system can legitimately require only service equivalence, while a financial transaction or a scientific experiment may require stronger preservation of state and provenance.

Repeated restart is therefore ambiguous. It may demonstrate successful fault containment, or it may repeatedly abandon work. In a trace, we should preserve incarnation identifiers and link them to logical request identifiers. Reuse of a process identifier or a stable service name must not silently join unrelated histories. Identification Messages and Adjoint Thread of Activity help make these relationships explicit [4].

3.11 Loss of the substrate or its supporting environment

Hardware destruction can end one physical execution without destroying the program or the service. The relationship changes when learned behavior depends on the particular physical substrate. Hinton's discussion of mortal computation considers systems whose learned parameters exploit the properties of an individual hardware instance; copying those parameters to a different instance need not preserve the computation [12]. That is a specific technical proposal, and an important precedent for any discussion of computational mortality.

Our diagnostic extension asks which parts are replaceable under the continuation contract. A device can be substituted for one workload but indispensable to another because its unique state or physical characteristics are required. The answer comes from the actual dependency, not from describing all hardware as mortal.

At another scale, a program can lose its usable environment while its source and executable remain intact. A required runtime may no longer be available, a remote service may be withdrawn, or an authorized execution environment may cease to exist. We can call this ecological death as a proposed viewpoint on dependency loss. A support deadline alone does not establish it. Continued execution, migration, emulation, and lawful replacement may remain possible.

3.12 Observational and evidential endings

A log can end while the computation continues. A collector can fail, a buffer can overflow, a trace can be stopped, or a portion of the collected files can be missing. Blackout, Discontinuity, Incomplete History, Silent Messages, and Visibility Limit distinguish several aspects of absent or incomplete trace evidence [4]. They should not be collapsed into a single assertion that the system died.

Blackout concerns a temporary gap in the overall recorded activity in its stated scope. Discontinuity concerns a gap in an individual activity. Visibility Limit concerns what a tracing arrangement cannot observe, such as execution before the tracing facility starts. Silent Messages provides a representation for absent messages within a chosen temporal resolution. Its interpretation depends on the expected message stream; it does not establish a universal requirement that every time slot contain a message.

We distinguish observational loss from evidential loss. Observational loss means that a particular view stops supplying observations. Evidential loss means that the materials needed to answer a specified diagnostic question are no longer recoverable. A collector failure may cause only a temporary observational loss if local buffers survive. Destruction of those buffers can turn it into an evidential loss.

Evidential death is a proposed term for the latter boundary. The running service and the evidence about a failed request can have opposite outcomes: the service recovers, while the explanation becomes permanently less recoverable. This is a property of the diagnostic problem, not proof that the workload itself has stopped.

4 Memory dumps and diagnostic afterlife

A memory dump can outlive the execution from which it was collected. It can preserve an exception context, thread relationships, objects, counters, and residual data. In this sense, it provides a diagnostic afterlife. The expression refers to the continued availability of evidence, without implying that the saved execution is still active.

Our earlier definition of memory dump analysis emphasizes modeling a system from a memory slice [3, “What is Memory Dump Analysis?"]. The slice can be collected during a hang or during normal execution. The existence of a dump therefore does not establish that the process crashed. Acquisition time, trigger, scope, and included memory belong to the interpretation.

Microsoft’s documentation also cautions against inferring coverage from the word minidump: user-mode dump contents vary with the acquisition options [15]. A dump may omit the state required by a particular hypothesis. Conversely, a large dump can still lack remote state and earlier events. More bytes do not automatically supply the missing causal relationship.

In “Four Causes of Crash Dumps,” we applied Aristotle’s material, formal, efficient, and final causes to dump production [22]. We can extend these questions to computational endings. The material viewpoint concerns the hardware and software involved. The formal viewpoint concerns the algorithm or arrangement of dependencies. The efficient viewpoint concerns the event or agent that brings about termination or capture. The final viewpoint concerns the intended purpose, such as containment or preservation of evidence. Here final refers to purpose; it does not mean the last event in a trace.

Consider a deliberately triggered kernel bugcheck to obtain a memory dump [22]. Execution ends in order to preserve diagnostic evidence. The original failure may have occurred earlier. We should therefore distinguish that failure, the terminating action, and the purpose of collecting the dump. In a trace, an error message, a capture request, and a termination record may belong to different causal layers. Their temporal order alone does not establish those relationships. The profile records what ended and what remains recoverable. The causal viewpoints help explain how and why, including cases where ending execution preserves its state for later analysis.

Execution Residue is particularly relevant to computational endings [16, 20]. Data remaining on a raw stack or elsewhere may help reconstruct earlier execution even when no corresponding message was retained. Such residue is not an automatic chronological record. We must distinguish currently active structures from values that survived earlier use, and we must avoid interpreting reused memory as one continuous history.

A zombie illustrates a different survival. In Linux, a terminated child can retain a kernel record until its status is collected; it is not executing user code merely because that record remains [13]. Our catalog also contains a Windows-oriented Zombie Processes pattern [16, 21]. Platform-specific lifecycle and reference semantics must be checked before transferring the name between systems. Residual representation and continuing execution are separate properties.

5 Connections to existing analysis patterns

The following mapping identifies useful starting points. It does not make the listed patterns necessary or sufficient conditions for a particular form of death. The memory pattern names follow our catalog, selected pattern articles, and reference volume [16–21]. The trace pattern names follow the fifth edition of our reference [4].

Table 2. Connections to established diagnostic patterns

Diagnostic question	Memory dump patterns	Trace and log patterns
Why did this incarnation terminate?	Invalid Pointer; Dynamic Memory Corruption; Lateral Damage	Last Activity; Exception Stack Trace; Incomplete History
What prevents the required continuation?	Deadlock; Wait Chain; Blocked Thread	Discontinuity; Incomplete History; Adjoint Thread of Activity
Is there activity without useful progress?	Livelock; Spiking Thread	Periodic Error; Counter Value; State Dump
Was a necessary resource exhausted?	Memory Leak; Insufficient Memory; Thread Starvation	Counter Value; Abnormal Value; Inter-Correlation
Did a replacement preserve the logical work?	Historical Information; Execution Residue	Identification Messages; Adjoint Thread of Activity; Incomplete History
Did the workload disappear or only the observation?	Manual Dump; Missing Thread; Truncated Dump	Blackout; Discontinuity; Visibility Limit; Silent Messages
What remains after termination?	Execution Residue; Zombie Processes; Historical Information	Last Activity; State Dump; Message Context

A useful pattern combination narrows a hypothesis. For example, Incomplete History locates a missing response, an Adjoint Thread of Activity groups the corresponding work, and a Wait Chain in a dump identifies the immediate dependency. If the dependency closes into a deadlock under the relevant synchronization rules, the combined evidence supports a stronger conclusion than any missing log message alone.

The reverse direction is also useful. A dump may show a blocked worker without explaining which user operation it serves. Correlated trace identifiers can supply that context. We then connect the structural failure to an obligation and a deadline. This is where a computational death judgment becomes a specific diagnostic statement.

6 Death Boundary as a candidate analysis pattern

We propose Death Boundary as an organizing pattern for cases in which several endings are incorrectly treated as one. It records a boundary for each relevant loss and keeps the evidence for that boundary separate from the chosen name of the failure.

Context. An investigation includes an apparent ending, a loss of progress, a recovery action, or a disputed declaration that a component is dead. More than one entity or continuation requirement is involved.

Problem. The last visible message, the first missed deadline, the terminating event, and the loss of recoverable state may occur at different times. Assigning all of them one timestamp can select the wrong causal interval or conceal the effect of intervention.

Evidence organization. Identify the entity and continuation contract. Record the last positively established acceptable state or progress event. Record the first positively established violation or terminal event. Preserve any interval between them. Add separate boundaries for execution termination, recovery, and loss of diagnostic evidence when they matter.

Interpretation. A boundary can be exact within the resolution and clock assumptions of an authoritative record. It can also remain an interval. In distributed evidence, a causal ordering may be more defensible than one global timestamp. A final trace line supplies the end of a retained recording; it supplies the time of execution death only when additional evidence establishes that relationship.

Exclusions. A normal idle interval, an intentionally stopped trace, or an expected completed batch does not establish functional death. A missing component in a restricted observation does not establish global absence. An unknown recovery path does not establish irreversible loss.

Result. The analyst obtains a set of qualified boundaries linked to the Computational Death Profile. These boundaries guide collection and intervention. If the useful state still exists before restart, the next action may preserve it. If the deadline has already expired, later recovery is described as recovery for subsequent work.

Death Boundary is a candidate proposed in this article. Its recurring usefulness needs to be established across independently analyzed cases. It complements Last Activity and Discontinuity by organizing the interpretation of several endings; it does not change their published definitions.

7 Constructed diagnostic examples

The following cases are constructed to illustrate the proposed distinctions. Their times and values are explanatory examples, not measurements from production incidents.

7.1 A service with a responding health endpoint

Consider a service with a dedicated health thread and two workers. At 10:00:00, request 417 is accepted with a five-second deadline. Shortly afterward, the workers acquire two exclusive locks in opposite order. Each waits for the lock held by the other. Their synchronization has no timeout or cancellation path. The health thread uses neither lock and continues responding.

At 10:00:05, the request's timely outcome is lost. At 10:00:07, a manually collected dump establishes the two ownership and waiting relationships. The trace contains the accepted request and no completion. The process has not terminated, and the health endpoint remains available.

Deadlock supplies the structural explanation. Incomplete History identifies the unfinished request. The profile separates the facts: execution is present, progress of the affected work is blocked, the deadline has expired, and evidence is preserved. Replacing the process can restore service for later requests if the locks and work state are local to the replaced instance. It cannot complete request 417 within the already expired deadline.

This distinction also explains why probe design matters. Kubernetes treats liveness and readiness probes differently: failed liveness checks can trigger container restart, while failed readiness checks affect readiness without directly terminating the container [14]. Neither probe's name establishes that its implementation tests the business obligation. A health endpoint that ignores the deadlocked workers can continue to succeed.

7.2 A restart that restores service and loses volatile evidence

Now consider a worker that retains a new response buffer after every failed retry. Request 802 has a 30-second deadline. Its input remains in a durable queue. Its retry state and retained buffers exist only in memory. Completion stops before the supervisor terminates the process. No dump is collected. The timing below comes from one clock in this constructed example.

Table 3. Distinct boundaries in a constructed restart case

Time	Established event	Boundary or interpretation
12:00:00	Request 802 is accepted	A specific obligation starts
12:00:01	The last successful suboperation completes	Last positively observed progress
12:00:01–12:00:30	Retries continue; retained memory rises	Activity continues without the required completion
12:00:30	The request deadline expires	The timely outcome is lost
12:00:40	The supervisor terminates the worker	The original execution ends
12:00:42	A replacement starts; durable input is retried	Later service recovery becomes possible

Time	Established event	Boundary or interpretation
12:01:00	A later request completes	Service recovery is positively observed

Several boundaries are visible. The last observed progress occurred before the deadline. The deadline occurred before termination. The first confirmed recovered service occurred after the replacement started. None of these times can be substituted for the others.

The missing dump creates a separate evidential problem. Suppose the retained logs record retries and aggregate memory use but omit buffer ownership. The live object relationships that could establish why buffers accumulated are no longer available through the retained artifacts. We can report the supported resource trend and the lost evidence without claiming that every possible explanation has vanished. A later reproduction may still answer the causal question.

Memory Leak and Insufficient Memory would be hypotheses to investigate, not facts established merely by the rising counter. Periodic Error and Counter Value describe the retained observations. A pretermination dump, appropriately scoped and timed, could have supplied the missing ownership evidence. The recovery restored service while reducing the diagnostic possibilities.

7.3 An apparently dead workload that continues

In the third case, a central dashboard stops receiving events at 14:03. An independent transaction record shows successful work at 14:04 and 14:05. Local files still contain those events. The collector, rather than the workload, has stopped forwarding them.

The central view has an observational ending. The workload's execution and service continue for the operations established by the independent evidence. The local recording preserves the missing interval, so the evidence has not yet died. If forwarding resumes and the central history is repaired, the apparent ending becomes a gap in that view.

Blackout and Inter-Correlation guide the analysis [4]. The independent record does not prove that every operation succeeded, and the collector failure does not prove that every workload is healthy. It establishes a narrower result: the original inference from central silence to total workload death was unsupported.

8 What is new in this proposal

The component ideas have substantial prior history. Fault and failure taxonomies distinguish causes, erroneous states, and service deviations [5]. Our earlier application of the four causes to dump production also distinguished the substrate, arrangement, triggering agent, and purpose involved [22]. Liveness concerns progress [6]. Failure detectors address uncertainty about failed processes [7]. Crash-only design, rejuvenation, and rollback recovery address continued service across local endings [8, 9, 11]. Gray failure addresses differences between observers [10]. Hinton gives mortal computation a particular relationship to physical hardware [12]. Our existing analysis patterns already describe deadlock, livelock, missing activity, residual state, and incomplete evidence [3, 4, 16–21].

The novelty claimed here is the particular diagnostic synthesis. First, the Computational Death Profile places execution, progress, service, continuity, substrate, and evidence in one representation while preserving their separate truth conditions. Second, every death judgment carries a continuation contract. This makes statements about irreversible loss testable against an explicit scope, deadline, identity requirement, and set of recovery assumptions. Third, Death Boundary organizes multiple endings within one incident and maps them to existing evidence patterns.

A further consequence is the explicit separation of service recovery from evidential survival. Restoring a service can remove the information needed to explain its earlier failure. Treating evidence as a separate dimension makes that tradeoff visible before intervention. It also distinguishes a temporary observation gap from the irreversible loss of the materials needed for a particular diagnostic question.

These contributions are proposed concepts and an analysis method. They are not an established exhaustive taxonomy, a demonstrated improvement in diagnostic accuracy, or a claim of historical priority for the general idea that computations can die. Their value should be evaluated by applying them to cases where ordinary crash and hang labels leave a practical ambiguity.

One validation method would give analysts the same incident artifacts with and without the profile, then compare unsupported death declarations, distinctions between requests and process incarnations, selected evidence-preservation actions, and agreement on what remains unknown. The case set should include successful termination, recoverable suspension, deadlock, prolonged retry, lost telemetry, and restart with lost state. A useful framework should improve these decisions without encouraging analysts to rename every transient delay as death.

9 Consequences for diagnostic practice

We can incorporate the proposal into an existing pattern-oriented investigation without adding a separate classification exercise to every incident. It is most useful when the meaning of dead affects the next action. Before terminating a process, we ask which obligations remain recoverable and which evidence will disappear. Before declaring recovery, we ask which continuity has actually been demonstrated.

The practical sequence is short. Identify the entity and outstanding obligation. Establish the strongest surviving evidence of execution and progress. Inspect the dependency or resource condition that prevents continuation. Check the recovery assumptions. Preserve the evidence needed to discriminate between the remaining explanations. After intervention, verify the intended outcome at the relevant scope.

For observability design, this suggests recording logical request and incarnation identifiers, meaningful progress indicators, termination reasons, and the relationship between recovery and previous work. The selected indicators must correspond to the function we care about. Increasing message volume cannot compensate for recording the wrong events.

The investigation ends with a more precise statement than “the system died.” We identify what ended, what could continue, what was restored, and what evidence remains. Such statements connect the language of computational death to the artifacts on which software diagnostics depends.

References

Online sources were checked on 3–4 October 2026. Pattern names are retained as catalog terms. References [3] and [4] also identify the earlier work from which the present diagnostic vocabulary develops.

- [1] Solé, Ricard. *Endless Endings: The Many Forms of Death and What They Teach Us About Life*. Translated by Eduardo Aparicio. Princeton University Press, 2026. ISBN 978-0-691-27627-4. Publisher description and contents: <https://books.google.com/books?id=W8aREQAAQBAI>
- [2] Princeton University Press. “Ricard Solé on Endless Endings.” Author interview, Ideas, 27 August 2026. <https://press.princeton.edu/ideas/ricard-sole-on-endless-endings>. Consulted publisher republication, 29 August 2026: <https://www.linkedin.com/pulse/ricard-sol%C3%A9-endless-endings-princeton-university-press-ff7cc>
- [3] Vostokov, Dmitry. *Theoretical Software Diagnostics: Collected Articles*. Fourth edition. OpenTask, 2024; consulted revision 4.01, April 2024. ISBN 978-1-912636-91-4. Relevant articles: “Four Causes of Crash Dumps”; “What is Memory Dump Analysis?”; “Crashes and Hangs Differentiated”; “On Subjectivity of Software Defects”; “Metaphorical Bijectionism: A Method of Inquiry.” <https://dumpanalysis.org/theoretical-software-diagnostics-book>
- [4] Vostokov, Dmitry. *Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection*. Fifth edition. OpenTask, 2023; consulted revision 5.06, August 2026. ISBN 978-1-912636-58-7. Relevant entries include Adjoint Thread of Activity, Blackout, Counter Value, Discontinuity, Incomplete History, Inter-Correlation, Last Activity, Periodic Error, Silent Messages, State Dump, and Visibility Limit. <https://www.dumpanalysis.org/trace-log-analysis-pattern-reference>
- [5] Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing.” *IEEE Transactions on Dependable and Secure Computing* 1, no. 1 (2004): 11–33. <https://doi.org/10.1109/TDSC.2004.2>
- [6] Alpern, Bowen, and Fred B. Schneider. “Defining Liveness.” *Information Processing Letters* 21, no. 4 (1985): 181–185. [https://doi.org/10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0)
- [7] Chandra, Tushar Deepak, and Sam Toueg. “Unreliable Failure Detectors for Reliable Distributed Systems.” *Journal of the ACM* 43, no. 2 (1996): 225–267. <https://doi.org/10.1145/226643.226647>
- [8] Candea, George, and Armando Fox. “Crash-Only Software.” *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003. <https://www.usenix.org/conference/hotos-ix/crash-only-software>
- [9] Huang, Yennun, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. “Software Rejuvenation: Analysis, Module and Applications.” *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995, 381–390. <https://doi.org/10.1109/FTCS.1995.466961>
- [10] Huang, Peng, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. “Gray Failure: The Achilles’ Heel of Cloud-Scale Systems.” *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, 150–155. <https://doi.org/10.1145/3102980.3103005>
- [11] Elnozahy, E. N. (Mootaz), Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems.” *ACM Computing Surveys* 34, no. 3 (2002): 375–408. <https://doi.org/10.1145/568522.568525>

- [12] Hinton, Geoffrey. "The Forward-Forward Algorithm: Some Preliminary Investigations." arXiv:2212.13345, 2022. Section 9, "Mortal Computation." <https://arxiv.org/abs/2212.13345>
- [13] Linux man-pages project. "wait(2) — Linux Manual Page." <https://man7.org/linux/man-pages/man2/wait.2.html>
- [14] Kubernetes. "Liveness, Readiness, and Startup Probes." <https://kubernetes.io/docs/concepts/workloads/pods/probes/>
- [15] Microsoft. "User-mode Dump Files." Microsoft Learn, Windows debugger documentation. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/user-mode-dump-files>
- [16] Vostokov, Dmitry. "Crash Dump Analysis Patterns." Software Diagnostics Library. Pattern catalog. <https://www.dumpanalysis.org/blog/index.php/crash-dump-analysis-patterns/>
- [17] Vostokov, Dmitry. "Crash Dump Analysis Patterns (Part 9a)." Deadlock (critical sections). Software Diagnostics Library, 9 February 2007. <https://www.dumpanalysis.org/blog/index.php/2007/02/09/crash-dump-analysis-patterns-part-9a/>
- [18] Vostokov, Dmitry. "Crash Dump Analysis Patterns (Part 42a)." Wait Chain (general). Software Diagnostics Library, 14 December 2007. <https://www.dumpanalysis.org/blog/index.php/2007/12/14/crash-dump-analysis-patterns-part-42a/>
- [19] Vostokov, Dmitry. "Crash Dump Analysis Patterns (Part 126)." Livelock. Software Diagnostics Library, 8 January 2011. <https://www.dumpanalysis.org/blog/index.php/2011/01/08/crash-dump-analysis-patterns-part-126/>
- [20] Vostokov, Dmitry. "Crash Dump Analysis Patterns (Part 60)." Execution Residue (unmanaged space, user). Software Diagnostics Library, 29 April 2008. <https://www.dumpanalysis.org/blog/index.php/2008/04/29/crash-dump-analysis-patterns-part-60/>
- [21] Vostokov, Dmitry. *Encyclopedia of Crash Dump Analysis Patterns: Detecting Abnormal Software Structure and Behavior in Computer Memory*. Third edition. OpenTask, 2020. See the Zombie Processes entry. ISBN 978-1-912636-30-3. <https://www.dumpanalysis.org/encyclopedia-crash-dump-analysis-patterns>
- [22] Vostokov, Dmitry. "Four Causes of Crash Dumps." Software Diagnostics Library, 23 November 2007. Also collected in [3]. <https://www.dumpanalysis.org/blog/index.php/2007/11/23/four-causes-of-crash-dumps/>