WHITE PAPER · COMPANION VOLUME

Supportability Engineering for Agentic Systems:

Shifting Left When the System Can Think

In traditional software, a failure has a call stack. In an agentic workflow, a failure has a reasoning chain — and that is far harder to reconstruct after the fact. This paper extends the six-phase Supportability Engineering framework to the specific and novel challenges of agentic AI systems: non-deterministic failure, silent confident errors, mid-execution intervention, and the observability of thought.

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

BEFORE WE START: A STORY YOU ALREADY KNOW

The Failure Nobody Saw Coming

The agent ran overnight. It was processing 200 customer data migration tasks — the kind of work that used to take a team three days. By morning it had completed all 200. The dashboard was green. The logs said success on every task. The model had been confident throughout.

Three days later, a customer called. Their data had migrated — but 40 of the 200 records had been silently transformed in ways the agent decided were correct and were not. No error was raised. No alert fired. No log entry described a decision point or flagged uncertainty. The agent had encountered ambiguous source data, made a judgment call at step 23 of a 47-step task, and continued. It did that 40 times. Each time it was wrong. Each time it reported success.

Support could not reconstruct what had happened. The reasoning that led to each wrong decision was not logged — because nobody had specified that it needed to be. There was no intervention trigger for ambiguous data — because nobody had defined what ambiguous looked like at requirements time. Remediation took two engineers four days, working through output records one by one, because there was no way to query which tasks had hit the ambiguous decision path.

“We didn’t have a logging problem. We had a design problem. Nobody ever asked: what does this agent do when it isn’t sure?”

This is not a hypothetical. It is the defining failure mode of agentic systems that are not designed for supportability. In traditional software, when something goes wrong, it usually tells you. In an agentic system, when something goes wrong, it may complete the task, report success, and leave no evidence of the decision that caused the problem. The system did not fail. The output was wrong. That distinction changes everything about how you design for support.

EXECUTIVE SUMMARY

The Problem Has Changed. The Principle Hasn’t.

The original Supportability Engineering framework — documented in the companion white paper “Why the Best Support Organizations Shift Left” — addresses a well-understood challenge: software that is never designed to be understood when it fails. The six-phase framework answers that challenge by moving supportability questions earlier in the development lifecycle, where they are cheap to answer rather than catastrophically expensive.

Agentic systems change the nature of the problem without changing the validity of the solution. The Shift Left principle still applies. The six phases still apply. The cost curve still applies. What changes is what the questions are, what the answers look like, and what new categories of failure the framework must account for.

This paper defines those changes precisely. It describes the new failure modes that agentic workflows introduce, explains why traditional supportability approaches are insufficient for them, and extends each of the six framework phases with the specific adaptations required to support agentic systems in production.

The result is a complete supportability approach for the agentic era — built on the same Shift Left foundation, extended for a fundamentally new class of system.

S E C T I O N 1

What Is Different About Agentic Systems

Before extending the framework, it is necessary to be precise about what makes agentic systems categorically different from traditional software from a supportability perspective. The differences are not cosmetic. They require genuinely new thinking.

The Fundamental Shift: From Call Stacks to Reasoning Chains

In traditional software, a failure is deterministic. Given the same inputs, the system produces the same outputs. When something goes wrong, you can replay the sequence of events, examine the call stack, and identify the precise line of code or service call that failed. The failure has an address.

In an agentic system, the execution path is not fixed. The agent decides — at runtime, based on context — which tools to call, in what order, with what parameters, based on what intermediate reasoning. Two identical user inputs can produce completely different execution paths depending on what the model “thinks” at each decision point. The failure does not have a fixed address. It has a reasoning chain that led to a decision that led to an outcome — and that chain may never be exactly reproducible.

> ***In traditional software, a failure has a call stack. In an agentic workflow, a failure has a reasoning chain — and that is far harder to reconstruct after the fact.***

The Six New Failure Categories

Agentic systems introduce six categories of failure that the original framework does not directly address. Each requires specific supportability design.

1. Non-Deterministic Failure

The same input, the same agent, the same tools — and different outcomes on different runs. A failure that appeared in production may not reproduce in a test environment. A fix that appears to work may only work probabilistically. Traditional QA assumptions break down entirely.

Supportability implication: You cannot validate a fix by rerunning the exact scenario. You need statistical confidence across a distribution of runs, and your logging must capture enough of the reasoning context to understand why a particular run failed even when others succeeded.

2. Silent Confident Failure

This is the most dangerous failure mode in agentic systems and the one most likely to go undetected. The agent completes the task. It does not raise an error. The output looks plausible. It is wrong. The model was confident and incorrect, and nothing in the traditional error-detection stack — no exception, no non-200 status code, no timeout — fires.

Supportability implication: Traditional alerting watches for system-level failures. Agentic supportability must also watch for semantic failures — outputs that are complete but incorrect. This requires output validation strategies, confidence thresholds, and human review triggers that have no equivalent in traditional software support.

3. Mid-Execution Intervention Triggers

An agent performing a multi-step task may reach a decision point where the correct action depends on information or judgment that should come from a human. In traditional software, the system either has the information or fails gracefully. An agent may proceed — making a reasonable-seeming decision — when it should have stopped and asked.

Supportability implication: The SRD failure mode inventory needs a new category: intervention triggers. For each agent task, what are the conditions under which the agent must pause and surface a decision to a human before proceeding? These triggers must be designed in, not discovered when an agent makes a consequential unilateral decision in production.

4. Context Window Drift

Long-running agentic tasks accumulate context. As the context window fills, earlier information gets compressed, summarized, or dropped. The agent's behavior in step 40 of a task may be

meaningfully different from its behavior in step 4 — not because the task changed, but because the model's effective working memory has changed.

Supportability implication: Support needs visibility into context composition at each reasoning step, not just at task boundaries. An incident that "started fine and then went wrong" may be a context drift incident — diagnosable only if the context state at each step was logged.

5. Tool Schema Drift

An agent that uses ten external tools is dependent on ten external APIs remaining stable. When a tool's schema changes — a parameter renamed, a response format updated, a new required field added — the agent may fail in ways that look like reasoning failures rather than integration failures. The model may attempt to compensate for unexpected tool responses in ways that produce plausible-looking but incorrect outputs.

Supportability implication: Tool schema changes need to be treated as deployment events for the agentic system, with the same supportability gates applied. The SAR dependency risk assessment must specifically address schema drift, not just availability and latency.

6. Prompt Injection and Adversarial Context

An agent that processes external content — documents, web pages, emails, database records — may encounter content that attempts to redirect its behavior. A document that contains instructions disguised as data. A web page that instructs the agent to ignore its system prompt. These are not traditional security vulnerabilities — they are a new class of failure that sits at the intersection of security and supportability.

Supportability implication: The SRD must include adversarial failure modes. The SIC must include input sanitization and instruction boundary verification as implementation requirements. The STP must include adversarial test cases as a standard component.

The Comparison: Traditional vs. Agentic Supportability Challenges

The table below maps the traditional supportability challenge against its agentic equivalent for each of the six failure dimensions. The point is not that the challenges are similar — it is that they are categorically different, and that difference has direct consequences for what the framework must require at each phase. A team that treats agentic supportability as "the same thing, with AI" will design for the wrong failure modes and discover that at the worst possible time.

Dimension	Traditional Software	Agentic Workflow
Failure type	Deterministic — same input, same failure	Probabilistic — same input, different outcomes on different runs
Detection	Error codes, exceptions, timeouts	Semantic validation, confidence thresholds,

method		output review
Root cause	Specific code path or service call	Reasoning chain decision at an unlogged step
Reproduction	Reliable — replay the inputs	Unreliable — nondeterminism means you may never see it again
Failure visibility	System tells you something went wrong	System reports success; output is quietly incorrect
Escalation trigger	Exception, alert, SLA breach	Output review, human-in-the-loop checkpoint, downstream effect
Runbook validity	Deterministic steps produce reliable results	Steps may need judgment calls; runbook cannot cover all branches
Compliance trace	Reconstruct from logs	Must log reasoning steps explicitly or trace is lost forever

S E C T I O N 2

Why Traditional Supportability Is Insufficient

The original Supportability Engineering framework was built on one foundational assumption: that a sufficiently experienced support engineer, given good logs and a good runbook, can diagnose any failure the system can produce.

That assumption holds for deterministic systems. For agentic systems, it breaks in three specific ways.

Logs Are No Longer Sufficient on Their Own

In traditional software, a well-structured log line tells you: what happened, when it happened, to what entity, with what result. In an agentic system, what happened is “the model decided to call this tool with these parameters based on reasoning that was not logged.” The gap between the observable inputs and the observable outputs contains the entire decision-making process of the agent — and if that process was not explicitly captured, it is gone.

You cannot reconstruct why an agent made a decision from the tool call alone, any more than you can reconstruct why a human made a decision by observing only their physical actions.

> ***In agentic systems, the most important thing that happened — the***

reasoning — is the thing most likely to be missing from your logs.

Runbooks Cannot Cover Probabilistic Branches

A traditional runbook says: if you see X, do Y. This works because the system is deterministic — X always leads to the same set of possible causes, and Y always addresses them in the same way.

An agentic runbook cannot make that guarantee. The same observable output — “agent returned an incorrect result” — might be caused by a flawed tool response, a context drift problem, a prompt injection, a schema change, or a genuine model reasoning error. Each has a different diagnosis and a different resolution. The runbook cannot branch deterministically on symptoms that look identical.

This means agentic runbooks must be diagnostic frameworks rather than deterministic procedures. They guide the support engineer through a structured investigation rather than a fixed sequence of steps.

The Feedback Loop Has a New Signal: Reasoning Quality

The original SFL captures three signals from every incident: supportability score, observability gap, and runbook accuracy. These are sufficient for traditional systems.

Agentic systems produce a fourth signal that is qualitatively different: reasoning quality data. Was the model’s intermediate reasoning sound? Did it correctly interpret tool responses? Did it identify the right decision point for human intervention? This signal cannot be captured by traditional logging — it requires explicit reasoning trace capture and a review process that includes someone capable of evaluating model behavior, not just system behavior.

S E C T I O N 3

Extending the Six-Phase Framework

Each of the six Supportability Engineering phases requires specific extensions to address agentic systems. The structure of each phase — its purpose, its sign-off requirements, its connection to adjacent phases — remains intact. What changes is the content of what each phase must capture and verify.

A note on the “more process” objection. Practitioners building agentic systems sometimes resist framework thinking on the grounds that AI moves fast and process slows it down. That objection

misunderstands what is being proposed here. The argument is not more process for agentic systems. The argument is that the traditional reactive approach — ship it and see what breaks — is more dangerous for agentic systems than for traditional software, not less. A deterministic system that breaks will tell you it broke. An agentic system that fails silently, mid-execution, in ways its confidence scores do not surface, may not. Retrofitting observability into a system designed without it is expensive in traditional software. In agentic systems, it may require rebuilding the agent's reasoning architecture from scratch. The six extensions below are not overhead. They are the minimum design discipline required to operate an agentic system in production without flying blind.

Phase	Original Deliverable	Agentic Extension Required
SRD Requirements	Observability requirements, failure mode inventory, customer impact classification, escalation paths	Intervention trigger inventory, acceptable confidence thresholds, reasoning trace requirements, adversarial failure modes, human-in-loop classification
SAR Design	Failure point map, blind spot list, trace boundary definition, dependency risk assessment	Reasoning checkpoint map, tool schema change protocol, context window boundary analysis, agent decision boundary definition
SIC Build	Logging standards, error handling, four golden signals, failure mode unit tests	Reasoning trace logging, prompt/response capture standards, output validation implementation, confidence score instrumentation, intervention trigger implementation
STP Test	Failure injection, log quality review, alert validation, runbook walkthrough, on-call simulation	Adversarial input testing, confidence threshold validation, intervention trigger testing, context drift simulation, non-determinism stress testing
SRR Release	Monitoring live, runbooks published, on-call updated, rollback tested, comms templates ready	Reasoning trace review process confirmed, output review rotation established, model version change protocol documented and tested
SFL Operate	Incident scoring, observability gap log, runbook accuracy tracking, quarterly review	Reasoning quality review, model behavior drift tracking, tool schema change incident log, intervention trigger accuracy review

Phase 1 — SRD: Adding the Agentic Failure Mode Inventory

The SRD for an agentic system must extend its failure mode inventory beyond the six standard categories (complete outage, partial failure, degraded state, data issue, performance, security/auth) to include three new categories specific to agentic behavior.

NEW SRD FAILURE MODE CATEGORIES FOR AGENTIC SYSTEMS
Reasoning Failure — The agent completes the task but the intermediate reasoning was flawed, leading to a correct-looking but incorrect output. Detection: output validation, downstream effect monitoring.

Intervention Miss — The agent proceeded past a defined intervention trigger without surfacing the decision to a human. Detection: intervention trigger audit log, task completion review.
Context Integrity Failure — The agent's behavior changed mid-task due to context window pressure, producing inconsistent outputs across a long-running task. Detection: step-level reasoning trace review.
Adversarial Redirect — The agent's behavior was altered by content in its input that was designed to override its instructions. Detection: instruction boundary monitoring, unexpected tool call patterns.

The SRD must also define acceptable confidence thresholds for each task type. Below what confidence level should the agent surface uncertainty to a human rather than proceeding? These thresholds are business decisions, not engineering decisions — which is exactly why they belong in the requirements phase, not the build phase.

IF YOU SKIP THIS — WHAT 2AM LOOKS LIKE

Without an agentic failure mode inventory in the A-SRD, your team will encounter every new failure category — non-deterministic output, silent confident error, context drift — for the first time during a live incident. Nobody will have defined what intervention should look like. Nobody will have classified which customers are at risk from a silent failure. Support will be writing triage criteria in real time, while the agent continues running. The cost of discovering an agentic failure mode in production is not just the incident. It is every subsequent incident of the same type, because without the A-SRD there is no structured path from operational experience back to design.

Phase 2 — SAR: Mapping the Reasoning Architecture

The SAR for an agentic system produces two artifacts in addition to the standard failure point map and observability gap list.

The Reasoning Checkpoint Map

Where in the agent's task execution can support independently verify that the reasoning is on track? A reasoning checkpoint is a point in the agent's execution where its intermediate state can be observed, evaluated, and if necessary interrupted. The SAR identifies every checkpoint — and, more critically, every gap between checkpoints where the agent is operating without observable state.

Long gaps between reasoning checkpoints are the architectural equivalent of a service with no logging. They are blind spots in the reasoning architecture, and they must be closed before build begins.

The Tool Schema Change Protocol

Every external tool in the agent's toolkit is a dependency. The SAR must define, for each tool, what happens when its schema changes: who is notified, what validation is run against the new

schema, and whether the agent is tested against the new schema before it is used in production. This protocol does not exist in most agentic systems today. Its absence is a latent incident waiting to happen.

IF YOU SKIP THIS — WHAT 2AM LOOKS LIKE

Without a Reasoning Checkpoint Map, your architecture contains blind spots that are invisible on the diagram and catastrophic in production. When a customer reports wrong output from a multi-step agent, support will know the final result was wrong. They will not know which step produced the error, which tool call returned bad data, or whether the agent recognized it had a problem. The investigation starts from zero on every incident. Engineers are pulled in to manually trace execution paths through logs that were never designed to surface reasoning. The architectural decision that created the blind spot is already locked in, and retrofitting it is a multi-sprint project that will be deprioritized indefinitely.

Phase 3 — SIC: The Reasoning Trace Standard

The most important addition to the SIC for agentic systems is a reasoning trace logging standard. This defines what must be captured at every reasoning step, not just at tool call boundaries.

REASONING TRACE LOGGING STANDARD — REQUIRED FIELDS PER STEP
Step identifier — unique ID for this reasoning step, linked to the parent task correlation ID
Input state — the context available to the model at this step (summarized, not full context dump)
Reasoning summary — the model's stated reasoning for its next action (captured from chain-of-thought output where available)
Action taken — tool called, parameter values, or decision made
Tool response — the raw response from any tool called at this step
Confidence signal — the model's expressed confidence in its action, if available
Intervention triggered? — boolean: did this step trigger a human intervention checkpoint
Context token count — current context window utilization as a percentage

This standard is not optional and not a nice-to-have. Without reasoning trace logging, incident investigation in agentic systems is archaeology. With it, a support engineer can reconstruct why an agent made every decision in a completed task, even if the task ran for an hour and touched thirty tools.

IF YOU SKIP THIS — WHAT 2AM LOOKS LIKE

Without reasoning trace logging in the implementation, the most important question in any agentic incident — why did the agent make that decision? — is permanently unanswerable. Logs will tell you what the agent did. They will not tell you why. Confidence scores will not be present, so there is no way to identify the steps where the agent was uncertain. Intervention triggers will not fire, so customers will absorb the impact of decisions the agent should have escalated. Every incident investigation will be archaeology with incomplete evidence. And because the reasoning was never instrumented, the same wrong decisions can be made again, indefinitely, because there is no way to detect the pattern.

Phase 4 — STP: Testing for Non-Determinism

The STP for an agentic system cannot rely on deterministic test cases. Every failure mode must be tested across a distribution of runs, not a single run. This has three specific implications.

- Adversarial input testing must be a standard STP component. Every agentic system that processes external content must be tested with inputs designed to redirect, confuse, or overwhelm the agent's reasoning.
- Confidence threshold validation must be explicitly tested. Trigger the conditions under which the agent should surface uncertainty to a human. Verify that the intervention trigger fires. Verify that it routes to the correct person with sufficient context.
- Context drift must be deliberately induced. Run the agent on tasks long enough to fill the context window past the point where drift is expected. Verify that the reasoning trace shows the drift, that an alert fires, and that the support runbook for context drift produces a correct diagnosis.

> ***The first time a support engineer practices diagnosing an adversarial redirect should be in the test environment — not during a live incident at 2am.***

IF YOU SKIP THIS — WHAT 2AM LOOKS LIKE

Without probabilistic runbook validation, your runbooks will be written for the average-case execution path and will fail on every variant. A support engineer following the runbook at 3am will discover that the steps do not match what they are seeing, because the agent took a different branch than the one the runbook author assumed. The runbook walkthrough that would have caught this in an hour in a test environment will cost three hours during a live incident instead. Non-determinism testing is not optional for agentic systems. It is the only way to know, before a real incident, whether your support tooling is adequate for the full range of behaviors the agent can produce.

Phase 5 — SRR: The Model Version Change Protocol

The SRR for an agentic system has one critical addition that has no equivalent in traditional software: the model version change protocol.

When the underlying model is updated — even a minor version change — the agent's behavior may change in ways that are not reflected in any code diff. A prompt that worked reliably with one model version may produce different outputs with the next. The SRR must confirm that a model version change protocol exists, is documented, and has been tested.

This is a release gate that has no equivalent in traditional software. In traditional software, a deployment is a deliberate act with a diff, a review, and a deployment record. A model version update is often automatic, invisible, and provider-controlled. The behavioral regression test suite is the only mechanism that makes a model update equivalent to a deployment — observable, controlled, and testable before production exposure. An agentic system that ships without this protocol is operating with an uncontrolled update surface. The SRR is the last checkpoint where that gap can be caught before it becomes a customer incident.

MODEL VERSION CHANGE PROTOCOL — REQUIRED ELEMENTS
Behavioral regression test suite — a defined set of tasks with expected output characteristics (not exact outputs) that is run against every new model version before deployment
Confidence threshold re-validation — thresholds defined for one model version may not be calibrated correctly for the next; re-validation is required
Reasoning trace format verification — new model versions may produce different chain-of-thought formats; logging must be verified to capture the new format correctly
Rollback procedure — the ability to revert to a previous model version must be tested, documented, and accessible to support without an engineering deployment

IF YOU SKIP THIS — WHAT 2AM LOOKS LIKE
Without a model version change protocol, a routine model update can silently change agent behavior in production with no code diff, no deployment event, and no alert. The first indication that something is different may be a customer complaint three days after the update. Support will not know when behavior changed. They will not know if the previous behavior was better or worse. The confidence threshold re-validation that would have caught the drift in an hour in a staging environment will instead require a two-day engineering investigation to determine which version introduced the change. Every production model update is a silent deployment of new behavior. Without the protocol, it is uncontrolled.

Phase 6 — SFL: Tracking Reasoning Quality Over Time

The SFL for an agentic system adds a fifth dimension to the standard incident supportability scoring: reasoning quality.

That fifth dimension is the most valuable new capability the agentic SFL provides. Reasoning quality scores, tracked over time, do something no other operational metric can: they tell you whether model behavior is drifting before customers feel it. A gradual decline in reasoning quality scores — not a sudden failure, not a customer complaint, but a measurable downward trend in the coherence and accuracy of the agent's documented reasoning — is an early warning signal that is invisible without this measurement framework. Organizations that track reasoning quality over time can demonstrate to leadership, with data, that model behavior is

improving or degrading. That is a competitive and operational advantage that compounds every quarter.

The SFL also closes the loop that makes the entire framework self-improving. Every incident generates an observability gap log entry. Every gap log entry becomes a backlog item. Every backlog item feeds back into the A-SRD for the next cycle. For agentic systems, this loop has a dimension traditional software never had: the gap log can reveal not just missing metrics or incomplete logging, but failure modes the team did not anticipate at requirements time — because the agent found an execution path nobody designed for. That is not a failure of the process. It is the process working as intended. The A-SFL converts that operational surprise into structured upstream improvement, so the next cycle starts with a more complete failure mode inventory than the last.

Score	Detectable	Diagnosable	Resolvable	Reasoning Trace	Intervention Accuracy
5	Auto-detected before impact	< 15 min by support	No engineering needed	Complete trace, all steps	All triggers fired correctly
4	Within 5 min of impact	< 30 min	Minor eng. input	Trace complete, minor gaps	Triggers fired, minor routing issues
3	Within 30 min	< 2 hours	Eng. escalation required	Trace partial, key steps missing	Some triggers missed
2	Via customer report	> 2 hours	Senior eng. involved	Trace minimal, reasoning unclear	Triggers frequently missed
1	Not detected	Root cause unknown	Code/model change required	No trace, reasoning lost	Triggers not implemented or non-functional

S E C T I O N 4

The New Observability Stack

Supporting agentic systems in production requires an observability stack that extends beyond what traditional distributed systems monitoring provides. The four golden signals — latency, error rate, throughput, saturation — remain necessary. They are no longer sufficient.

Five Additional Signals for Agentic Systems

THE AGENTIC OBSERVABILITY SIGNALS

Reasoning step count — how many steps did this task require? Unusual step counts (too few or too many) are often the first indicator of a reasoning problem.
Confidence distribution — the distribution of confidence scores across reasoning steps. A task that starts confident and becomes progressively less confident may be approaching a context drift failure.
Intervention trigger rate — what percentage of tasks triggered a human intervention checkpoint? Significant changes in this rate indicate behavioral drift in the agent.
Tool call anomaly rate — how often did the agent call an unexpected tool, call a tool with unexpected parameters, or receive an unexpected response? These are early indicators of schema drift or adversarial input.
Context utilization at failure — when tasks fail or produce incorrect outputs, what was the context window utilization? Clustering of failures at high context utilization confirms context drift as a failure mode.

These signals do not require a new observability platform. They require that the reasoning trace logging standard defined in the SIC is implemented correctly, and that the metrics derived from that logging are surfaced in the dashboards that support engineers use during incident response.

> ***An agentic system that is not logging its reasoning is not observable. And a system that is not observable cannot be supported.***

S E C T I O N 5

The Business Case Remains and Strengthens

The cost curve that drives the original Supportability Engineering framework — gaps caught early cost minutes, gaps caught in production cost months — applies with even greater force to agentic systems.

In traditional software, a production incident caused by a missing correlation ID is expensive but bounded. You investigate, you find the cause, you fix the logging, and the class of problem is resolved. The incident cost is one-time.

In an agentic system, a production incident caused by missing reasoning trace logging may be impossible to diagnose fully — because the evidence of what the agent decided and why is gone. You can observe the outcome. You cannot reconstruct the path. The incident cost includes not just the resolution time, but the permanent uncertainty about whether you understood the root cause, and therefore whether your fix was correct.

Agentic systems that are not designed for supportability from the start do not just generate expensive incidents. They generate incidents that cannot be fully closed — because the observability required to confirm the root cause was never built in.

> ***In traditional software, you can reconstruct what happened. In an agentic system without reasoning trace logging, you can only observe that something went wrong. The cost of that uncertainty compounds with every subsequent incident.***

The Shift Left Multiplier

The return on Shift Left investment is higher for agentic systems than for traditional software for one specific reason: the cost of retrofitting observability into an agentic system is not just an engineering sprint. It may require fundamental changes to the agent architecture — changes that affect how the agent structures its reasoning, how it calls tools, and how it surfaces uncertainty.

An agentic system designed from the start with reasoning trace logging, intervention triggers, and confidence thresholds builds these capabilities into its core architecture. An agentic system that tries to add them later may find that they cannot be added without rebuilding the agent from scratch.

The SRD question — “what do we need to be able to observe about this system?” — is even more valuable asked before an agentic system is designed than before a traditional system is designed. The answer shapes the architecture itself.

S E C T I O N 6

A Complete Framework for the Agentic Era

The Supportability Engineering framework was built on a principle that does not change with the technology: the earlier a supportability gap is found, the cheaper it is to fix. Agentic systems do not invalidate that principle. They raise the stakes on it.

The six-phase framework — SRD, SAR, SIC, STP, SRR, SFL — remains the right structure. What changes is what each phase must contain when the system being built can reason, can make decisions, and can fail silently in ways that look like success.

The extensions described in this paper are not theoretical. They are the direct result of applying the original framework's logic to a new class of system and asking the same questions the framework has always asked: how will we know this is working, how will we know it has broken, and what does support need to handle it at 2am without calling the engineer who built it?

The answers are different for agentic systems. The questions are the same. That is the point.

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, and operational reliability.

This paper is a companion to "Supportability Engineering: Why the Best Support Organizations Shift Left," which presents the foundational six-phase framework. The full framework template package — including all six phase templates extended for agentic systems — is available on request.

John is available for consulting engagements, staff roles in support engineering or operational readiness, and advisory work with teams building or maturing their supportability practice for agentic systems. He responds to every inquiry personally.

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