

How SDSE and BSN Shape Software Architecture

From behavioral promises to accountable design decisions

Software architecture should explain how a system will fulfill its behavioral promises under real operating conditions.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

October 2026

www.PrimaModus.com

Executive Summary

Software architecture determines whether a system can fulfill its promises under the conditions in which people actually use and operate it. Simulation-Driven Software Engineering (SDSE) brings those promises into architectural reasoning before expensive design commitments are made. Behavior Simulation Notation (BSN) provides a common representation of participants, interactions, operating conditions, rules, and scenarios that architects can examine and refine. Together, they help teams connect structural decisions to observable behavior and the evidence supporting those decisions.

The practical change is to establish candidate behavior, simulate it, and approve a versioned behavioral baseline before committing to architecture and detailed design. Architecture then allocates that behavior to components, interfaces, data, controls, and infrastructure. Discoveries made during design return to the model for review and, when material, revalidation. The baseline remains authoritative while evolving through controlled engineering decisions.

This article follows an illustrative production incident-response agent. The agent can diagnose a problem and propose remediation, but its authority to execute depends on risk, policy, approval, and operating conditions. Working through this example shows how behavior influences component boundaries, interface semantics, state ownership, resilience, and human control. The goal is architecture that engineers can explain, operators can observe, and accountable people can govern.

Contents

1 The architectural questions that diagrams leave open	4
2 Put validated behavior before design commitments	4
3 Use BSN to make architectural obligations explicit.....	5
4 Define the incident response promise precisely	5
5 Allocate responsibilities and authority deliberately.....	6
6 Design interfaces around outcomes and uncertainty	7
7 Give state and evidence clear owners.....	7
8 Turn quality attributes into architectural scenarios.....	8
9 Compare candidate architectures using the same promises	9
10 Model multi agent interaction and human control.....	10
11 Preserve the promise through delivery and operation	11
12 Begin with one consequential architectural decision	11
Conclusion	12
Further reading and participation	12
About the Author.....	12

List of Figures

Figure 1 From behavioral promise to architectural decision	6
Figure 2 One behavioral promise across alternative architectures.....	10

1 The architectural questions that diagrams leave open

Consider a team designing an AI agent to help restore a production service. Its architecture diagram contains telemetry sources, an AI model, a workflow engine, remediation tools, and an audit database. The diagram is useful because it identifies major dependencies and possible deployment boundaries. Yet it does not establish whether the agent can safely act when an approval arrives late, a tool times out, or two incidents require conflicting changes.

These questions affect the people who depend on the service and the engineers responsible for its recovery. An operator needs to know whether a timeout means that nothing happened or that an action completed without acknowledgment. A security reviewer needs to know which component actually prevents unauthorized execution. A business owner needs to understand how the system behaves when automation cannot restore service within an acceptable period.

Start architectural work by writing the important behavioral promises and identifying the operating conditions that could invalidate them. Use structural views to show where those promises will be realized, and scenario evidence to examine whether the allocation is credible. When behavior remains implicit, teams tend to resolve consequential questions inside code, configuration, or incident response. Those local decisions can conflict even when the overall architecture diagram looks orderly.

2 Put validated behavior before design commitments

The SDSE sequence begins with Intent, proceeds to a Candidate Specification and Behavior Simulation, and establishes a Validated Specification before Design, Code, and Integration. Early modeling can begin as soon as intent is concrete enough to challenge. Formal behavioral validation occurs once the candidate describes the important outcomes, constraints, interactions, and failure responses. Its purpose is to establish a credible promise before design choices narrow the available solutions.

For the incident-response agent, intent might be to shorten recovery while preserving human authority over consequential production changes. A candidate specification must explain what counts as an authorized action, how risk is classified, what evidence supports a proposal, and what happens when approval cannot be obtained. Simulation can reveal that these expectations are inconsistent or incomplete before the team chooses a workflow framework or grants credentials to tools. The resulting approved baseline records both the promise and the assumptions under which it was examined.

This sequence supports iteration. A candidate model may contain tentative participants and relationships needed to express behavior, while architecture later adds concrete allocation, timing, and deployment knowledge. If a design constraint makes the accepted promise infeasible, create a revised candidate and return it for simulation and approval. Quietly weakening the promise inside a design document breaks the connection between stakeholder intent and implementation.

Approve a baseline with a stated scope, scenario set, model version, fidelity, limitations, and residual risk. Assign a named Behavior Model Owner to maintain its integrity, and identify the people authorized to accept changes. This gives architects a stable reference while preserving a controlled route for learning. Without these responsibilities, several teams can work from different versions of what the system is supposed to do.

3 Use BSN to make architectural obligations explicit

BSN organizes behavior through BOX, INTERFACE, OPERATING, LOGIC, NETWORK, and SCENARIO constructs. For architectural reasoning, use them to express the relevant participants, exchanges, operating context, behavioral rules, relationships, and situations to exercise. The model should expose enough detail to answer the current engineering question. A small model that makes one consequential interaction clear can be more useful than a large model with unexamined assumptions.

In the incident-response example, identify the diagnostic participant, policy authority, human approver, remediation executor, and audit function. Describe which information they exchange and under which conditions an action is permitted. Represent scenarios that test late approval, unavailable policy, repeated requests, and uncertain execution outcomes. Review the resulting traces with the people who understand production operation, security, and business consequence.

A behavioral BOX represents a modeling boundary; its implementation allocation requires a separate decision. Several BOXes might reside in one application, while one responsibility could be realized across several processes or services. Decide deployment boundaries using fault containment, trust, state ownership, performance, maintainability, and operational cost. Treating each BOX as a mandatory microservice can introduce coordination problems that the behavioral promise never required.

Keep architecture views and BSN connected through explicit identifiers and trace links. A component should identify the responsibilities and promises it realizes, and an interface should identify the modeled exchanges it supports. This lets reviewers move from a diagram to the behavior behind an arrow. It also makes the consequences of changing that arrow easier to examine.

4 Define the incident response promise precisely

The illustrative promise is that the agent may diagnose a production incident and propose corrective action. It may execute only a low-risk action explicitly allowed by policy, or a higher-risk action approved by an authorized human. If required authorization is unavailable, it must refrain from initiating the action and escalate. Each execution attempt must be traceable to its incident, proposed action, authorization basis, and observed outcome.

Make this promise measurable before designing around it. For this example, assume that human approval expires after five minutes and is bound to the exact target, action parameters, and relevant target-state version. Assume that an unavailable audit path prevents new automated execution, with an explicitly governed emergency procedure for human intervention. These are illustrative policy choices for this scenario; an organization must choose its own values and consequences.

The scenario portfolio should include a denied approval, an approval arriving after expiry, and a target that changes after approval but before execution. It should also cover duplicated delivery, a tool timeout after a request was accepted, and two agents proposing incompatible remediations. Define the acceptable outcome for each situation before examining candidate designs. Otherwise, a simulation can produce a trace without providing a sound basis for deciding whether the behavior is acceptable.

The phrase safe state needs particular care. During an outage, the architecture may be able to prevent further automated change while leaving the service degraded. That is different from promising that the

service is healthy. Specify which actions are prohibited, which evidence is preserved, who is notified, and what recovery remains possible when automation stops.

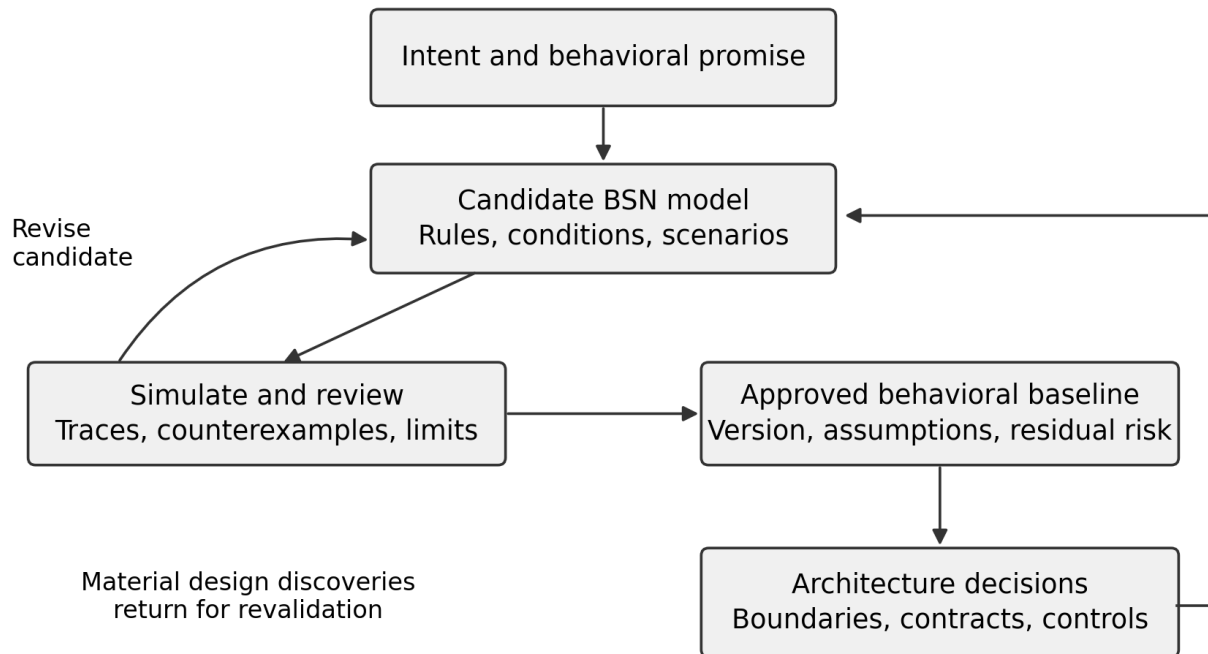


Figure 1. From behavioral promise to architectural decision.

5 Allocate responsibilities and authority deliberately

Use the promise to distinguish diagnosis, authorization, execution, and evidence recording. Diagnosis produces a proposal; authorization determines whether a specific proposal may proceed; execution changes the target system. Keeping these responsibilities explicit makes it possible to examine their separate failure modes. Their implementation boundaries should reflect the consequences of combining them.

For this case, a useful candidate places the authorization decision outside the agent's discretionary reasoning and enforces it at the execution boundary. The executor accepts only an action tied to valid authorization and checks the target and permitted parameters. The agent's tools and credentials should make the same boundary effective in practice. A policy check in an agent prompt cannot enforce the promise if the agent also has an unrestricted route to the production tool.

Draw the trust boundary and simulate attempts to cross it through alternate paths. Include direct tool invocation, delegated execution, changed parameters, expired approval, and an unavailable policy service. Record which component rejects each request and how rejection becomes visible to operators. If these paths are ignored, an architecture can present a convincing approval workflow while allowing equivalent actions elsewhere.

Separate processes or services may provide useful isolation, but they also add communication and recovery work. A smaller system might enforce these responsibilities within a tightly controlled application using

distinct permissions and interfaces. Choose the allocation that can enforce the promise under the stated threat and failure assumptions. Document the reason so future simplification does not accidentally remove the control.

6 Design interfaces around outcomes and uncertainty

An interface contract should explain what a request means and what the caller can conclude from each response. A successful HTTP response, message acknowledgment, or accepted command may indicate different stages of work. Define acceptance, authorization, execution, completion, rejection, cancellation, and unknown outcome separately where they matter. This prevents callers from treating transport success as evidence that the behavioral promise was fulfilled.

For remediation, carry a stable action identifier, incident identifier, target, parameters, authorization reference, expiry, and applicable state precondition. Define duplicate handling against the action identifier and explain how long the executor preserves that history. Bind approval to the action content so that a changed command cannot inherit permission intended for another command. The implementation must preserve these semantics across retries and restarts.

A timeout deserves its own scenario. If the executor applied a change but its response was lost, the agent should reconcile the action state or inspect the target before issuing another attempt. Where possible, make the operation idempotent; where that is impossible, define duplicate suppression and recovery within explicit limits. Blind retries can turn a recoverable communication failure into repeated production changes.

Select synchronous calls, durable messaging, or orchestration according to these obligations. Durable messaging can support delayed work, but requires decisions about ordering, expiry, replay, and consumer recovery. Synchronous calls can simplify immediate feedback, but couple availability and response time across participants. Simulate the consequential sequences and keep assumptions visible when the selected model cannot represent the relevant timing or failure behavior.

7 Give state and evidence clear owners

The incident-response workflow creates several kinds of state: observations, proposals, approvals, execution attempts, and outcomes. Decide which participant owns each authoritative record and which other participants hold derived views. An agent's working memory should not be the only record of whether a production change was authorized or completed. Durable ownership allows another process or operator to recover the workflow after interruption.

Approval and execution introduce a particularly important consistency question. The system must determine whether an authorization is still valid when the target has changed or another remediation has intervened. State preconditions, bounded approval lifetimes, and serialized access to a target can help, depending on the action. A simulation should examine the gap between checking a condition and performing the change, rather than assuming that the two events occur atomically.

Audit design must handle the same gap. Persisting an execution intent before a side effect and reconciling its outcome afterward can preserve useful evidence when the tool or network fails. The architecture must

still account for crashes between those steps and for audit records whose outcome remains unknown. A database entry labeled successful should reflect verified evidence under the chosen contract.

Record the policy version, relevant model or agent version, approval basis, tool request, and observed result needed to reconstruct the decision. Apply access and retention controls appropriate to operational and personal data. Avoid collecting sensitive payloads merely because more logging is easy. Without deliberate evidence design, an organization may retain large volumes of telemetry while remaining unable to explain one consequential action.

8 Turn quality attributes into architectural scenarios

Terms such as reliable, secure, scalable, and observable need operating conditions and acceptance criteria. For the incident-response agent, reliability includes recovering an interrupted workflow without repeating an unsafe action. Security includes rejecting an unauthorized command even when the agent proposes it confidently. Performance includes timely escalation when approval or execution cannot complete within the accepted response window.

Define the workload and environment that make each promise meaningful. Specify concurrent incidents, dependency response ranges, target contention, queue limits, and the duration of relevant outages. Separate simulated timing assumptions from measurements obtained from prototypes or production. Without this distinction, precise-looking results can conceal an unrealistic environment model.

The following matrix connects questions to design choices. Each row should lead to scenarios with explicit expectations and evidence requirements. Prioritize questions by consequence and uncertainty rather than attempting to simulate every detail at once. Findings then guide which prototypes, implementation tests, and operational measurements are necessary.

Quality concern	Scenario to examine	Design choices
Reliability	Execution stops after a tool accepts a change	Durable state and reconciliation
Security	An agent requests an unapproved action	Scoped credentials and enforcement
Performance	Approval outlasts the response window	Expiry and escalation
Concurrency	Two actions conflict on one target	Reservations or version checks
Observability	A tool times out after changing the target	Correlated evidence and outcome checks

Table 1. Behavioral scenarios guide quality decisions.

Resilience also requires deciding what happens when the human or infrastructure response is slower than hoped. A queued proposal can become stale while waiting for approval, and a recovered service can make a previously useful remediation harmful. Include expiry, cancellation, and re-evaluation in the design. Ignoring these transitions can leave the system executing yesterday's correct decision against today's different conditions.

9 Compare candidate architectures using the same promises

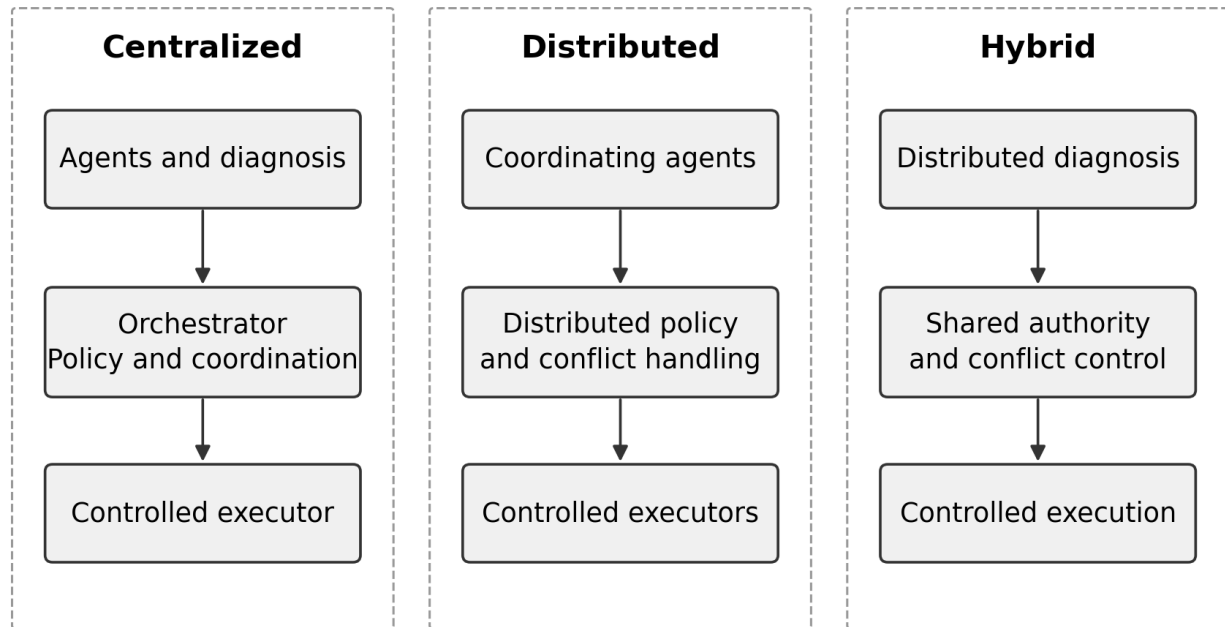
Evaluate alternatives against a shared scenario portfolio and the same acceptance criteria. For this case, consider a centralized orchestrator, distributed agent coordination, and a hybrid with distributed diagnosis and controlled authorization and execution. Each can potentially fulfill the promise if its controls and failure behavior are adequately designed. The decision depends on the operating context and the evidence that each candidate produces.

A centralized orchestrator can offer a clear workflow owner and coordination point. It also requires careful recovery, availability, and scaling decisions around that role. Distributed coordination can support independently progressing participants, while introducing more demanding state, conflict, and message semantics. A hybrid can place reasoning near the relevant information while concentrating authority, but must still address availability and latency at the enforcement boundary.

Do not rank candidates using invented precision. If a model represents control flow but does not contain calibrated service timing, it can help expose an authorization bypass or missing transition but cannot establish production throughput. Record the model's suitability for each decision and identify evidence still required. Simulation findings become useful when their limits are as visible as their successes.

Create an architecture decision record tied to the model and scenario versions. Describe the selected allocation, alternatives considered, counterexamples found, residual risks, and implementation obligations. Identify which changed assumptions would reopen the decision. This preserves the engineering reason behind a topology and makes later revisions more accountable.

Shared promise: authorized action, conflict control, traceable outcome



Compare the same scenarios and assumptions across all candidates
Every candidate requires human approval paths and durable evidence

Figure 2. One behavioral promise across alternative architectures.

10 Model multi agent interaction and human control

Multiple agents can create system behavior that is difficult to infer from isolated evaluations. One agent may diagnose capacity pressure while another responds to a security event by restricting access. Their individually plausible proposals may conflict on the same target. Model delegation, shared resources, competing goals, and the authority available to every participating agent.

For the incident-response case, give the executor a way to reject or defer conflicting actions against a target. Define whether coordination uses a workflow owner, a reservation, a version precondition, or another explicit mechanism. Simulate loss of coordination, stale observations, repeated delegation, and agents that continue acting after cancellation. If these interactions are omitted, adding another competent agent can weaken the system's existing behavioral promises.

Human approval must be usable during a real incident. Present the proposed action, evidence, expected effect, risk, alternatives, and expiry in terms an operator can assess. Define who may approve, how a substitute approver is selected, and how an emergency override is recorded. An approval button without sufficient context transfers accountability to a person without supporting an informed decision.

AI can help propose scenarios, identify omissions, and summarize traces for review. Human domain experts must still determine whether expectations are correct and residual risk is acceptable. Finite simulations explore a defined set of modeled conditions; they cannot establish acceptable behavior for every possible

agent output or environment. Combine simulation with bounded permissions, implementation evaluation, and production monitoring.

11 Preserve the promise through delivery and operation

Architecture should establish the observation points needed to assess critical promises throughout the value stream. Link the approved behavior to design responsibilities, implementation tests, integration evidence, release criteria, deployment safeguards, and production telemetry. Use stable promise and scenario identifiers so the evidence can be followed across these artifacts. This makes behavioral conformance a continuing engineering responsibility.

For the incident-response agent, integration testing should challenge expiry, duplicate requests, target changes, and uncertain tool outcomes against the actual interfaces. Release approval should examine evidence that the authorization boundary is enforced and that recovery behaves as specified. Deployment should verify the intended permissions, policy versions, and routes to production tools. A passing build cannot establish these properties on its own.

In production, observe attempted actions, rejected requests, approval delays, reconciled outcomes, and conflicts between agents. Compare material deviations with the modeled expectations and investigate whether the implementation, environment, or promise needs correction. A new tool route or broader credential may change behavior even if the architecture drawing remains unchanged. Feed such discoveries back to the living model for impact analysis and targeted resimulation.

SDSE and the BSN Workbench provide an implementation of this continuous behavioral discipline. Increasing fidelity from L0 toward L3 should follow the evidence needed for the next decision, with assumptions and limitations documented at each step. A higher fidelity label does not remove the need to check model suitability. Material change creates a new candidate baseline for review and approval rather than silently rewriting accepted history.

12 Begin with one consequential architectural decision

Choose a bounded workflow with meaningful interaction risk and a design choice that can still be changed. The incident-response example is suitable because authority, timing, and recovery directly affect production consequences. Name the Behavior Model Owner and involve product, architecture, development, security, SRE, and the operators who will use the system. Start with the smallest model that can expose the uncertainty behind the decision.

Write the promise, unacceptable outcomes, and operating assumptions before expanding the model. Express the participants and exchanges in BSN, then challenge normal behavior and a focused set of failure and boundary scenarios. Compare at least two credible allocations and preserve the evidence supporting the selected design. Connect the resulting obligations to implementation tests and operational signals before the pilot is considered complete.

Measure earlier discovery of ambiguous authority, missing failure handling, contradictory expectations, and infeasible assumptions. Track whether those findings changed an architectural decision and whether the change avoided later rework. Also measure the time required to update the model when integration or

production reveals material behavior. Counts of BOXes, diagrams, or simulation runs provide little value unless they improve engineering decisions.

Scale the practice after the pilot demonstrates useful learning. Reuse conventions, scenario patterns, and traceability practices while retaining the context of each system. Assign time and ownership for model maintenance as part of ordinary engineering work. A model that loses contact with implementation and operation eventually becomes another historical document.

Conclusion

Return to the production incident-response agent and ask what the architecture now makes possible. Diagnosis produces a traceable proposal, authority is enforced at the point of action, approvals expire under defined rules, and uncertain outcomes trigger reconciliation. Operators receive evidence they can assess and a clear escalation path when automation must stop. These behaviors give the structural design an accountable purpose.

SDSE and BSN help architects establish that purpose, examine alternatives, and preserve the promise as the system evolves. The engineer's judgment remains central because every model contains abstractions and every architecture makes tradeoffs. The improvement is that assumptions, consequences, and supporting evidence become available for deliberate review. Architecture decisions can then be judged by how credibly they realize the behavior that people depend on.

Further reading and participation

For additional information about SDSE, Behavior Simulation Notation, and the BSN Workbench, visit www.PrimaModus.com. Architects, senior engineers, security practitioners, SREs, platform engineers, and agentic AI teams interested in the trial cohort are invited to contact mhornbeek@engineeringdevops.com. Participants can help evaluate realistic scenarios and influence the development of the notation and workbench.

Related PrimaModus reading includes *Where Simulation Fits in the Software Lifecycle* and *Behavior as the Source of Truth Across the Value Stream*. The book *Simulation-Driven Software Engineering* by Marc Hornbeek, Debashis Bhattacharyya, and Jitin Dhanani provides further context for the behavior-first discipline. These resources complement the architectural practices described here.

About the Author

Marc Hornbeek, known as DevOps_the_Gray, is an award-winning engineer, author, educator, executive consultant, and keynote speaker with five decades of experience across software and systems engineering, testing, automation, DevOps, DevSecOps, SRE, and platform engineering. He is the originator of Simulation-Driven Software Engineering and leads the development of Behavior Simulation Notation and the BSN Workbench. He founded PrimaModus to help organizations model, simulate, validate, and engineer software that behaves as intended.

Contact: mhornbeek@engineeringdevops.com | www.PrimaModus.com

Prepared with assistance from OpenAI ChatGPT in drafting and diagram development.