

How SDSE Improves the Role of the Software Engineer

From Code Producer to Behavioral Engineer, Systems Thinker, and
Trusted Decision-Maker

The central proposition

SDSE does not move software engineers away from engineering. It enables them to perform the most important parts of engineering earlier, more visibly, and with better evidence.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

August 2026

Executive Summary

Software engineering is entering a period in which producing code is becoming easier while establishing confidence in software behavior is becoming harder. AI assistants and coding agents can generate, modify, test, and document software at extraordinary speed. Cloud platforms, distributed services, policy engines, event streams, autonomous agents, third-party ecosystems, and continuous delivery make the resulting systems more dynamic and interaction-dependent. The critical engineering question is shifting from “Can we produce this code?” to “Do we have sufficient evidence that this system will behave as intended?”

The change does not make software engineers less important. It raises the value of the work that only accountable engineers and their organizations can perform: clarifying intent, reasoning about interactions, challenging assumptions, evaluating alternatives, understanding failure, setting limits on autonomy, interpreting evidence, accepting residual risk, and learning from operational reality. U.S. Bureau of Labor Statistics projections continue to show strong demand for software developers, quality assurance analysts, and testers, including demand associated with AI, robotics, IoT, and automation [2]. The role is evolving rather than disappearing.

Simulation-Driven Software Engineering (SDSE) supports that evolution. Engineers create a Candidate Specification, express consequential behavior in an executable model, simulate normal and challenging scenarios, examine traces and outcomes, revise what is wrong, and approve a versioned Validated Behavioral Baseline. The living behavior model then connects design, code, tests, deployment controls, telemetry, incidents, and learning. Behavior Simulation Notation (BSN) and the BSN Workbench provide a practical means to perform and maintain this work.

Core conclusion

SDSE elevates the software engineer from an implementer of assumed behavior to an engineer of validated behavior: a systems thinker, evidence producer, collaborator, and trusted decision-maker.

1. Engineers receive clearer behavioral intent before implementation begins.
2. Architecture and design alternatives can be compared through executable evidence.
3. Tests, controls, deployment decisions, and operational learning remain connected to the same promise.
4. AI-generated software can be evaluated against a validated behavioral baseline rather than accepted because it compiles or passes weak tests.
5. Career value shifts toward judgment, systems thinking, accountability, and cross-functional influence.

Contents

Executive Summary

- 1. The Changing Role of the Software Engineer**
- 2. What SDSE Adds to Software Engineering**
- 3. How SDSE Improves Daily Engineering Work**
- 4. How SDSE Improves Specific Engineering Roles**
- 5. SDSE as a Professional Force Multiplier**
- 6. Improving Engineering Collaboration**
- 7. Improving Quality and Productivity**
- 8. Human Accountability in AI-Native Engineering**
- 9. New and Improved Engineering Competencies**
- 10. The Emerging Behavioral Software Engineer**
- 11. Career Benefits for Software Engineers**
- 12. How Engineering Leaders Should Introduce SDSE**
- 13. Measuring Improvement to Engineering Roles**
- 14. Risks, Misconceptions, and Safeguards**

Conclusion

Appendix A. Role Development Worksheet

Appendix B. Role Catalogue at a Glance

References

About the Author

1. The Changing Role of the Software Engineer

Software engineering has never been only the production of source code. The IEEE Computer Society's SWEBOK describes a broad discipline spanning requirements, architecture, design, construction, testing, operations, maintenance, management, quality, security, professional practice, and other knowledge areas [1]. Yet many organizational workflows still position engineers late in the decision chain. Requirements are passed downstream, architecture is treated as fixed, and engineers are measured primarily by implementation throughput. Important behavioral questions are discovered while coding or after integration has made correction costly.

1.1 From writing code to engineering behavior

Code is one implementation of intended behavior. It is not the original source of that intent, nor is it reliable proof that the intended behavior will emerge when services, users, agents, policies, data, and operating conditions interact. The engineer who receives an apparently complete requirement must still interpret missing state, sequence, timing, exception, recovery, and authority rules. Those interpretations become design and code whether or not the team recognizes them as decisions.

SDSE makes those decisions visible earlier. The engineer participates in expressing the consequential behavior before implementation and can ask whether the proposed behavior is internally consistent, safe, testable, observable, recoverable, and aligned with the stakeholder promise. Coding remains important, but it begins from a stronger behavioral foundation.

1.2 Why faster coding increases the need for judgment

AI-assisted development compresses the time between an instruction and an implemented artifact. Google Cloud's 2025 DORA research characterizes AI as an amplifier: it can magnify the strengths of capable engineering systems and the weaknesses of struggling ones [3]. If the intent is sound and the feedback system is strong, speed can create value. If requirements are ambiguous, tests are weak, or ownership is unclear, speed can accelerate the wrong behavior into production.

The differentiating value of the software engineer therefore moves upstream and outward. Engineers must be able to frame the behavioral problem, decide which assumptions matter, direct AI toward bounded tasks, evaluate generated artifacts, and refuse false confidence. They also move downstream into operations by using real evidence to improve the behavioral model. The professional role becomes more lifecycle-oriented, not less technical.

1.3 The cost of implementation-centered discovery

When a behavioral error is discovered late, the organization may need to change requirements, architecture, code, tests, configuration, data, controls, documentation, release plans, runbooks, and partner commitments together. NASA systems engineering guidance notes that lifecycle costs become increasingly locked in early and that problems identified late cost considerably more to correct [6]. SDSE applies this economic principle to software behavior by moving high-value discovery toward the stage where change is still comparatively inexpensive.

Dimension	Implementation-centered	SDSE-enhanced
Primary artifact	Requirement, design, code, and test interpreted separately	Living behavior model connects the promise across artifacts
Engineer entry point	After major behavioral choices are assumed or approved	Before implementation, while behavior can still be challenged
Evidence	Code, reviews, and tests after implementation	Simulation evidence before implementation plus lifecycle evidence afterward
Change response	Correct each affected artifact after divergence appears	Revise the baseline and propagate targeted change
Professional value	Implementation throughput and technical specialization	Judgment, systems thinking, evidence, accountability, and influence

Table 1. SDSE broadens the engineer's role without diminishing implementation skill.

2. What SDSE Adds to Software Engineering

SDSE is a behavior-first engineering discipline. It does not replace requirements engineering, architecture, development, testing, security, DevOps, SRE, platform engineering, or governance. It supplies an executable behavioral reference that those practices can challenge, use, and improve. The model is deliberately proportional: teams invest fidelity and scenario coverage according to consequence, uncertainty, interaction complexity, autonomy, and the cost of discovering an error later.

2.1 The SDSE workflow

1. **Establish intent.** State the outcome stakeholders believe the system must produce and the consequences of being wrong.
2. **Create a Candidate Specification.** Describe participants, responsibilities, constraints, assumptions, acceptance conditions, and unresolved questions.
3. **Express consequential behavior.** Use BSN to model interactions, states, guards, decisions, timing, expectations, failure, and recovery.

4. **Simulate scenarios.** Execute normal, boundary, failure, misuse, adversarial, degraded, and recovery situations appropriate to the risk.
5. **Examine evidence.** Review traces, outcomes, counterexamples, violated expectations, ambiguity, and model limitations.
6. **Revise what is wrong.** Change the specification, behavior model, architecture concept, control, or assumption before implementation hardens it.
7. **Approve a baseline.** Version the Validated Behavioral Baseline, named owners, residual risk, evidence, and scope.
8. **Maintain the living model.** Connect implementation and operational evidence to the baseline and resimulate material change.

2.2 The living behavior model

A one-time design simulation can prevent an error, but a living behavior model creates continuity. Requirements identify the promise. Architecture assigns responsibilities. Code implements choices. Tests challenge the implementation. Deployment changes operating conditions. Telemetry and incidents reveal reality. The living model connects those stages so that a material change or observation can be evaluated against the same behavioral intent.

This continuity also creates organizational memory. Engineers joining a system can execute important scenarios and inspect why a baseline was accepted. They need not reconstruct every assumption from source code, old tickets, diagrams, and the memory of experienced staff. The model does not eliminate the need to read code or understand infrastructure; it provides a behavioral map for deciding where deeper inspection matters.

2.3 The role of BSN and the BSN Workbench

Behavior Simulation Notation makes participants, interactions, state, decision logic, expectations, and outcomes explicit without requiring the behavior to be hidden inside production code. The BSN Workbench supports model authoring, scenario execution, trace inspection, comparison, revision, versioning, and lifecycle linkage. Its purpose is not to make every engineer a specialist modeler. Its purpose is to make consequential behavior executable and reviewable by the people who must understand and own it.

Tooling principle

The Workbench is valuable when it helps engineers change an important decision, discover a consequential problem earlier, or preserve evidence across the lifecycle. Model count is not the outcome.

3. How SDSE Improves Daily Engineering Work

3.1 Clearer work before coding begins

Engineers often spend the first part of implementation discovering that words such as “approved,” “available,” “complete,” “recovered,” or “secure” mean different things to different stakeholders. SDSE turns those meanings into observable scenarios. The engineer can expose contradictory states, missing transitions, time limits, invalid sequences, exception ownership, and recovery obligations before converting assumptions into code.

9. Ambiguous requirements become specific behavioral questions.
10. Implicit business rules become visible guards, states, and expected outcomes.
11. Interfaces are reviewed as interactions over time, not only message schemas.
12. Acceptance becomes a review of executable evidence rather than an interpretation contest at the end.

3.2 Fewer late surprises and less avoidable rework

An engineer who detects a broken assumption in a model can revise a scenario, rule, or state transition. The same discovery after implementation may require coordinated changes across services, tests, databases, security controls, release plans, and operating procedures. SDSE does not promise that every defect will be prevented. It concentrates early attention on the behavioral defects whose late correction would be most disruptive.

3.3 Better technical decisions

Simulation gives engineers a disciplined way to compare alternatives before implementation effort or organizational commitment biases the choice. A distributed team can compare retry strategies under partial failure. A security team can examine whether automated containment blocks recovery. An AI team can test authority and escalation policies. A release team can evaluate canary, blue-green, and rolling behavior under interruption. The discussion changes from “Which design do we prefer?” to “Which design produces the strongest evidence under the scenarios that matter?”

3.4 Stronger collaboration and professional voice

The executable model becomes a boundary object: technical enough to be precise, yet visible enough for domain, product, security, reliability, and business stakeholders to challenge. Engineers gain influence because they can connect a technical interaction to a customer promise and demonstrate why

an assumption, control, or release decision matters. This is not influence through jargon; it is influence through shared evidence.

Daily-work improvement

SDSE replaces part of the engineer's invisible interpretive labor with explicit, collaborative, and reusable behavioral engineering.

4. How SDSE Improves Specific Engineering Roles

SDSE is a shared discipline, but it changes each engineering role differently. The following sections describe how existing responsibilities are strengthened rather than displaced.

4.1 Software developer

Conventional emphasis. Translate assigned requirements and designs into working code, correct defects, and maintain implementation quality.

SDSE-enhanced role. Participate in shaping behavioral intent, challenge assumptions before implementation, and use the validated baseline to guide code and AI-assisted development.

- 13. Review Candidate Specifications and identify behavior that cannot yet be implemented unambiguously.
- 14. Contribute implementation constraints and counterexamples to model scenarios.
- 15. Trace code, configuration, and generated artifacts to behavioral obligations.
- 16. Compare observed implementation behavior with EXPECT outcomes and resolve divergence explicitly.

Role improvement. The developer performs less speculative coding, receives clearer acceptance intent, and gains responsibility for end-to-end outcomes rather than only assigned components.

Boundary to preserve

The model should define the promise and important constraints without prescribing unnecessary implementation detail.

4.2 Software architect

Conventional emphasis. Define structures, components, interfaces, technologies, deployment patterns, and system qualities.

SDSE-enhanced role. Evaluate architecture through executable interactions, failure behavior, timing, control, and recovery before committing the organization to a design.

- 17. Model coordination across services, agents, people, data, and external dependencies.
- 18. Compare alternative responsibility allocations and interaction patterns.
- 19. Test whether locally correct components produce acceptable system behavior together.
- 20. Carry architecture decisions and constraints into scenarios, tests, deployment, and telemetry.

Role improvement. Architecture decisions become supported by scenario evidence, explicit assumptions, and observable tradeoffs rather than diagrams and expert judgment alone.

4.3 Requirements and systems engineer

Conventional emphasis. Document stakeholder needs, functions, constraints, interfaces, and acceptance criteria.

SDSE-enhanced role. Transform consequential intent into executable scenarios and use simulation to determine whether requirements are behaviorally complete and consistent.

21. Identify participants, states, triggers, obligations, exceptions, and recovery responsibilities.
22. Expose conflicts between functional, safety, security, policy, and operational expectations.
23. Maintain traceability from promise to scenarios, decisions, controls, tests, and observed outcomes.
24. Preserve unresolved questions and model limits instead of allowing assumptions to disappear into implementation.

Role improvement. Requirements become challengeable evidence rather than static statements, improving stakeholder convergence and lifecycle traceability.

4.4 Quality and test engineer

Conventional emphasis. Validate implemented software against documented expectations and identify defects before release.

SDSE-enhanced role. Help determine whether the intended behavior is correct before implementation, then derive focused implementation evidence from the validated model.

25. Design boundary, failure, misuse, degraded, adversarial, and recovery scenarios during specification and design.
26. Convert model scenarios and EXPECT outcomes into acceptance, integration, property, and fault-injection tests.
27. Identify when code and model disagree and require an explicit decision about which is wrong.
28. Use production observations to improve scenario coverage instead of only adding regression tests after defects.

Role improvement. Testing gains earlier influence, stronger behavioral intent, and coverage organized by consequence and uncertainty rather than raw test quantity.

Boundary to preserve

A green pipeline proves that configured checks passed. It does not prove that the behavioral promise or the selected checks are complete.

4.5 Security engineer

Conventional emphasis. Identify vulnerabilities, define controls, perform threat analysis, verify secure development practices, and govern risk.

SDSE-enhanced role. Model adversarial interactions and the behavior of preventive, detective, responsive, and recovery controls across the lifecycle.

- 29. Simulate misuse, privilege escalation, policy conflict, compromised identity, delayed detection, containment, and recovery.
- 30. Evaluate whether individually reasonable controls create harmful sequences when combined.
- 31. Connect threat scenarios to implementation tests, monitoring, response policy, and residual-risk approval.
- 32. Use SDSE alongside secure lifecycle practices such as the NIST Secure Software Development Framework [5].

Role improvement. Security becomes more predictive and behavior-aware, while automated response receives explicit authority, blast-radius, escalation, and safe-stop boundaries.

4.6 DevOps and release engineer

Conventional emphasis. Automate integration, delivery, deployment, environment, configuration, and release processes.

SDSE-enhanced role. Treat deployment as consequential behavior with state, version skew, migration, evidence gates, rollback, irreversibility, and human intervention.

- 33. Simulate rollout and rollback across compatible and incompatible versions.
- 34. Define telemetry, hold points, stop conditions, and accountable approval for consequential releases.
- 35. Connect validated behavioral scenarios to targeted pipeline gates and environment tests.
- 36. Compare deployment strategies against the same risk and recovery scenarios.

Role improvement. Pipelines provide evidence about system behavior and recovery, not merely evidence that configured workflow steps completed.

4.7 Site reliability engineer

Conventional emphasis. Protect reliability, establish service objectives, improve operability, respond to incidents, and reduce recurrence.

SDSE-enhanced role. Make failure propagation, detection, automated and human response, degraded operation, reconciliation, and recovery executable before and after incidents.

- 37. Simulate misleading telemetry, partial dependency failure, concurrent response, repeated remediation, and recovery failure.
- 38. Derive observability requirements from scenario decisions and evidence needs.
- 39. Use game days to compare model expectations with controlled system observations.
- 40. Update models, tests, controls, and operating guidance when incidents reveal model-reality differences.

Role improvement. Reliability engineering becomes more proactive, runbooks become behaviorally grounded, and incident learning updates a living engineering baseline.

4.8 Platform engineer

Conventional emphasis. Provide reusable infrastructure, services, templates, controls, developer experience, and golden paths.

SDSE-enhanced role. Provide reusable behavioral golden paths and evidence for service onboarding, identity, deployment, observability, scaling, failure, and recovery.

- 41. Model platform-product interactions and the consequences of shared defaults.
- 42. Simulate platform changes against representative product scenarios before broad rollout.
- 43. Publish reusable behavior models, scenario libraries, and exception patterns.
- 44. Enable product teams to extend domain behavior without copying or bypassing platform controls.

Role improvement. The platform supplies reusable assurance as well as reusable technology, increasing cross-team leverage and reducing unsafe variation.

4.9 AI and agent engineer

Conventional emphasis. Develop prompts, models, memory, policies, tools, evaluation, and agent workflows.

SDSE-enhanced role. Engineer an executable control envelope around autonomy, including permitted actions, evidence thresholds, approval, escalation, override, blast radius, and safe stop.

- 45. Simulate ambiguous, stale, conflicting, malicious, or incomplete evidence.

46. Evaluate repeated actions, tool failure, policy conflict, human interruption, and non-deterministic response.

47. Compare agent and human decisions against the same consequential scenarios.

48. Maintain traces that support the Govern, Map, Measure, and Manage functions of the NIST AI Risk Management Framework [4].

Role improvement. The engineer can expand useful autonomy incrementally while maintaining accountable human control and stable behavioral expectations across model or tool changes.

Boundary to preserve

Simulation supports a decision about autonomy; it does not transfer accountability to the model or agent.

5. SDSE as a Professional Force Multiplier

5.1 Better systems thinking

Component expertise remains valuable, but system outcomes are increasingly produced by interaction. SDSE trains engineers to ask what state the system is in, what happened previously, who has authority, which evidence is current, what happens concurrently, how failure propagates, and how the system returns to an acceptable state. These questions improve reasoning even when a formal model is not required.

5.2 Stronger decision-making

A trusted engineer does not merely recommend a design. The engineer can state the behavioral promise, describe the alternatives, show the scenarios used to challenge them, explain the evidence and model limitations, identify residual risk, and name the observation that would require reconsideration. SDSE makes that decision discipline repeatable.

5.3 Greater business influence

Engineers gain business influence when they connect technical behavior to customer outcomes, operational exposure, cost, compliance, safety, and trust. The executable scenario helps a business leader see why a retry rule, identity boundary, deployment hold point, agent authority level, or recovery path affects the promise. Technical advice becomes easier to evaluate and fund.

5.4 More effective use of AI

AI can help draft Candidate Specifications, propose scenarios, identify missing paths, generate model elements, produce tests, implement code, and analyze traces. The engineer remains responsible for deciding whether those artifacts reflect the domain and deserve acceptance. A validated behavioral baseline gives AI-assisted work a stronger target and gives reviewers a more stable basis for rejecting plausible but incorrect output.

Professional multiplier

SDSE converts individual engineering judgment into explicit, reviewable, reusable, and improvable organizational evidence.

6. Improving Engineering Collaboration

Software behavior crosses organizational boundaries. Product defines outcomes, domain experts know exceptions, architects allocate responsibility, developers implement, testers challenge, security constrains, operations observe, and business leaders accept consequences. Each function sees only part of the system. SDSE creates a shared behavioral reference without requiring everyone to read the same code or surrender specialized expertise.

Participants	Contribution	Collaboration improvement
Product and domain	Clarify the promise, important rules, exceptions, and value.	Scenarios reveal whether the behavior matches stakeholder intent.
Architecture and development	Define responsibility, interaction, state, and implementation constraints.	Traces expose system effects before code hardens the choice.
Test and security	Challenge normal assumptions and design failure, misuse, and adversarial paths.	EXPECT outcomes become assurance targets.
DevOps and SRE	Define release evidence, observability, response, recovery, and learning.	Operational reality updates the same behavioral baseline.
Business and risk	Evaluate consequence, evidence, limitations, and residual risk.	Approval becomes tied to an explicit promise and decision record.

Table 2. The model creates a shared conversation while preserving specialist responsibilities.

6.1 From interpretation debates to evidence reviews

Without executable behavior, cross-functional review can become a debate about what prose, diagrams, or interface definitions imply. With SDSE, the team can ask what happened in the trace, which state permitted the action, which expectation was violated, which assumption produced the result, and what must change before acceptance. Disagreement does not disappear, but it becomes more specific and productive.

6.2 Preserving decision history

Versioned models and scenarios preserve why a decision was made. New engineers can reproduce the evidence rather than rely on folklore. When operations contradict an assumption, the team can identify which baseline and dependent artifacts are affected. This reduces the human cost of turnover and helps experienced engineers transfer judgment rather than merely transfer documents.

7. Improving Quality and Productivity

SDSE improves productivity by improving the quality and timing of engineering decisions. It should not be justified through a generic claim that modeling makes every engineer a fixed percentage faster. Its practical value comes from preventing selected late corrections, accelerating convergence, focusing expensive assurance, reusing behavioral evidence, and reducing avoidable incident work.

7.1 Productivity without reducing rigor

- 49. Earlier clarification reduces speculative implementation and repeated reinterpretation.
- 50. Reusable scenarios reduce duplicated acceptance and test design across teams and releases.
- 51. Simulation narrows the scenarios that require expensive integrated or production-like environments.
- 52. Traceability reduces evidence reconstruction during release, audit, incident, and change review.
- 53. Operational observations become targeted model, test, and control improvements instead of isolated action items.

7.2 Protecting engineers from low-value work

A significant part of engineering frustration comes from avoidable uncertainty: contradictory requirements, unstable acceptance, hidden ownership, repeated emergency remediation, and decisions that must be reconstructed after the people involved have moved on. SDSE does not remove difficult work. It moves more of that difficulty into intentional engineering, where teams can explore it collaboratively and learn without production consequences.

7.3 Measure outcomes, not modeling activity

The number of diagrams, models, scenarios, or simulation runs does not prove value. The relevant measures are whether important ambiguity was resolved earlier, a design decision improved, a missing test or control was identified, late rework declined, an unsafe release or agent action was prevented, recovery became safer, or operational learning reached the baseline faster.

Productivity rule

A model that does not improve a decision, reduce consequential uncertainty, strengthen evidence, or preserve useful learning may have little engineering value.

8. Human Accountability in AI-Native Engineering

AI has no skin in the game. It does not experience customer harm, service loss, regulatory exposure, professional consequence, or damaged trust. It can recommend, generate, summarize, and act within granted authority, but it cannot assume moral or organizational accountability. NIST's AI Risk Management Framework emphasizes trustworthy characteristics and lifecycle risk-management functions; those functions still require accountable people and institutions [4].

8.1 The Human-in-Control principle

SDSE strengthens Human-in-Control by making the boundaries of delegation executable. The model can distinguish advisory, assistive, semi-autonomous, and controlled-autonomous behavior; define required evidence; specify approval and escalation; limit scope and blast radius; preserve override; and establish safe-stop conditions. Engineers can test whether a proposed authority model behaves acceptably under uncertainty before granting that authority in production.

Human responsibility	Why it cannot be delegated away
Define the promise	People decide which outcomes matter, to whom, and under which constraints.
Approve model scope and fidelity	People decide what the model represents and which limitations remain material.
Interpret evidence	People decide whether traces and outcomes support acceptance or require revision.
Accept residual risk	People and authorized organizations own the consequences that simulation cannot eliminate.
Authorize autonomy and release	People decide what an agent or system may do and under what operating evidence.
Learn from reality	People require changes when observed behavior differs materially from the baseline.

Table 3. SDSE makes accountability more explicit; it does not transfer accountability to a tool or model.

8.2 The engineer as an accountable evidence producer

The future engineer will increasingly be asked not merely whether software was built, but why leaders should trust it within a defined scope. A responsible answer includes the promise, scenarios, evidence, limitations, residual risk, operating signals, and accountable approvals. SDSE gives engineers a disciplined way to assemble that answer without pretending that simulation proves everything.

9. New and Improved Engineering Competencies

SDSE adds competencies to existing software engineering rather than replacing the body of knowledge. Some are technical, some analytical, and some collaborative. Together they improve an engineer's ability to handle consequence and complexity.

Competency	Observable capability
Behavioral modeling	Express participants, interactions, states, decisions, expectations, timing, failure, and recovery.
Scenario engineering	Design representative success, boundary, failure, misuse, adversarial, degraded, and recovery situations.
Simulation analysis	Interpret traces, counterexamples, invariant violations, and unexpected outcomes.
Systems thinking	Reason about emergent behavior across components, people, policies, agents, and environments.
Evidence-based decisions	Compare alternatives, state assumptions, expose uncertainty, and justify acceptance or revision.
Fidelity management	Invest detail in proportion to consequence and identify where the model is not a reliable decision proxy.
Traceability	Connect behavioral obligations to requirements, architecture, code, tests, controls, release, and telemetry.
Human-control engineering	Design authority, approval, escalation, override, blast radius, and safe stop for automation.
Cross-functional facilitation	Help different stakeholders expose assumptions and converge on observable behavior.
Operational learning	Use telemetry and incidents to revise the model, scenarios, controls, and baseline.

Table 4. SDSE competencies strengthen technical judgment and lifecycle responsibility.

9.1 A progression model

Level	Demonstrated practice
Aware	Reads a behavior model, follows a trace, and recognizes where SDSE applies.
Practitioner	Contributes scenarios, expectations, and evidence for a bounded use case.
Facilitator	Leads cross-functional modeling and simulation workshops and manages disagreement.
Owner	Maintains a living behavioral baseline, lifecycle traceability, and change triggers.
Leader	Builds organizational capability, proportional governance, reusable libraries, and measurement.

Table 5. SDSE can be developed as a practical competency progression rather than a tool certification alone.

10. The Emerging Behavioral Software Engineer

The Behavioral Software Engineer is a useful name for the person who ensures that consequential software behavior is understood, simulated, challenged, validated, implemented, observed, and improved. The title need not become a separate position in every organization. It may describe a specialization, a responsibility held by a senior engineer or architect, a team capability, or a role supported by an SDSE enablement function.

10.1 Core responsibilities

1. **Facilitate behavioral discovery.** Help stakeholders identify the promise, participants, states, interactions, assumptions, and consequences.
2. **Create and maintain executable models.** Use BSN and the Workbench to represent the scope required for the decision.
3. **Design and execute scenarios.** Cover the normal and challenge conditions proportionate to consequence and uncertainty.
4. **Analyze and communicate evidence.** Explain traces, counterexamples, limitations, alternatives, and residual risk.
5. **Connect lifecycle artifacts.** Relate the baseline to architecture, code, tests, controls, deployment, telemetry, and incidents.
6. **Own behavioral change control.** Identify triggers, coordinate resimulation, and preserve approved versions.
7. **Support accountable decisions.** Ensure that the people authorized to accept behavior and risk receive understandable evidence.

10.2 Relationship to existing roles

The Behavioral Software Engineer does not become the sole owner of system behavior. Product owners still own outcomes, domain experts still own critical rules, architects still own structural decisions, developers still own implementation quality, test and security engineers still challenge evidence, SRE still owns reliability practice, and authorized leaders still accept residual risk. The behavioral engineer connects and strengthens those responsibilities.

Role design principle

Create a separate title only when the scope and organizational need justify it. Every engineering team can still adopt behavioral responsibilities without adding a new job family.

How SDSE Elevates the Software Engineer

HOW SDSE ELEVATES THE SOFTWARE ENGINEER

Engineers move from implementing assumed behavior to producing evidence that behavior deserves implementation.

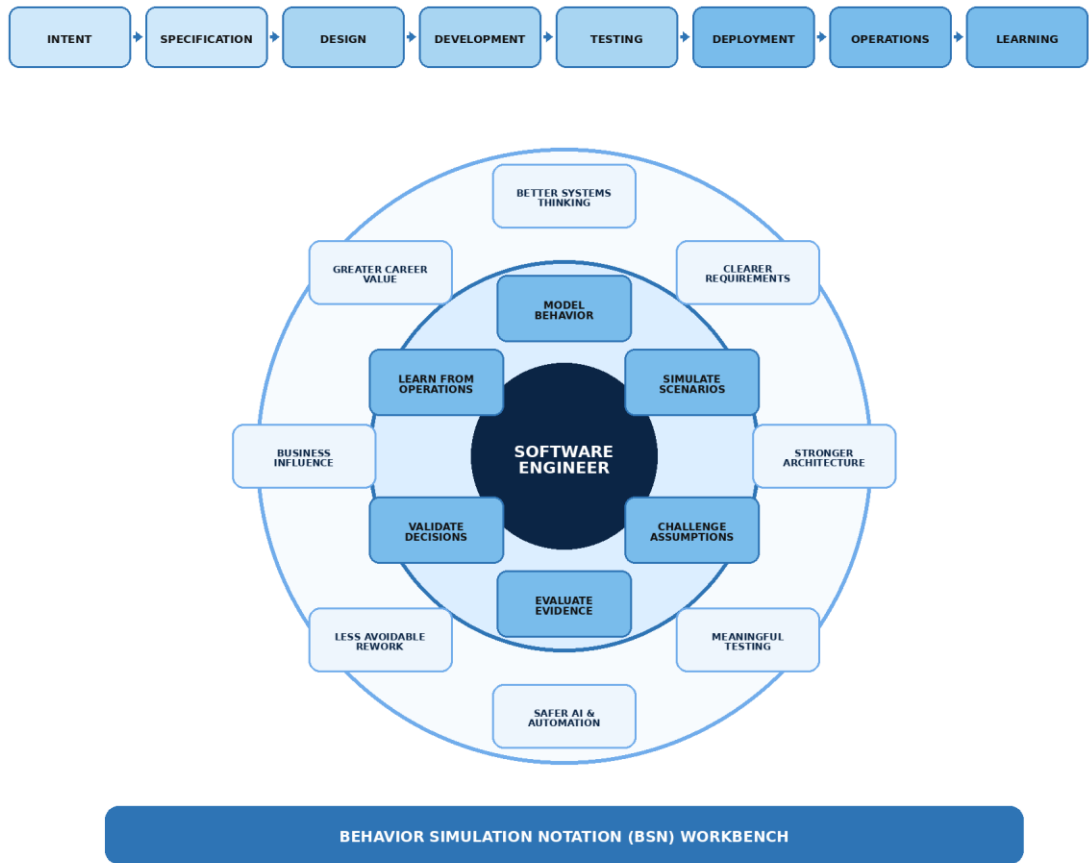


Figure 1. SDSE connects engineer capabilities to professional outcomes across the software lifecycle.

11. Career Benefits for Software Engineers

11.1 Greater resistance to commoditization

As implementation becomes more automated, engineers who compete only on the speed of producing routine code face pressure. Engineers who can understand ambiguous domains, reason across complex systems, frame meaningful scenarios, evaluate evidence, make tradeoffs, and own consequences remain difficult to commoditize. SDSE develops precisely those capabilities while keeping them grounded in executable technical work.

11.2 Expanded career paths

Behavioral competence supports advancement into senior engineering, architecture, systems engineering, security, reliability, platform leadership, AI assurance, technical product leadership, engineering management, Behavior Model Ownership, and dedicated Behavioral Software Engineering. The common progression is from implementing local tasks toward owning consequential system decisions.

11.3 Stronger professional credibility

Credibility grows when an engineer can explain not only what was built, but what the system is expected to do, which scenarios were examined, what evidence supports the decision, where the model is limited, which risk remains, and how the team will learn. This is especially valuable when executives, customers, regulators, or partners must understand a technical recommendation without reviewing implementation detail.

Career stage	SDSE contribution	Professional benefit
Junior engineer	Learn to read scenarios and traces; connect tasks to behavioral promises.	Faster domain learning and better questions
Mid-level engineer	Author scenarios, challenge assumptions, and trace implementation to EXPECT outcomes.	Greater ownership and cross-functional contribution
Senior engineer	Lead behavioral decisions across components and disciplines.	Broader system accountability and technical influence
Staff/principal engineer	Define reusable patterns, fidelity policy, governance, and lifecycle integration.	Organizational leverage and strategic authority
Engineering leader	Build capability, ownership, measurement, and a culture of evidence.	Safer acceleration and stronger investment decisions

Table 6. SDSE responsibilities can grow with engineering scope and authority.

12. How Engineering Leaders Should Introduce SDSE

SDSE adoption should improve engineering work, not create a parallel bureaucracy. Leaders should begin with one consequential, bounded behavior and a real decision. The first pilot must be large enough to matter, small enough to simulate, and measurable enough to teach the organization whether the method deserves broader use.

1. **Select a consequential use case.** Choose behavior with meaningful consequence, ambiguity, interaction complexity, autonomy, or expensive late discovery.
2. **Create a cross-functional team.** Include the sponsor, domain expertise, engineering, architecture, test or assurance, operations, and a Behavior Model Owner.
3. **Protect learning time.** Provide BSN and Workbench training, facilitation, and time to revise early models without judging success by modeling speed.
4. **Define the decision and evidence gate.** Agree what the simulation can cause the organization to accept, revise, reject, restrict, or observe.
5. **Integrate with existing work.** Connect the baseline to requirements, architecture, development, testing, pipelines, controls, telemetry, and incident learning.
6. **Measure and decide.** Compare the baseline with observed outcomes and explicitly scale, revise, collect more evidence, or stop.

12.1 Create psychological and professional safety

Simulation is valuable because it exposes what the team has not yet understood. Leaders must not punish engineers for finding contradictions, unsafe interactions, or weak assumptions in a Candidate Specification or design. Those discoveries are the work. A culture that treats early counterexamples as embarrassment will push ambiguity back into code and production.

12.2 Reward decision quality

Performance systems should recognize engineers who prevent unnecessary implementation, clarify cross-team behavior, improve recovery, constrain unsafe automation, preserve traceability, and turn incidents into reusable learning. If only code volume and delivery speed are rewarded, SDSE will be perceived as extra work even when it protects the organization from greater loss.

13. Measuring Improvement to Engineering Roles

Measurement should show whether engineering work and outcomes improved. It should combine leading indicators, such as earlier discovery and decision latency, with downstream evidence such as rework, change failure, incident effort, and learning-loop speed. Employee experience measures can be useful when connected to observable changes in unclear work, emergency interruption, or cross-functional decision quality.

Dimension	Possible measure	Evidence
Requirements clarity	Material ambiguities or contradictions resolved before development	Decision and issue records
Behavioral decision latency	Time from a consequential question to evidence-backed acceptance, revision, rejection, or escalation	Review and decision timestamps
Early discovery	Unsafe interactions, missing controls, and recovery defects found before code or release	Scenario and trace findings
Avoided late rework	Reduction in behavior-related requirement, design, integration, release, or incident correction	Change records and time sampling
Assurance continuity	Critical promises linked to scenarios, tests, controls, release evidence, and telemetry	Traceability record
Operational outcomes	Behavior-related change failures, customer impact, unsafe automation, and recovery effort	Delivery, security, and incident systems
Learning-loop speed	Time from a material observation to model revision, resimulation, and approved baseline	Model and release history
Engineer experience	Reduction in unclear assignments, preventable emergencies, and decision reconstruction	Survey plus workflow evidence
Role growth	Engineers demonstrating scenario, simulation, systems-thinking, and evidence competencies	Skill assessment and work products

Table 7. Measures should connect role improvement to observable engineering and business outcomes.

13.1 Measurement safeguards

7. Define measures and the baseline before the pilot begins.
8. Separate modeled counterexamples from production incidents that were certainly prevented.
9. Record other process, architecture, staffing, traffic, and tooling changes that could explain outcomes.
10. Report unfavorable evidence, including modeling effort, delay, or maintenance cost.
11. Measure engineers as contributors to system outcomes, not by a simplistic quota of models or scenarios.

14. Risks, Misconceptions, and Safeguards

Misconception	Practical safeguard
"SDSE adds more work."	It adds targeted early effort to reduce larger late effort. Scope the model to the decision and measure the net effect.
"Engineers already model systems mentally."	Mental models are valuable but cannot be executed, reviewed, versioned, traced, or shared reliably.
"Testing already validates behavior."	Testing evaluates an implementation against selected expectations. SDSE challenges the intended behavior before implementation and strengthens those expectations.
"The model will restrict creativity."	The baseline defines the promise and important constraints while preserving implementation freedom.
"We must model everything."	Fidelity and coverage should be proportional to consequence, uncertainty, and decision value.
"AI can create and approve the model."	AI can assist, but people must validate intent, evidence, model limits, authority, and residual risk.
"The Behavioral Software Engineer owns all behavior."	Ownership remains distributed. The role connects evidence and responsibilities rather than absorbing them.
"A successful simulation proves production safety."	Simulation supports a bounded decision. It must be combined with implementation tests, operational evidence, monitoring, review, and accountable judgment.

Table 8. SDSE remains useful only when it is proportional, accountable, and connected to reality.

14.1 Model risk

A model can be incomplete, incorrect, stale, or misleadingly precise. Teams must identify the scope and fidelity required for the decision, document what is not represented, validate important assumptions, compare model and implementation behavior, and revise the baseline when operations reveal divergence. The Behavior Model Owner is accountable for maintaining the artifact, but reviewers remain accountable for their decisions.

14.2 Adoption risk

SDSE can fail if it becomes a specialist ceremony performed after decisions are made, a documentation requirement with no decision authority, or a tool rollout measured by usage. Adoption must begin with a consequential question, cross-functional participation, an evidence gate, a named owner, and a decision that can genuinely change.

Conclusion

SDSE improves the role of the software engineer by moving important engineering work to the point where it can have the greatest influence. Engineers help establish intent, make consequential behavior executable, challenge assumptions, evaluate alternatives, produce evidence, guide implementation, constrain autonomy, and learn from operations. The result is not less coding. It is coding performed within a stronger engineering system.

This evolution is especially important in AI-native software engineering. Code generation will continue to accelerate, but AI does not own the promise or its consequences. Accountable engineers must decide what deserves to be built, what evidence is sufficient, what authority may be delegated, what risk remains, and what reality requires the organization to change. A living behavior model gives those decisions continuity across the lifecycle.

Organizations should introduce SDSE proportionally: one bounded promise, one consequential decision, one cross-functional team, and one measurable pilot. Engineers should be rewarded for earlier discovery, clearer collaboration, stronger assurance, safer operation, and preserved learning. BSN and the BSN Workbench provide a practical foundation, but the professional value comes from the judgment and responsibility of the people who use them.

Closing statement

SDSE does not move software engineers away from engineering. It enables them to perform the most important parts of engineering earlier, more visibly, and with better evidence.

Appendix A. Role Development Worksheet

Use this worksheet with an engineer and manager to identify how SDSE can improve a real role. Select one consequential work area rather than assessing every responsibility at once.

Field	Working answer
Current role	What role, level, team, and system scope are being considered?
Behavioral promise	Which customer, business, safety, security, or operational promise does the engineer influence?
Current entry point	When does the engineer first participate in defining or challenging that behavior?
Recurring ambiguity	Which requirements, states, interactions, ownership, or acceptance questions repeatedly appear?
Late discovery	Which behavior-related problems are usually found during implementation, release, or operation?
Decision to improve	What can simulation evidence cause the team to accept, revise, reject, restrict, or observe?
Model scope	Which participants, states, sequences, controls, failures, and recovery paths matter?
Engineer contribution	Will the engineer model, design scenarios, analyze traces, link implementation, or own the baseline?
Competency goal	Which SDSE competency should the engineer demonstrate or develop?
Support	What training, coaching, domain access, tool access, and protected time are required?
Evidence of improvement	Which decision, rework, assurance, operational, learning, or experience measure should change?
Ownership	Who owns the promise, model, review, approval, residual risk, and ongoing learning?
Review date	When will the engineer and leader review evidence and decide the next step?

Table 9. Practical worksheet for developing an engineer's SDSE responsibilities.

Appendix B. Role Catalogue at a Glance

Role	Conventional emphasis	SDSE enhancement	Primary benefit
Developer	Implements assigned requirements	Challenges intent and implements against a validated baseline	Less speculative coding; greater outcome ownership
Architect	Defines structures and interfaces	Compares architecture through behavior, failure, and recovery	Evidence-supported decisions
Requirements/systems	Documents needs and constraints	Makes consequential requirements executable	Earlier convergence and stronger traceability
Quality/test	Tests implemented software	Challenges behavior before code and derives focused evidence	Earlier influence and meaningful coverage
Security	Finds vulnerabilities and verifies controls	Simulates adversarial and control behavior	Predictive security and safer automation

Role	Conventional emphasis	SDSE enhancement	Primary benefit
DevOps/release	Automates delivery and deployment	Models rollout, evidence gates, rollback, and recovery	Behavior-aware release decisions
SRE	Responds and improves reliability	Simulates failure and feeds operations back into the model	Proactive resilience and faster learning
Platform	Provides reusable technology and golden paths	Provides reusable behavioral evidence and scenarios	Cross-team assurance leverage
AI/agent	Builds models, prompts, tools, and workflows	Engineers authority, escalation, override, and safe stop	Controlled autonomy and human accountability

Table 10. Summary of how SDSE strengthens nine software engineering roles.

References

Sources were accessed on August 15, 2026. References support contextual claims about the profession, AI, secure development, and lifecycle economics. The SDSE role framework is presented by the author.

- [1] IEEE Computer Society. “Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 4.” [Official SWEBOK resource](#)
- [2] U.S. Bureau of Labor Statistics. “Software Developers, Quality Assurance Analysts, and Testers.” Occupational Outlook Handbook, 2025. [Official occupational outlook](#)
- [3] Google Cloud. “Announcing the 2025 DORA Report.” [Official report announcement](#)
- [4] National Institute of Standards and Technology. “Artificial Intelligence Risk Management Framework (AI RMF 1.0).” [Official NIST resource](#)
- [5] National Institute of Standards and Technology. “SP 800-218, Secure Software Development Framework (SSDF) Version 1.1.” [Official publication](#)
- [6] National Aeronautics and Space Administration. “Systems Engineering Handbook 2.5: Cost Effectiveness Considerations.” [Official NASA guidance](#)
- [7] Hornbeek, Marc, with Debashis Bhattacharyya and Jitin Dhanani. Simulation-Driven Software Engineering: A Behavior-First Discipline for Modern Systems. [Purchase on Amazon](#)

About the Author

Marc Hornbeek, a.k.a. DevOps_the_Gray, is an engineer, author, educator, and the originator of Simulation-Driven Software Engineering. He is the principal author of Simulation-Driven Software Engineering: A Behavior-First Discipline for Modern Systems and the creator of Behavior Simulation Notation and the BSN Workbench. His experience spans software, networks, automation, DevOps, security, reliability, platforms, AI, simulation, and engineering tools. Through PrimaModus®, he helps teams apply SDSE to consequential software decisions.

Contact: mhornbeek@engineeringdevops.com