

Where Simulation Fits in the Software Lifecycle

Why SDSE Encapsulates the Entire Lifecycle - and Why the Behavior Model Must Stay Alive

The central proposition

Simulation is not a box in the lifecycle. It is the evidence system that connects intent to operational reality.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

August 2026

Executive Summary

The question “Where does simulation fit in the software lifecycle?” appears to invite a single-stage answer. Some teams place simulation in ideation, others after requirements, others in architecture or design, and still others in testing, resilience engineering, capacity planning, or operations. Each answer captures part of the truth. The error is treating simulation as though it belongs to one box.

Simulation-Driven Software Engineering (SDSE) takes a lifecycle-wide view. Simulation begins lightly as soon as intent can be expressed through candidate behavior. It becomes a formal validation activity after a Candidate Specification exists and before architecture and detailed design commitments become expensive. It then continues at increasing fidelity through architecture, design, implementation, integration, deployment, operation, and learning. SDSE therefore encapsulates the lifecycle without replacing its established disciplines [1].

The continuity mechanism is a living behavior model: a versioned, traceable, executable engineering asset connected to the current Validated Behavioral Baseline. The baseline is authoritative, but not immutable. Architecture contributes structural knowledge; design contributes more precise behavior; implementation contributes executable evidence; integration contributes interaction and conformance evidence; and operation contributes production evidence. Material discoveries return to the same model for resimulation, risk review, and approval of a new baseline.

Core idea

Simulation fits before architecture and design as a formal behavioral validation gate - and it fits everywhere afterward as a continuous evidence and learning capability.

1. Early simulation improves the quality of intent by exposing assumptions before they become commitments.
2. Formal pre-design simulation challenges candidate behavior under normal, boundary, failure, load, security, and recovery conditions.
3. Architecture, detailed design, TDD, implementation, integration, and operations add evidence to the same behavioral promise.
4. A named Behavior Model Owner protects model integrity, currency, traceability, and lifecycle continuity.

Contents

- 1. The Wrong Shape of the Question**
- 2. The SDSE Lifecycle Principle**
- 3. Where Simulation Fits at Every Stage**
- 4. The Formal Pre-Design Validation Gate**
- 5. Architecture, Design, and Fidelity**
- 6. TDD, Implementation, and Conformance**
- 7. Operations as the Most Honest Evidence**
- 8. The Living Behavior Model**
- 9. Ownership, Roles, and Decision Rights**
- 10. A Practical Maintenance Rhythm**
- 11. Integration with the Delivery and Operating Model**
- 12. Adoption Roadmap and Measures**

Conclusion

References

About the Author

1. The Wrong Shape of the Question

Engineering specialists naturally see simulation through the lens of their work. Product leaders use it to explore journeys and policies. Requirements and systems engineers use it to resolve ambiguity. Architects use it to compare boundaries and tradeoffs. Designers use it to examine state, timing, and failure handling. Test engineers use it to derive and challenge expected outcomes. Security and reliability specialists use it to exercise misuse, degradation, and recovery. Operations teams use it to understand capacity, resilience, and incidents.

These views do not compete. They reveal that simulation changes purpose and fidelity across the lifecycle. The useful question is not “Which stage owns simulation?” It is “What decision requires behavioral evidence at this stage, and what model fidelity is proportionate to that decision?”

Reframed question

At each lifecycle stage, ask what behavior must become visible, executable, and reviewable before the organization makes the next consequential commitment.

This distinction prevents two common mistakes. The first is postponing simulation until testing, after architecture and code have already encoded unvalidated assumptions. The second is treating an early model as disposable once implementation begins, breaking the connection between intent, delivery, and operational evidence.

2. The SDSE Lifecycle Principle

SDSE is often summarized as “simulate before you build.” That phrase is correct but incomplete. The discipline is not merely an early prototyping technique. It creates a continuous behavioral evidence system around the complete software lifecycle. Product management, requirements engineering, architecture, detailed design, TDD, testing, security engineering, DevSecOps, SRE, platform engineering, deployment, and operations remain necessary. SDSE connects them with a persistent question: are we still building, integrating, operating, and evolving the behavior we intend to promise?

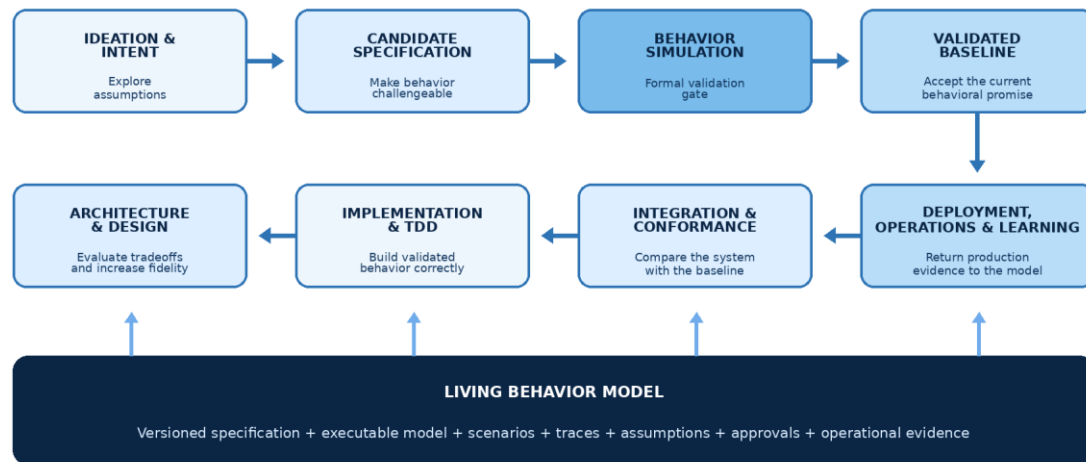
The lifecycle begins with ideas and stakeholder intent. Intent is organized into a Candidate Specification: an explicit but provisional description of behavior, constraints, quality attributes, interfaces, policies, risk assumptions, and failure responses. The word candidate is important. The specification is detailed enough to simulate, but it has not yet earned the authority of a validated promise.

Behavior simulation produces traces, counterexamples, sensitivity findings, expectation results, unresolved assumptions, confidence limits, and change proposals. Teams refine the specification and executable model together. When the evidence is sufficient for the decision and its risk, accountable

stakeholders approve a versioned Validated Behavioral Baseline. Downstream engineering then proceeds from that evidence-backed baseline rather than an untested interpretation.

WHERE SIMULATION FITS IN THE SOFTWARE LIFECYCLE

It begins lightly with intent, becomes a formal pre-design gate, and continues as lifecycle evidence.



Simulation does not replace the lifecycle disciplines. It connects them with one persistent behavioral promise.

Figure 1. Simulation begins lightly, becomes a formal pre-design gate, and remains active through operational learning.

3. Where Simulation Fits at Every Stage

Ideation and intent: simulation begins lightly

At the beginning, the team is not trying to prove a design. It is clarifying desired outcomes, stakeholders, boundaries, unacceptable outcomes, risk tolerances, decision criteria, and important uncertainties.

Simulation contribution. Use low-fidelity scenarios, journey sketches, workflow models, state concepts, simplified system dynamics, and misuse narratives to explore alternative concepts and expose hidden assumptions.

Evidence produced. Questions, scenario fragments, stakeholder conflicts, boundary conditions, preliminary invariants, and a clearer statement of intent.

Decision enabled. Which concept or behavioral direction deserves more detailed specification and investment?

Candidate specification: enough behavior to challenge

Requirements are not bypassed in SDSE. Functional behavior, quality attributes, policies, constraints, interface expectations, security properties, service promises, and failure responses are organized into a provisional but executable candidate.

Simulation contribution. Translate the critical rules, participants, operating conditions, states, interactions, scenarios, and expected outcomes into a model that can be run and revised.

Evidence produced. A traceable Candidate Specification, executable behavior model, initial scenario portfolio, explicit assumptions, and known omissions.

Decision enabled. Is the candidate sufficiently coherent and explicit to enter formal behavioral validation?

Behavior simulation: the formal pre-design gate

Once candidate behavior exists, simulation becomes a formal engineering activity. The team challenges the proposed behavior before architecture and detailed design constrain the solution space.

Simulation contribution. Execute normal, edge, timing, concurrency, load, dependency failure, degraded infrastructure, malicious action, unexpected sequence, recovery, and human-intervention scenarios according to risk.

Evidence produced. Traces, failed expectations, counterexamples, sensitivity results, contradictions, infeasible assumptions, residual risks, and revised candidate behavior.

Decision enabled. Is the evidence sufficient to approve the current behavioral promise as a Validated Behavioral Baseline?

Validated baseline: accept the current promise

The validated baseline is the versioned behavioral promise that downstream disciplines are expected to realize. It records not only accepted outcomes, but also the evidence, scope, fidelity, assumptions, limitations, and accountable approvals behind them.

Simulation contribution. Run final regression and risk-focused scenario sets against the proposed baseline; archive the exact model, specification, scenarios, traces, expectations, and decision record.

Evidence produced. An authoritative baseline, residual-risk statement, downstream constraints, trace links, and a clear supersession rule.

Decision enabled. May architecture and design commit resources against this version of the promised behavior?

4. The Formal Pre-Design Validation Gate

The pre-design gate is the most distinctive placement of simulation in SDSE. It occurs after the organization has expressed intent as candidate behavior and before architecture and detailed design decisions make change expensive. The gate does not demand a complete model of the future implementation. It demands enough fidelity to challenge the promises and risks that should constrain downstream work.

Gate element	Minimum evidence
Candidate behavior	Critical rules, states, interactions, constraints, qualities, interfaces, and operating assumptions are explicit enough to execute.
Risk-based scenario portfolio	Normal, boundary, failure, security, performance, recovery, degraded, and misuse conditions are selected according to consequence and uncertainty.
Observable evidence	Runs produce traces, outcomes, failed expectations, counterexamples, sensitivity findings, and unresolved assumptions rather than only pass/fail summaries.
Model suitability	Scope, abstractions, timing, data, probability, environment, and known exclusions are appropriate for the decision being made.
Cross-functional review	Product, domain, architecture, engineering, test, security, reliability, and operations perspectives are represented where material.
Accountable decision	Authorized humans accept the baseline and residual risk, request revision, or reject the candidate.

Gate element	Minimum evidence
Controlled change	A material change creates a new candidate version that must be resimulated and revalidated before superseding the baseline.

Table 1. The formal gate converts a provisional specification into an evidence-backed behavioral baseline.

A gate, not a freeze

The baseline is authoritative, not immutable. It controls commitments while preserving the ability to learn and change through explicit versioning and revalidation.

Fidelity should follow risk

Not every behavior requires the same simulation detail. A low-consequence workflow decision may be supported by discrete scenarios and simplified state. A safety-, security-, finance-, or mission-critical interaction may require precise timing, probabilistic variation, resource constraints, environment models, and specialized validation. The correct question is whether the model is fit for the decision, not whether it is universally complete.

Fidelity level	Typical content	Decision use
Conceptual	Actors, outcomes, alternatives, high-level state, and major constraints.	Ideation, stakeholder alignment, early policy exploration.
Behavioral	Executable rules, sequences, interfaces, modes, expectations, and adverse scenarios.	Candidate validation and cross-system interaction decisions.
Architectural	Allocated responsibilities, boundaries, protocols, timing budgets, capacity, and resilience patterns.	Architecture comparison and design constraint.
Implementation-informed	Concrete contracts, retry and timeout behavior, concurrency, data rules, resource use, and test mappings.	Detailed design, TDD, integration, and conformance.
Operationally calibrated	Production distributions, usage, incident paths, latency, capacity, cost, environment, and human response.	Resilience, change impact, operations, and lifecycle learning.

Table 2. Simulation fidelity increases as decisions require more precise evidence.

5. Architecture, Design, and Fidelity

The validated baseline constrains and informs architecture; it does not prescribe every architecture decision. Architects allocate responsibilities, define boundaries, compare alternatives, examine interfaces, explore performance and resilience, and evaluate security controls against the promised behavior. Each decision adds knowledge that can expose an earlier assumption or require more detailed simulation.

Detailed design increases fidelity. State transitions, component contracts, data consistency, concurrency, retry and timeout behavior, resource use, failure handling, recovery, and observability become concrete. If this knowledge changes the promised behavior or invalidates an assumption, the affected behavior returns to candidate status. The model is updated, targeted scenarios are rerun, and the baseline is revalidated before the change is treated as authoritative.

Two-way flow

Design is downstream of behavioral validation, but design discoveries are also upstream inputs to the next validation cycle.

Design lens	Question returned to the behavior model
Responsibility	Which element owns each state, decision, control, and recovery action?
Boundary	Which interfaces and trust boundaries must preserve the validated promise?
Timing and load	Do latency, ordering, capacity, and resource assumptions remain feasible?
Failure and recovery	What happens under partial failure, retry, duplicate, timeout, rollback, and degraded operation?
Security and policy	Can misuse, privilege, privacy, data handling, or policy interactions violate the promise?
Observability	Which signals are required to demonstrate conformance and detect model-reality divergence?

Table 3. Architecture and design enrich the model with structural and operational constraints.

6. TDD, Implementation, and Conformance

Test-Driven Development and SDSE solve different but connected problems. SDSE asks whether the proposed system behavior is the right behavior across interactions, environments, failures, quality constraints, and risk conditions. TDD helps developers implement validated behavior correctly in small increments. TDD belongs primarily in detailed design and implementation; SDSE supplies the system-level behavioral promise that should inform those tests.

Tests should be derived from validated scenarios, expected outcomes, invariants, and component contracts. They should not become an undocumented competing specification. When a test exposes a coding defect, the implementation changes. When it exposes missing, contradictory, infeasible, or changed behavior, the discovery returns to the living behavior model for resimulation and revalidation.

Evidence source	Conformance question	Connection to the model
Unit and TDD tests	Does a component implement its allocated behavior and contracts correctly?	Validated rules, states, interfaces, invariants, and design contracts.

Evidence source	Conformance question	Connection to the model
Contract tests	Do collaborating components preserve agreed interaction behavior?	Modeled interfaces, message expectations, sequencing, errors, and recovery.
Acceptance tests	Does the implementation fulfill stakeholder-visible scenarios and outcomes?	Validated scenarios and expected outcomes.
Security tests	Do controls and behavior hold under threat, misuse, privilege, and policy conditions?	Threat and misuse scenarios, trust boundaries, and security expectations.
Performance and resilience tests	Does the assembled system conform under load, variation, degradation, and failure?	Timing, capacity, retry, timeout, failure, and recovery models.
Operational checks	Does deployed behavior continue to match the baseline in its real environment?	SLOs, invariants, telemetry requirements, incident scenarios, and divergence rules.

Table 4. Tests at every level should trace to the same behavioral promise.

Divergence is a decision

When implementation behavior differs from the model, do not quietly patch the test. Decide explicitly whether the implementation is wrong, the promise must change, or both.

7. Operations as the Most Honest Evidence

Operation reveals conditions no laboratory can fully predict. Telemetry, service-level indicators, incidents, near misses, real usage, environmental change, cost behavior, dependency behavior, and human response expose the difference between the world the team modeled and the world the system inhabits. This evidence is not a reason to abandon simulation. It is how the living model becomes more credible.

Operational evidence should be reviewed for materiality. A transient metric variation may require no model change. A repeated interaction pattern, unmodeled failure path, violated assumption, changed workload, new adversary technique, or human workaround may require a new candidate version. The team then performs targeted resimulation, updates trace links and controls, reviews risk, and approves or rejects the proposed baseline change.

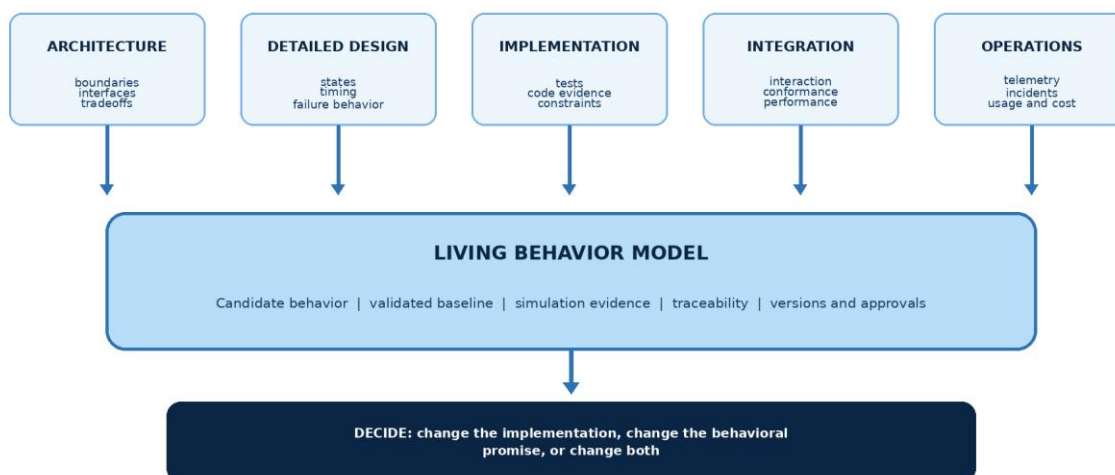
Operational evidence	Question for the living model
Telemetry and SLOs	Do operating distributions and service outcomes remain within modeled expectations and promises?
Incidents and near misses	Which scenario, assumption, interaction, or control was missing or wrong?
Usage and workflow evidence	Are users, operators, or agents exercising paths the model did not anticipate?

Operational evidence	Question for the living model
Environmental change	Have dependencies, infrastructure, regulations, threats, scale, or business constraints changed?
Cost and capacity evidence	Do resource and economic outcomes reveal infeasible behavioral assumptions?
Human response	Do overrides, workarounds, escalation, or decision behavior change system outcomes?

Table 5. Production evidence closes the loop by challenging assumptions and model fidelity.

REALITY CLOSES THE LOOP THROUGH THE LIVING BEHAVIOR MODEL

Every discipline adds evidence; every material discovery returns to the same controlled promise thread.



Material change creates a new candidate version, targeted resimulation, risk review, and an approved new baseline.

Figure 2. Every material discovery returns to the same living behavior model and controlled decision process.

8. The Living Behavior Model

A behavior model should not be a temporary analyst artifact, a presentation animation, or a pre-code experiment that disappears when implementation starts. In SDSE, it remains versioned, traceable, executable, and synchronized with the current validated baseline. It is the system's executable memory and the single promise thread connecting intent, specification, simulation, architecture, design, tests, deployment, operation, and learning.

Living-model element	What must remain controlled
Intent and obligations	Stakeholders, desired outcomes, unacceptable outcomes, policies, constraints, risks, and decision criteria.

Living-model element	What must remain controlled
Candidate and validated behavior	Current proposed behavior and the approved baseline it may supersede.
Executable model	Participants, interfaces, states, operating conditions, rules, interactions, scenarios, and expectations.
Evidence	Traces, results, counterexamples, sensitivities, test mappings, conformance findings, telemetry, and incidents.
Fidelity and limits	Scope, abstractions, exclusions, uncertainty, synchronization, and decisions the model is or is not fit to support.
Traceability	Links to requirements, architecture, code, tests, controls, deployments, runbooks, SLOs, changes, and incidents.
Governance	Owners, reviewers, approvals, delegated authority, residual risks, supersession, and retention.

Table 6. The living behavior model carries both the promise and the evidence behind it.

Why continuity matters

5. It reduces misunderstanding while behavior is still inexpensive to change.
6. It gives architects an executable basis for comparing decisions instead of independent interpretations of static requirements.
7. It aligns simulation, acceptance, contract, TDD, integration, security, resilience, and operational checks to one promise.
8. It makes change impact visible across scenarios, interfaces, tests, controls, and downstream artifacts.
9. It turns incidents and telemetry into engineering knowledge rather than isolated tickets.
10. It gives AI-enabled delivery a durable behavioral boundary while humans retain authority over meaning and acceptance.
11. It makes governance a continuous engineering outcome supported by traceable evidence and decisions.

The governing rule

A validated baseline is authoritative, not immutable. Material change produces a new candidate version that must be resimulated, reviewed, and approved.

9. Ownership, Roles, and Decision Rights

The model cannot belong only to testing, architecture, requirements, or a modeling specialist. Every lifecycle discipline contributes knowledge. Yet distributed authorship must not become shared

ambiguity. The practical operating rule is: model authorship is distributed, but accountability for model integrity is singular.

Every significant product or system should have a named Behavior Model Owner. Depending on the organization, the role may be filled by a chief engineer, lead systems engineer, solution architect, senior product engineer, or explicit SDSE lead. The title can vary; the accountability should not. The owner establishes conventions, coordinates contributions, controls versions and traceability, determines whether evidence represents a material change, organizes revalidation, and ensures the approved model reflects the current promise.

Role	Contribution	Primary accountability
Product / business owner	Outcomes, users, value, priorities, acceptance criteria, and unacceptable business outcomes.	Accepts the business promise and authorizes changes to intent.
Behavior Model Owner	Model structure, conventions, integrated contributions, versions, traceability, and revalidation.	Model integrity, currency, and lifecycle continuity.
Domain experts	Rules, exceptions, constraints, realistic scenarios, and human workflows.	Domain correctness and exposure of invalid assumptions.
Requirements / systems engineers	Candidate behavior, interfaces, qualities, constraints, and coherence.	Continuity from intent through specification and model.
Architects and designers	Boundaries, contracts, timing, failure behavior, allocation, and tradeoff evidence.	Behavioral consistency of architecture and design.
Developers and test engineers	Implementation evidence, TDD, contract, acceptance, and conformance results.	Implementation conformance and return of behavioral discoveries.
Security, safety, and risk	Threat, misuse, hazard, privacy, policy, and control scenarios.	Specialized assurance and informed risk acceptance.
SRE and operations	SLO assumptions, telemetry, incidents, capacity, recovery, cost, and environment.	Connection between the model and production reality.
Governance / approval authority	Evidence review, residual risk, materiality, and downstream impact.	Approval or rejection of a new validated baseline.

Table 7. Contribution is cross-functional; integrity and approval remain explicit human accountabilities.

AI can assist every role by proposing scenarios, finding omissions, generating variations, correlating traces, identifying affected artifacts, and suggesting model changes. AI should not become the accountable owner. Humans who understand the domain, consequence, risk, and organizational promise must approve what the model means and when a new baseline is accepted.

10. A Practical Maintenance Rhythm

A living model requires an operating cadence. The objective is not continuous administrative activity. It is a small number of reliable triggers that keep material behavioral knowledge from remaining trapped in architecture notes, code, tests, tickets, incident reports, or individual memory.

Trigger	Required model action	Typical participants
Intent change	Update outcomes, constraints, assumptions, and affected candidate scenarios before downstream work proceeds.	Product / business owner and Behavior Model Owner
Architecture or design review	Record new contracts, boundaries, resource assumptions, failure behavior, and tradeoff decisions.	Architects, designers, domain and model owners
TDD and implementation	Link tests to validated behavior; return specification gaps instead of hiding them in fixtures or code.	Developers, test engineers, Behavior Model Owner
Integration and release	Compare observed behavior with simulated traces; classify every material divergence explicitly.	Integration, security, performance, quality, release owners
Operations review	Assess telemetry, incidents, near misses, SLO results, usage, cost, and environment for material evidence.	SRE, operations, product, risk, model owner
Material baseline change	Preserve the prior baseline, create a candidate, rerun affected simulations and regressions, review risk, approve, and update trace links.	Behavior Model Owner and delegated approval authority

Table 8. Maintenance should be triggered by material knowledge, not by ceremonial documentation schedules.

Materiality test

12. Does the discovery change a stakeholder-visible outcome, invariant, policy, or unacceptable outcome?
13. Does it change an interface, trust boundary, operating condition, performance assumption, safety or security property?
14. Does it expose a scenario, sequence, failure mode, recovery path, or human interaction not represented by the current baseline?
15. Would a reasonable architecture, test, control, SLO, runbook, or approval decision change if this evidence were known?
16. Does it alter the model's suitability, uncertainty, fidelity, or consequence of error for an intended decision?

11. Integration with the Delivery and Operating Model

SDSE should not become an isolated modeling program. The living behavior model gains value when it serves as a common behavioral thread across product management, architecture, engineering, DevOps,

DevSecOps, SRE, platform engineering, testing, operations, risk, and AI governance. Each discipline contributes evidence and receives actionable artifacts from the same promise.

Discipline	Evidence contributed	Value returned from the living model
Product and requirements	Intent, value, obligations, constraints, changes	Executable scenarios, expectation evidence, ambiguity and risk findings
Architecture and design	Boundaries, allocation, contracts, timing, capacity, tradeoffs	Behavioral constraints, comparative evidence, impact analysis
Development and TDD	Implementation choices, code evidence, component limits	Reference behavior, states, interfaces, invariants, test derivation
Testing and quality	Coverage, defects, conformance, boundary findings	Scenario portfolio, expected outcomes, trace patterns, regression targets
DevSecOps	Threats, controls, misuse, pipeline and supply-chain evidence	Executable abuse scenarios, behavioral security gates, approval criteria
SRE and operations	SLOs, telemetry, incidents, recovery, capacity and cost	Failure and recovery models, observability needs, operational scenarios
Platform engineering	Golden paths, policy, infrastructure behavior, experience	Platform-consumer scenarios and consequences of platform change
AI governance	Uncertainty, agent action, escalation, evaluation, audit and risk	Behavioral boundaries, counterfactuals, human-control expectations

Table 9. The model is a shared behavioral contract, not a replacement for each discipline's native artifacts.

NIST's Secure Software Development Framework similarly emphasizes that secure practices should be integrated into each software lifecycle implementation rather than treated as a detached activity [2]. NASA software requirements and systems-engineering guidance demonstrate lifecycle-wide verification, validation, and the disciplined use of validated models and simulations in high-consequence contexts [3][4]. SDSE applies the same continuity principle specifically to executable software behavior.

12. Adoption Roadmap and Measures

Adoption should begin with a bounded, consequential interaction where behavior is difficult to infer and the next commitment can still be changed. The first goal is to demonstrate earlier learning and better decisions, not to maximize model size or replace existing lifecycle tooling.

1. Select a decision-centered pilot. Choose a workflow, service interaction, policy, control function, recovery path, or agent behavior with meaningful uncertainty, cross-component risk, or repeated rework.

2. Name the Behavior Model Owner. Make one person accountable for conventions, currency, versions, traceability, materiality, and revalidation coordination.

- 3. Clarify intent and consequence.** Define outcomes, stakeholders, unacceptable outcomes, assumptions, risk tolerance, operating context, and the consequence of a wrong decision.
- 4. Create the Candidate Specification.** Express the critical behavior, constraints, qualities, interfaces, failure responses, and policy conditions clearly enough to challenge.
- 5. Build the smallest useful executable model.** Represent only the participants, states, interactions, rules, conditions, scenarios, and expectations needed for the pilot decision.
- 6. Simulate by risk.** Exercise normal, boundary, timing, concurrency, failure, security, degraded, recovery, and human-intervention conditions that matter.
- 7. Review traces across disciplines.** Resolve ambiguity, identify infeasible assumptions, compare alternatives, and expose missing expectations or observability.
- 8. Approve the baseline.** Version the specification, model, scenarios, evidence, fidelity statement, limitations, residual risks, and accountable approval.
- 9. Connect delivery and operation.** Link architecture, code, tests, controls, deployment safeguards, telemetry, SLOs, runbooks, and incidents to the behavioral promise.
- 10. Close the learning loop.** Turn material production evidence into candidate model changes, targeted resimulation, updated controls, and a new approved baseline.
- 11. Scale based on demonstrated value.** Expand to additional behavior only after the pilot improves decisions, reduces rework, or lowers operational risk.

Measures that matter

Measure	Observable evidence
Earlier discovery	Contradictions, unsafe interactions, infeasible assumptions, and missing scenarios found before design or code.
Decision latency	Time from behavioral question to evidence-backed product, architecture, security, or risk decision.
Avoided rework	Late requirement, design, integration, acceptance, and recovery changes prevented by earlier simulation.
Promise traceability	Critical behaviors linked to scenarios, expectations, tests, controls, deployments, telemetry, and incidents.
Conformance quality	Material implementation-model divergences detected, classified, resolved, and governed.
Model-reality learning	Time from operational evidence to updated scenario, resimulation, decision, and baseline where required.
Delivery performance	Lead time, deployment frequency, failed deployment recovery time, change failure percentage, and deployment rework rate observed alongside behavioral learning [6].

Measure	Observable evidence
Risk reduction	Fewer severe incidents, escaped behavioral defects, policy violations, or unsafe changes attributable to misunderstood interaction.

Table 10. Measure whether the model improves learning and decisions, not how many diagrams it produces.

Conclusion

Simulation does not fit in one software lifecycle box. It begins when intent is concrete enough to express as candidate behavior. It becomes a formal behavioral validation gate before architecture and detailed design commitments. It then continues with changing purpose and increasing fidelity through architecture, design, TDD, implementation, integration, conformance, deployment, operations, and learning.

The living behavior model makes this continuity possible. It is the executable memory of the system's promise: what the organization intended, specified, simulated, approved, designed, implemented, tested, deployed, observed, and learned. Each lifecycle discipline can add evidence without losing the connection to intent. Material discoveries do not disappear into local artifacts; they return to the same controlled model for decision and revalidation.

SDSE therefore extends the familiar idea of simulation before code into a lifecycle discipline: build the right behavior, build it correctly, demonstrate that it remains conformant, and learn when operational reality teaches something new. Behavior Simulation Notation and the BSN Workbench provide a practical foundation for authoring, executing, tracing, reviewing, and maintaining that promise.

Final answer

Simulation fits before architecture and design as a formal validation gate - and everywhere afterward as continuous behavioral evidence and learning.

References

- [1] Marc Hornbeek, “Where Simulation Fits in the Software Lifecycle,” LinkedIn article, August 11, 2026. [Read the source article](#)
- [2] M. Souppaya, K. Scarfone, and D. Dodson, “Secure Software Development Framework (SSDF) Version 1.1,” NIST Special Publication 800-218, February 2022. [NIST SP 800-218](#)
- [3] NASA, “Software Engineering Requirements, Chapter 4: Software Engineering Life-Cycle Requirements,” NPR 7150.2B. [NASA lifecycle requirements](#)
- [4] NASA, “Product Verification,” NASA Systems Engineering Handbook reference, September 29, 2023. [NASA product verification guidance](#)
- [5] INCOSE, “Essential Systems Engineering: A Lifecycle Process,” accessed August 14, 2026. [INCOSE lifecycle process](#)
- [6] DORA, “Software Delivery Performance Metrics,” accessed August 14, 2026. [DORA metrics guide](#)
- [7] Marc Hornbeek, Debashis Bhattacharyya, and Jitin Dhanani, Simulation-Driven Software Engineering: A Behavior-First Discipline for Modern Systems, 2026. [Purchase on Amazon](#)

About the Author

Marc Hornbeek, a.k.a. DevOps_the_Gray, is an award-winning engineer, author, educator, executive consultant, and keynote speaker with five decades of experience across systems engineering, packet networks, software development, behavioral simulation, testing, automation, DevOps, DevSecOps, SRE, platform engineering, and intelligent software delivery. His early engineering work included integrated tools for packet-network simulation, performance, conformance, observation, and test automation. He is the originator of Simulation-Driven Software Engineering and leads the development of Behavior Simulation Notation and the BSN Workbench. Marc received the IEEE Region 6 Outstanding Engineer Award in 2016 and has authored multiple books and professional certification courses. He founded PrimaModus® to help organizations model, simulate, validate, and engineer software that behaves as intended.

Contact: mhornbeek@engineeringdevops.com