WHITE PAPER · VOLUME 3 OF 3

Supportability Engineering for Agentic Development:

When the Builder Can't Sign Off

The first two volumes in this series addressed supportability from the perspective of what is built: how to design traditional software and agentic AI systems so that support can operate them at 2am without calling the engineer who built them. This volume addresses a harder problem: what happens when there is no engineer who built it — because the code was generated by an agent, the architecture emerged from autonomous sessions, and the human in the loop reviewed outputs rather than authored them.

John A. Bowman
Supportability Engineering Practice
dooohhead@gmail.com • 902-489-2429
2026

EXECUTIVE SUMMARY

The Framework Assumes Humans. What Happens When They Aren't?

The Supportability Engineering framework was built on a set of assumptions that have held true for as long as software has been built by humans: that requirements are discussed in meetings, that architectures are drawn by engineers, that code is written by developers who understand what they are building, and that every phase can be signed off by a named person who is accountable for the decision.

Agentic development workflows — Copilot, Cursor, Claude Code, and fully autonomous coding agents — are eroding each of those assumptions simultaneously. Code is being generated faster than it can be understood. Architectures are emerging from accumulated agent sessions rather than being designed. Pull requests are reviewed by engineers who can verify that the code works but may not fully understand why it was written the way it was. And the sign-off that the Supportability Engineering framework requires at every phase is being placed on outputs that no human fully authored.

This is not a criticism of agentic development. The productivity gains are real and the trajectory is clear. It is an observation that the supportability problem becomes harder — not easier — as the human authorship of code decreases. The question this paper answers is: what does the Supportability Engineering framework look like when the builder is an agent, and how do you ensure that the systems produced by agentic development can still be operated, diagnosed, and supported by a human at 2am?

SECTION 1

The Four Assumptions That Break

The Supportability Engineering framework makes four implicit assumptions about the development process. Agentic development workflows break all four. Understanding exactly how they break is the prerequisite for knowing how to fix them.

Assumption 1: Someone Knows Why the Code Was Written This Way

Every supportability decision in the original framework — the failure mode inventory, the observability gap remediation, the error handling standards — assumes that the engineers who built the system can explain the decisions they made. A support engineer escalates. An engineer picks up the call. They explain: "This service calls that API because of this business rule, and when it fails it logs this error because we knew that would be the most useful signal."

In an agentic development workflow, that explanation may not exist. The agent generated the service call because it was the most statistically likely solution given the context it was provided. The error logging was added because the agent's training included patterns for error logging. Nobody made a decision. A pattern was applied. When support escalates and asks "why does it work this way?" the honest answer may be: "We don't fully know. The agent wrote it."

> ***When an agent writes the code, the institutional knowledge that used to live in the engineer's head now lives nowhere. Unless it was deliberately captured, it does not exist.***

Assumption 2: The Architecture Was Designed

The SAR — Supportability Architecture Review — assumes that someone drew an architecture diagram before the system was built, and that a reviewer can mark up that diagram to identify blind spots, trace gaps, and dependency risks before they are locked into code.

Agentic development does not necessarily produce architecture first and code second. It may produce code in many autonomous sessions, each of which makes locally sensible decisions that collectively produce an architecture nobody planned. The system that emerges may have service boundaries that no architect would have drawn, dependency patterns that reflect the agent's training data rather than the team's design principles, and observability gaps that nobody noticed because nobody was watching the architecture take shape.

You cannot review an architecture that was never drawn. You can only reverse-engineer one from the code that was generated — and by that point, the cost of changing it has already escalated from hours to sprints.

Assumption 3: The Reviewer Understands What They Are Reviewing

The SIC — Supportability Implementation Checklist — places explicit accountability on the code reviewer. The reviewer is required to independently verify that logging is complete, that correlation IDs propagate, that failure modes are handled, and that no sensitive data appears in log output. The reviewer signs off. Their name is on the checklist.

When code is generated by an agent, the reviewer is in a fundamentally different position. They are reviewing an output they did not produce, often at a level of complexity and volume that exceeds what a human would write in the same time. Research consistently shows that humans reviewing AI-generated code are less likely to catch subtle errors than they are in human-written code — partly because of the volume, partly because the code often looks correct at first inspection, and partly because the reviewer's accountability instincts are calibrated for a world

where they are catching human mistakes, not AI-generated patterns that are wrong in ways that require deeper analysis to detect.

THE REVIEW CONFIDENCE TRAP

AI-generated code tends to look clean, well-structured, and syntactically correct. This creates a false confidence effect: reviewers spend less time scrutinizing code that looks professional, regardless of whether the underlying logic is sound.

Supportability issues — missing correlation IDs, incorrect log levels, unhandled failure modes — are not syntactically visible. They require the reviewer to mentally simulate the code's behaviour under failure conditions. This is exactly the kind of analysis that is hardest to do at volume.

A reviewer who signs the SIC on agent-generated code without a structured verification process is not providing the accountability the framework requires. They are providing the appearance of it.

Assumption 4: The Feedback Loop Has a Human Memory

The SFL — Supportability Feedback Loop — works by capturing operational experience in a quarterly review and converting it into requirements for the next development cycle. A support lead reviews incident scores, identifies patterns, and brings those patterns into the next SRD conversation with product and engineering.

If the next development cycle is also agentic, that conversation needs to produce something an agent can act on — not just a set of human intentions that an engineer translates into code. The feedback loop has to close not just into human memory but into agent context: the constraints, the patterns, the anti-patterns, and the requirements that the agent will use when it generates the next version of the system.

An SFL that produces a slide deck for a quarterly meeting is not sufficient when the next sprint is going to be executed by an agent that will never attend that meeting.

S E C T I O N 2

The New Failure Modes of Agent-Built Systems

When agents build the software, new classes of supportability failure emerge that have no equivalent in human-authored systems. These are not bugs in the traditional sense. They are structural properties of how agentic development produces code.

Accretion Architecture

Human development teams make architectural decisions intentionally. They debate service boundaries, agree on data flow patterns, and document the reasoning. Even when the architecture evolves, the evolution has a history: pull requests, design documents, architectural decision records.

Agentic development can produce architecture through accretion — each session adding components that make local sense, with no session having visibility into the full picture. The result is a system whose architecture was never decided, only accumulated. Service boundaries exist where they do because an agent session stopped there. Dependencies exist because an agent session found them convenient. The system works — but nobody designed it to be debugged, and the debugging experience reflects that.

> ***An accreted architecture has no blind spots by design. It has blind spots by default — everywhere the agent didn't think to add visibility, because nobody asked it to.***

Confident Mediocrity in Observability

A human engineer who understands a system deeply tends to add observability that reflects that understanding — logging the things that have been hard to debug in the past, instrumenting the metrics that the team has learned to watch, writing error messages that capture the context that previously made incidents hard to diagnose.

An agent adds observability based on patterns in its training data. Those patterns are statistically representative of how observability is typically implemented — which means they reflect the average quality of observability across the codebases the agent was trained on. Average quality observability is not sufficient for a production system under support. It produces logs that exist but don't say the right things, metrics that are instrumented but don't capture the signals that matter, and error messages that are syntactically correct but operationally useless.

The agent does not know what it does not know about the system's operational context. It cannot write the log line that says "this failure will look like X but is actually caused by Y" because that knowledge comes from incident history, not from training data.

Pattern-Matched Security and Compliance Risks in Logs

One of the most important requirements in the SIC is that no sensitive data appears in log output. PII, credentials, tokens, financial data — these must be explicitly excluded from every log line. Human engineers who understand the data model can make that judgment for each log statement they write.

Agents make that judgment based on patterns. They know that passwords should not be logged because their training data contains that pattern. They may not know that a specific field in a specific internal data model is PII under the company's data classification policy — because that classification is institutional knowledge that was not in the context they were given. The result is log lines that look correct but contain data that should not be there, sitting quietly in a log aggregation platform until a compliance audit or a breach reveals it.

The Invisible Dependency

Human engineers who add a dependency to a system know they have done so. They made the decision. They understand the implications. They can explain it in a code review.

Agents add dependencies as implementation details. An agent solving a problem may call an external API, import a library, or establish a service connection because it is the most direct path to a working solution. The engineer reviewing the output sees that the code works. They may not notice that a new external dependency has been introduced, that this dependency has no fallback behavior, that it is not covered by the SRD dependency risk assessment, and that if it goes down on a Saturday night, support will have no idea what they are looking at.

S E C T I O N 3

Adapting the Framework for Agentic Development

The response to these challenges is not to reject agentic development. It is to adapt the Supportability Engineering framework so that it works with agentic development workflows rather than assuming they do not exist. Each phase requires specific adaptations.

Phase 1 — SRD: Supportability as Agent Context, Not Meeting Notes

The SRD's purpose is to capture, before design begins, the requirements that support needs to operate the system independently. In a human development workflow, this happens in a meeting and the output is a document that engineers read before they design.

In an agentic development workflow, the SRD must also produce something an agent can consume directly: a structured supportability context that is injected into every agent session working on the feature. This is not a reformatted version of the SRD document. It is a purpose-built context block that tells the agent, at the start of every session, what it must produce for support to be able to operate this system.

THE SUPPORTABILITY CONTEXT BLOCK — INJECTED INTO EVERY AGENT SESSION
Failure modes this agent session must handle: [list from SRD failure mode inventory]
Logging requirements: every transaction boundary must emit a structured log entry containing correlation ID, customer ID, component name, and a meaningful description of the state change.
Sensitive data exclusion list: the following fields must NEVER appear in log output: [list from SRD compliance flags]
External dependencies introduced in this session must be added to the dependency inventory and must include: timeout handling, fallback behavior, and a circuit breaker pattern.
Every error state must return a message that a support engineer can act on without engineering escalation. Generic error messages are not acceptable.
Observability requirement: the four golden signals (latency, error rate, throughput, saturation) must be instrumented for every component this session introduces.

This context block is not optional and not advisory. It is a constraint that shapes every line of code the agent generates. The SRD author is now also responsible for producing the agent context block — a new deliverable that did not exist in the original framework.

Phase 2 — SAR: Continuous Architecture Observation

The SAR was designed as a point-in-time review: an architect draws a diagram, a reviewer marks it up, open items become acceptance criteria. That model assumes the architecture exists before the code. In agentic development, it may not.

The SAR must become a continuous process rather than a gate. After each significant agent development session, an automated architecture extraction runs against the current codebase and produces an updated failure point map. This is not a human effort — it is a tool that reads the code and identifies: new service boundaries, new external dependencies, new data flows, and new points where a failure could occur without generating an observable signal.

Traditional SAR	Agentic SAR
Point-in-time, pre-build	Continuous, post-session
Human draws architecture diagram	Tool extracts architecture from code
Reviewer marks up blind spots manually	Automated analysis identifies observability gaps
Open items become acceptance criteria for build	Gap alerts block the next agent session until resolved
Runs once per feature	Runs after every agent session that introduces new components

Produces a document	Produces a live architecture map with gap annotations
Signed off by architecture reviewer	Signed off by support lead after reviewing gap report

The key principle is that the architecture review cannot happen before the code if the code is being generated autonomously. It must happen after — but before the next session builds on top of an unreviewed foundation. The gap between sessions is the window for architecture observation.

Phase 3 — SIC: Structural Verification, Not Human Review

The SIC's current model — a developer completes a checklist, a reviewer independently verifies each item, both sign — is not sufficient for agent-generated code at volume. The review confidence trap described in Section 1 means that human reviewers cannot be relied upon to catch all supportability issues in generated code through manual inspection alone.

The SIC for agentic development must be enforced structurally, not attested manually. This means moving the supportability checks from a review-time checklist to build-time gates that run automatically on every agent-generated PR.

STRUCTURAL SIC GATES FOR AGENT-GENERATED CODE
Correlation ID propagation scan: automated analysis confirms that every inbound request context carries a correlation ID and that it appears in every outbound call and log statement within the same transaction boundary. PRs that fail this check are blocked.
Log content analysis: static analysis scans every log statement for the presence of field names on the sensitive data exclusion list from the SRD. Any match blocks the PR.
Failure mode coverage check: every failure mode in the SRD failure mode inventory must have a corresponding test that validates the correct error handling and log output. PRs that do not cover all SRD failure modes are blocked.
Dependency registry check: every external dependency call is compared against the SAR dependency inventory. New dependencies that are not registered and assessed block the PR.
Golden signal instrumentation check: automated verification that latency, error rate, throughput, and saturation metrics are instrumented for every new component. Missing instrumentation blocks the PR.
Human review gate: after all structural checks pass, a human reviewer signs the SIC — but their review scope is narrowed to the items that cannot be checked structurally, primarily the semantic quality of error messages and the business logic of failure mode handling.

The human reviewer's role in the agentic SIC is not eliminated — it is made more focused and more effective. By the time a human reviews a PR, the structural checks have already confirmed that the observable properties of the code meet the standard. The human is reviewing meaning, not mechanics.

Phase 4 — STP: Adversarial and Generation-Aware Testing

The STP for agent-generated code must account for a failure mode that human-written code does not produce: the statistically plausible but operationally wrong implementation. An agent generating error handling may produce code that looks correct, handles the expected cases, and passes functional tests — but fails in an edge case that a human engineer would have anticipated from operational experience and an agent could not.

The STP must therefore include generation-aware test cases: scenarios specifically designed to probe the edge cases that agent-generated code is most likely to miss. These are not random fuzz tests. They are structured scenarios derived from the failure mode inventory that target the specific weaknesses of pattern-matched code generation.

Test Type	What It Catches in Human Code	What It Must Also Catch in Agent Code
Failure injection	Known failure modes handled incorrectly	Failure modes that were in the SRD but not handled because the agent's context did not include them
Log quality review	Log levels wrong, context missing	Log content that looks correct but contains excluded fields, or omits required fields in edge cases
Alert validation	Alert routing or severity wrong	Alerts for failure modes that were in the SRD but not instrumented because the agent did not generate the metric
Runbook walkthrough	Runbook steps that don't work in practice	Runbook steps that assume human-authored code behaviour that the agent-generated code does not match
Dependency failure	Known dependency failure handling	Invisible dependencies the agent introduced without registering, which have no failure handling at all
Sensitive data scan	Deliberate PII inclusion	PII in log output from fields the agent did not recognize as sensitive because they were not in its training context

Phase 5 — SRR: Accountability Without Full Authorship

The SRR requires both the support lead and engineering lead to sign off that the team is ready to operate this feature. That accountability is meaningful when the engineering lead has reviewed and understands the system that was built.

In an agentic development workflow, the engineering lead may be signing off on a system that was substantially generated rather than authored. The SRR must therefore include an explicit accountability declaration that is specific to agent-generated code.

AGENTIC DEVELOPMENT ACCOUNTABILITY DECLARATION — SRR ADDITION
Percentage of code in this release that was agent-generated: ____%
Structural SIC gates passed: Yes / No (if No, release is blocked)
Architecture extraction review completed and gap report reviewed: Yes / No
Sensitive data scan completed with zero findings: Yes / No
All SRD failure modes covered by generation-aware test cases: Yes / No
Dependency registry complete and all dependencies assessed: Yes / No
Engineering lead attestation: I have reviewed the structural verification results, the architecture gap report, and the generation-aware test results for this release. I attest that the supportability requirements from the SRD are met to the degree that can be verified through structural analysis and test coverage.
Support lead attestation: I have reviewed the runbook walkthrough results and the alert validation results. I attest that support can operate this feature independently at 2am using only the documentation and tooling that will be available in production.

The accountability declaration does not pretend that the engineering lead has the same depth of understanding they would have if they had authored every line. It requires them to attest to what can be verified — and makes the basis of that attestation explicit. This is more honest, and ultimately more useful, than requiring a sign-off that implies a level of understanding that may not exist.

Phase 6 — SFL: Closing the Loop Into Agent Context

The SFL's quarterly review produces improvements that feed back into the next development cycle. In a human development workflow, those improvements flow through human memory: engineers read the findings, adjust their practices, and write different code next time.

In an agentic development workflow, that feedback must also flow into the agent context block that is injected into every future session. The SFL output is not just a set of findings for the engineering team. It is an updated supportability context block that encodes what was learned from operational experience into the constraints that will shape the next generation of agent-produced code.

SFL Finding Type	How It Updates the Agent Context Block
Logging gap: specific field missing from log output	Add field to required log fields list in context block
Failure mode not handled: edge case produced unhandled exception	Add failure mode to required handling list in context block
Dependency failure: invisible dependency had no fallback	Add dependency to known-risk list; require explicit fallback pattern in context block
Sensitive data in logs: unrecognized field appeared in log output	Add field to sensitive data exclusion list in context block
Runbook mismatch: agent-generated code behaved differently than runbook assumed	Update runbook and add behavioral constraint to context block
Alert gap: failure mode not instrumented for alerting	Add metric requirement to context block for all future sessions touching this component

The SFL for agentic development is the mechanism by which operational experience becomes agent constraint. Every incident that reveals a gap in agent-generated code produces a specific, machine-readable addition to the context block that prevents the same class of gap in future sessions. The system improves not through human habit change but through explicit constraint evolution.

SECTION 4

The Governance Question

Beyond the six framework phases, agentic development raises a governance question that the original framework does not address: who is accountable when a supportability failure in production can be traced to a decision that an agent made, that no human fully reviewed, and that nobody consciously chose?

This is not a philosophical question. It is a practical one with direct implications for how organizations structure their engineering and support organizations as agentic development becomes more prevalent.

The Accountability Gap

In traditional development, accountability for a supportability failure follows a clear path. The developer who wrote the code is accountable for the implementation. The reviewer who signed off is accountable for the review. The support lead who signed the SRR is accountable for the

readiness assessment. If any of those sign-offs were inadequate, there is a named person whose judgment was insufficient.

In agentic development, that path becomes unclear. If an agent generated code that passed structural checks, cleared the automated gates, and was reviewed by a human who could not reasonably have caught the subtle failure mode that produced an incident — who is accountable? The agent is not a person. The reviewer reviewed in good faith. The gates passed.

> ***The answer cannot be 'nobody is accountable'. That answer produces a culture where agentic development becomes a shield against accountability rather than a tool for productivity.***

The answer the framework provides is process accountability rather than individual accountability. The organization is accountable for the adequacy of the structural gates, the completeness of the SRD context block, the quality of the generation-aware test cases, and the rigor of the architecture observation process. If those processes are adequate and a failure still occurs, it is a process gap. If those processes are skipped or inadequate, it is an organizational failure with clear ownership.

The Minimum Viable Governance Model

Organizations adopting agentic development alongside the Supportability Engineering framework need a governance model that defines, at minimum, the following:

- A designated owner for the supportability context block for each feature or service — the person responsible for ensuring it is complete, accurate, and updated based on SFL findings.
- A designated owner for the structural SIC gate configuration — the person responsible for ensuring that the automated checks cover the current SRD requirements and that new requirement types produce new gate configurations.
- A designated owner for the architecture observation process — the person responsible for reviewing the post-session architecture extraction reports and clearing gap alerts before the next session begins.
- A policy on the maximum percentage of agent-generated code that can be released without a full human architecture review — a threshold above which the continuous SAR process triggers a mandatory point-in-time human review.
- A clear statement of what the engineering lead's SRR attestation means in the context of agent-generated code — so that sign-off is an informed commitment, not a formality.

SECTION 5

The Opportunity

This paper has described the challenges of applying Supportability Engineering to agentic development workflows. It is worth being equally clear about the opportunity.

Agentic development, if it is constrained by the right supportability requirements from the start, has the potential to produce more supportable code than human-authored code — not less.

A human engineer who understands a system deeply may add excellent observability. They may also skip the correlation ID, forget to handle the degraded-state failure mode, and write an error message that is meaningful to them and cryptic to support — because they are tired, or under deadline pressure, or simply did not think it through. Human inconsistency is the enemy of supportability at scale.

An agent, given a complete and precise supportability context block, will apply the logging standard consistently to every component it generates. It will not forget the correlation ID on a Friday afternoon. It will not skip the circuit breaker because the sprint is ending. It will apply the pattern it was given to every line of code it produces, without fatigue, without distraction, and without the judgment lapses that make human-authored code inconsistently supportable.

> ***The agent is only as supportable as the context it was given. The job of Supportability Engineering in the agentic era is to make that context precise, complete, and continuously improved by what happens in production.***

This is the version of agentic development that the Supportability Engineering framework makes possible: not a world where agents produce code that support cannot operate, but a world where agents produce code that is more consistently supportable than anything a human team could produce at the same velocity — because the supportability requirements are encoded in the agent's context rather than dependent on individual human judgment.

That is not a speculative future. It is the direct result of applying the framework described in this paper to the agentic development workflows that are already in use today.

SECTION 6

The Complete Three-Volume Framework

The three volumes of the Supportability Engineering series address the same fundamental problem — how to design systems that support can operate independently at 2am without calling the engineer who built them — from three perspectives that together cover the full landscape of modern software development.

Volume	Focus	Core Question
Vol. 1	Traditional software development	How do we ensure that the humans building this system capture everything support needs to operate it?
Vol. 2	Agentic AI systems as the subject	How do we support systems that can reason, decide, and fail in non-deterministic ways that traditional observability cannot capture?
Vol. 3	Agentic AI as the builder	How do we ensure that systems produced by autonomous development agents are built to the supportability standard when no human fully authored them?

The three questions are related but distinct. A team could be building traditional software with an agentic development workflow (Vol. 1 + Vol. 3). They could be building an agentic AI product using human developers (Vol. 1 + Vol. 2). They could be building an agentic AI product using an agentic development workflow (all three volumes simultaneously). The framework is designed to be applied in whatever combination the organization's reality requires.

What does not change across all three volumes is the foundational principle: the cost of a supportability gap grows exponentially the later it is found. Whether the builder is a human, an agent, or a combination of both — the questions that support needs answered must be asked at the beginning, not discovered at 2am.

A B O U T T H E A U T H O R

John A. Bowman

John A. Bowman is a Supportability Engineering practitioner focused on the design and implementation of shift-left supportability frameworks in enterprise software environments. His work sits at the intersection of support operations, software architecture, AI governance, and operational reliability.

This paper is the third volume in the Supportability Engineering series. Volume 1 — "Why the Best Support Organizations Shift Left" — presents the foundational six-phase framework. Volume 2 — "Shifting Left When the System Can Think" — extends the framework for agentic AI systems as the subject of support. The full framework template package, including all adaptations for agentic development workflows, is available on request.

John is available for consulting engagements, staff roles in support engineering, operational readiness, or AI governance, and advisory work with teams navigating the supportability challenges of agentic development at scale. He responds to every inquiry personally.

Contact: dooohhead@gmail.com • 902-489-2429