

From Spec-Driven to Simulation-Driven

Why Executable Behavior Must Become the Next Engineering Baseline

The central proposition

Specifications clarify intent. Simulation tests whether that intent can behave correctly before the production system exists.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

August 2026

Executive Summary

Software development is returning to specifications. AI coding agents can generate code at machine speed, but they still need explicit intent, context, constraints, and acceptance conditions. GitHub Spec Kit and Kiro are prominent examples of a broader move toward specification-driven development: define what should be built before asking humans or agents to build it [2][3][4]. This is an important correction to prompt-first and code-first development.

But a better specification does not, by itself, prove that the proposed system will behave correctly. A specification may be detailed yet still contain contradictory rules, unsafe state transitions, timing assumptions, unmodeled failure paths, or locally correct components whose combined behavior is wrong. The specification tells us what we believe should happen. It does not necessarily show what will happen when those behaviors interact.

This white paper proposes Simulation-Driven Software Engineering (SDSE) as the next step. SDSE makes an executable behavior model the central engineering artifact. Teams create a Candidate Specification, express critical behavior in an executable notation, run scenarios before implementation, examine traces and outcomes, refine assumptions, and approve a versioned Validated Behavioral Baseline. That living model then continues through design, code, testing, deployment, operations, and learning.

Core idea

Do not wait for code to discover behavior. Execute the behavior of the system before the production system exists.

1. Spec-driven development improves the clarity, traceability, and structure of intent.
2. Simulation-driven engineering adds evidence by executing interacting behaviors before implementation.
3. The living behavior model becomes an executable memory and a common promise thread across the lifecycle.
4. Behavior Simulation Notation (BSN) and the BSN Workbench provide a practical way to author, simulate, trace, review, and maintain that model.

Contents

- 1. The Return of the Specification**
- 2. Why Specification Alone Is Not Enough**
- 3. The Simulation-Driven Step**
- 4. Continuity, Not Rejection**
- 5. The SDSE Lifecycle and the Living Behavior Model**
- 6. Behavior Simulation Notation and the BSN Workbench**
- 7. Evidence, Trust, and Human-in-Control Governance**
- 8. Relationship to Established Practices**
- 9. A Practical Adoption Roadmap**
- 10. Implications for Engineering Leaders**

Conclusion

References

About the Author

1. The Return of the Specification

Software engineering has spent years optimizing the speed of implementation. Agile reduced batch size. DevOps improved flow and feedback. Continuous delivery automated the path to production. AI-assisted development now accelerates code generation itself. These advances are valuable, but they have also exposed a fundamental constraint: faster construction amplifies the cost of misunderstood intent.

Specification-driven development is a rational response. GitHub describes Spec Kit as an intent-driven process that keeps the definition of what to build at the center of an AI-enabled workflow [2]. Kiro similarly structures work around requirements, system design, and implementation tasks before code is written [3][4]. The objective is not paperwork. It is to give people and agents durable context, expose decisions, and reduce the ambiguity of unstructured prompting.

This shift matters because AI-generated code is highly sensitive to context. A vague prompt can produce convincing output while silently embedding assumptions the team never approved. Structured specifications improve the quality of the conversation between stakeholders, engineers, and coding agents. They also create reviewable artifacts that can be versioned, traced, and governed.

What specification-driven development improves

5. Intent becomes explicit instead of remaining distributed across meetings, prompts, tickets, and individual memory.
6. Requirements, design decisions, tasks, and acceptance conditions become easier to inspect and revise.
7. AI agents receive stronger context and more stable boundaries for planning and implementation.
8. Teams can reason about a proposed change before generated code creates the illusion of progress.

The return of the specification is therefore a positive development. The problem is not that specification-driven development goes too far. The problem is that most implementations do not go far enough.

2. Why Specification Alone Is Not Enough

A specification is a representation of intended behavior. Its value depends on what it expresses, what it leaves implicit, and how it is validated. Natural-language requirements, structured templates, user stories, interface contracts, diagrams, and formal models each reduce some forms of ambiguity. None

automatically proves that the combined system will behave correctly under realistic sequences, timing, concurrency, failures, misuse, or changing operating conditions.

The critical gap is the difference between describing behavior and executing behavior. A review can confirm that a statement is understandable. Traceability can confirm that each requirement maps to a design element or test. Static analysis can identify inconsistencies in structure. Yet important defects often emerge only when multiple rules, states, actors, interfaces, and events interact over time.

Four recurring failure modes

9. Locally correct, systemically wrong. Each component satisfies its own specification, but their interactions create an incorrect outcome.
10. Happy-path completeness. The specification describes expected flow but underrepresents timeouts, retries, partial failure, recovery, misuse, and degraded operation.
11. Translation loss. Meaning changes as intent becomes requirements, design, code, tests, policies, alerts, and operational procedures.
12. False confidence from green indicators. Code compiles, tests pass, and the pipeline is green, while the delivered behavior still violates the stakeholder promise.

These are not arguments against specifications. They are arguments for a stronger validation mechanism. If the system behavior matters, teams should be able to challenge that behavior before expensive implementation choices harden around an untested interpretation.

3. The Simulation-Driven Step

Simulation-Driven Software Engineering is a behavior-first discipline in which a team creates and executes a behavioral representation of the proposed system before committing to production implementation. The purpose is not to predict every implementation detail. It is to make the most important behavioral promises explicit, executable, observable, and reviewable at the point where change is still inexpensive.

The SDSE sequence begins with intent. The team develops a Candidate Specification and expresses the critical behavior as an executable model. Scenarios challenge normal flow, boundary conditions, failures, recovery, security misuse, and operational conditions. Simulation produces traces and observable outcomes. Stakeholders refine the model until they approve a versioned Validated Behavioral Baseline. Design and implementation then proceed from evidence rather than assumption.

This changes the engineering question. Specification-driven development asks, “Have we described what to build?” SDSE adds, “Have we demonstrated that the proposed behavior is coherent, safe, and acceptable?”

FROM SPEC-DRIVEN TO SIMULATION-DRIVEN

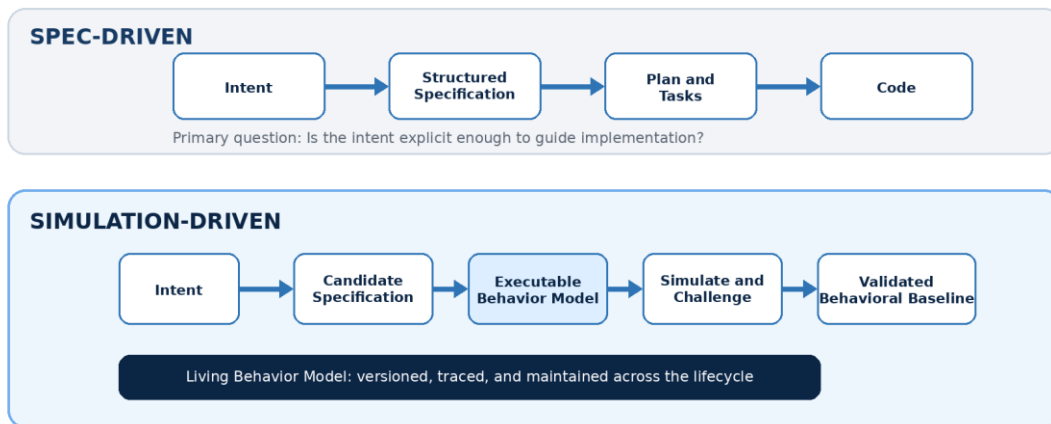


Figure 1. Specification clarifies intent; simulation adds behavioral evidence before implementation.

4. Continuity, Not Rejection

SDSE does not reject specification-driven development. It strengthens it. The Candidate Specification is the starting point for behavioral modeling, not a discarded artifact. Simulation reveals ambiguities and contradictions that improve the specification. The improved specification, model, scenarios, traces, and approval evidence form a stronger engineering baseline together.

Dimension	Specification-Driven Development	Simulation-Driven Software Engineering
Primary question	What should be built?	Will the proposed behavior work under meaningful conditions?
Central artifact	Structured specification	Executable living behavior model plus its specification
Validation timing	Reviews and checks before or during implementation	Behavioral simulation before implementation and throughout the lifecycle
Evidence	Clarity, completeness, traceability, tasks, acceptance criteria	Simulation outcomes, deterministic traces, scenario coverage, approved behavioral baseline
Relationship to code	Specification guides generated or human-written code	Validated behavior guides code and remains a reference after deployment
Typical residual risk	Important behavior remains inferred until code or integration exists	Model fidelity and scenario coverage may be insufficient and must be governed

Table 1. SDSE extends the value of specification-driven development by executing the proposed behavior.

The distinction is architectural, not rhetorical. If a “simulation” is only a disposable prototype, a single happy-path animation, or a test harness created after implementation, the approach has not established the living behavioral baseline that SDSE requires.

5. The SDSE Lifecycle and the Living Behavior Model

SDSE encapsulates the complete software lifecycle. Simulation is not a gate inserted between specification and design. The behavior model begins early, establishes the validated baseline, and then evolves as architecture, implementation, evidence, and operating conditions change.

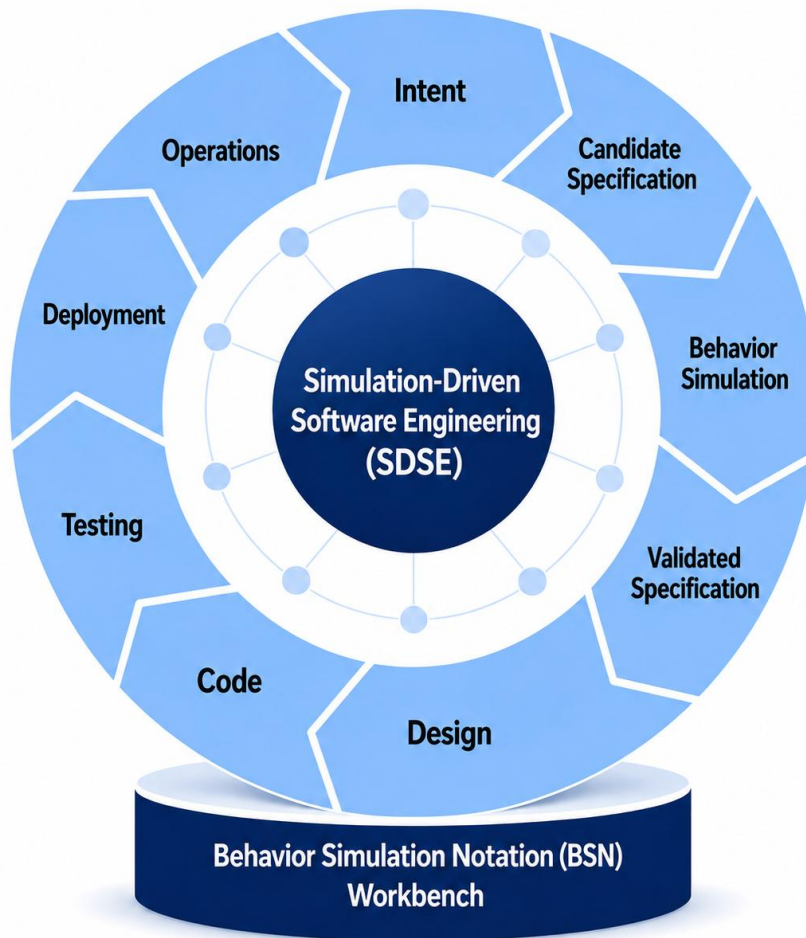


Figure 2. SDSE spans the lifecycle; the living behavior model is maintained rather than discarded.

Intent. Define the stakeholder outcomes, promises, constraints, risks, and operating context.

Candidate Specification. Describe proposed behavior clearly enough to model, question, and revise.

Behavior Simulation. Execute scenarios, observe state and interaction, and challenge normal and adverse conditions.

Validated Specification. Approve a versioned behavioral baseline supported by traces and stakeholder review.

Design and Code. Translate validated behavior into architecture and implementation while preserving traceability.

Testing. Derive and align tests with scenarios, expected outcomes, boundaries, and failure behavior.

Deployment and Operations. Compare runtime evidence with the behavioral promise; learn, update, re-simulate, and reapprove material changes.

Living behavior model

The model is the system's executable memory: a durable, versioned record of what stakeholders agreed the system should do and the evidence used to justify that agreement.

6. Behavior Simulation Notation and the BSN Workbench

A discipline becomes practical when teams have a usable way to express and execute it. Behavior Simulation Notation (BSN) is designed as a readable, executable notation for distributed and interactive systems. It makes system boundaries, interfaces, operating conditions, behavior rules, network relationships, scenarios, and expected outcomes explicit.

BSN construct	Engineering purpose
BOX	Defines a behavioral participant, service, subsystem, agent, device, or other modeled entity.
INTERFACE	Defines what a BOX can receive, expose, signal, or exchange.
OPERATING	Represents operating modes, environmental conditions, constraints, and relevant state.
LOGIC	Defines behavior rules, decisions, transitions, and responses.
NETWORK	Connects participants and makes interaction topology explicit.
SCENARIO	Defines sequences and conditions used to exercise the model.
EXPECT	States the outcomes or invariants that must hold for a scenario to be accepted.

Table 2. Core BSN constructs used to express and validate behavior.

The BSN Workbench provides the working environment for authoring, executing, reviewing, and maintaining BSN models. Each simulation can generate a deterministic trace showing events, rule evaluation, state changes, outputs, signals, conditions, errors, and EXPECT results. Metrics summarize trends; traces explain why the behavior occurred.

This distinction matters. A dashboard may show that a result changed. A trace connects the result to the sequence of decisions and interactions that produced it. That evidence supports engineering review, defect analysis, governance, regression, and trust.

The Workbench as a collaboration surface

13. Product and domain experts validate that modeled outcomes reflect intent and real-world rules.
14. Architects and developers assess interfaces, states, dependencies, and implementation implications.
15. Test, security, reliability, and operations specialists contribute adverse scenarios and measurable expectations.
16. Leaders and risk owners approve behavioral baselines and material changes based on evidence.

7. Evidence, Trust, and Human-in-Control Governance

AI can help generate candidate scenarios, propose transitions, identify missing paths, correlate traces, and estimate the impact of a change. Those capabilities can increase the breadth and speed of exploration. They do not transfer accountability to the model or the agent.

Human-in-Control governance means that accountable people retain authority over domain meaning, acceptable risk, scenario sufficiency, and approval of the behavioral baseline. Material changes should be versioned, affected simulations and regressions should be rerun, risk should be reviewed, approvals should be recorded, and trace links should be updated.

Green is not proof

A successful pipeline proves that its configured checks passed. It does not automatically prove that the system fulfills the stakeholder promise. SDSE adds behavioral evidence to the definition of “green.”

Minimum governance controls

17. Version the specification, model, scenarios, expectations, traces, and behavioral baseline together.
18. Define who may propose, review, approve, and supersede behavioral baselines.
19. Classify scenarios by business criticality, safety, security, reliability, and operational impact.
20. Record model fidelity and known exclusions so simulation confidence is not overstated.
21. Connect production evidence and incidents back to the model and regression scenarios.

8. Relationship to Established Practices

SDSE should be positioned as a complementary discipline. It does not replace requirements engineering, architecture, Behavior-Driven Development, Test-Driven Development, Model-Based Systems Engineering, digital twins, DevOps, DevSecOps, SRE, or platform engineering. It provides a common behavioral thread that those practices can use.

Practice	Primary contribution	Relationship to SDSE
BDD	Collaborative discovery and executable examples that are automated against implementation [5].	BSN can model interacting system behavior before implementation and provide scenarios that later inform BDD and tests.
TDD	Uses tests to guide implementation design at code or component level.	SDSE validates system behavior before production code exists; TDD helps implement that behavior correctly.
Prototyping	Explores feasibility, usability, or design through a partial implementation.	SDSE focuses on an explicit, executable, versioned behavior model rather than a disposable approximation.
MBSE	Applies modeling across requirements, design, analysis, verification, and validation throughout the lifecycle [6].	SDSE shares the model-based commitment but focuses specifically on executable software behavior and its continuous evidence.
Digital twins	Use synchronized models to represent, diagnose, predict, or optimize physical systems and operations [8][9].	An SDSE living behavior model can inform a software behavioral twin, but it begins before the operational system exists.
DevOps / DevSecOps / SRE	Improve flow, security, reliability, feedback, and operation of delivered systems.	SDSE supplies validated behavioral promises and scenarios that can guide pipelines, controls, observability, and operations.

Table 3. SDSE complements established practices rather than replacing them.

INCOSE defines MBSE as the formalized application of modeling across requirements, design, analysis, verification, and validation, beginning in conceptual design and continuing through later lifecycle phases [6]. SDSE is compatible with that direction. Its differentiator is the deliberate elevation of executable behavior and scenario evidence as the baseline for software engineering decisions.

9. A Practical Adoption Roadmap

Organizations should begin with a bounded, consequential behavior rather than attempt to model an entire enterprise. The goal of the first implementation is to prove that simulation improves decisions, not to maximize model size.

1. Select a meaningful pilot. Choose a feature, service interaction, workflow, control function, or agent behavior with important ambiguity or cross-component risk.

2. Clarify intent. Identify stakeholders, outcomes, constraints, failure concerns, security obligations, reliability expectations, and operating conditions.

3. Create the Candidate Specification. Document the proposed behavior in language that domain and engineering stakeholders can challenge.

4. Model critical behavior in BSN. Represent the relevant BOXes, INTERFACES, OPERATING conditions, LOGIC, NETWORK relationships, SCENARIOS, and EXPECT outcomes.

5. Simulate normal and adverse scenarios. Exercise success, boundary, timing, concurrency, failure, misuse, recovery, and degraded-operation paths.

6. Review traces and refine. Resolve contradictions, expose assumptions, change the specification or model, and rerun affected scenarios.

7. Approve the baseline. Version the validated specification, model, scenarios, traces, risks, and approvals.

8. Connect the lifecycle. Link architecture, code, tests, security controls, observability, incidents, and production evidence to the behavioral promise.

Measures that indicate useful adoption

Measure	Observable evidence
Earlier discovery	Behavioral conflicts and missing scenarios found before design or code.
Reduced rework	Fewer late requirement, integration, and acceptance changes attributable to misunderstood behavior.
Traceable decisions	Greater coverage of critical promises by scenarios, EXPECTs, traces, and approvals.
Faster shared understanding	Less time required for domain, engineering, security, test, and operations stakeholders to agree on behavior.
Operational learning	Incidents and runtime deviations converted into model changes and regression scenarios.

10. Implications for Engineering Leaders

Adopting SDSE is not primarily a tooling decision. It changes when the organization expects evidence, which artifact carries shared meaning, and who is accountable for behavioral approval. Leaders should treat the living behavior model as an engineering asset, not optional documentation.

22. Fund behavioral modeling and simulation as part of product engineering, not as a late testing activity.

- 23. Create cross-functional ownership involving product, domain, architecture, development, test, security, reliability, and operations.
- 24. Require explicit fidelity statements and risk-based scenario coverage instead of claiming that a model is “complete.”
- 25. Use AI to expand exploration while preserving human authority over meaning, risk, and approval.
- 26. Measure avoided rework, improved decisions, and behavioral evidence—not the number of diagrams or models produced.

The deeper implication is that delivery speed is no longer enough. AI can increase the velocity of production faster than most organizations can increase the velocity of understanding. SDSE is an approach for restoring that balance by making understanding executable.

Conclusion

Specification-driven development is an important advance because it restores intent to the center of software creation. In an AI-native environment, specifications give humans and agents a stronger basis for planning, implementation, review, and governance. But a specification remains a proposition about behavior.

Simulation-Driven Software Engineering adds the missing engineering step: execute the proposed behavior before the production system exists. Build a Candidate Specification. Express the behavior in an executable model. Challenge it with scenarios. Examine the traces. Refine the assumptions. Approve a Validated Behavioral Baseline. Then carry that living model through design, code, testing, deployment, operations, and learning.

The next generation of software teams will not merely write better specifications. They will run them. BSN provides a behavior-first approach to software engineering and enables simulation-driven validation. The BSN Workbench helps engineers design with confidence, simulate with precision, and implement with clarity.

Final proposition

Specifications tell us what we intend to build. Simulation provides evidence that the intended behavior deserves to be built.

References

- [1] Marc Hornbeek, "Spec-Driven Development is Rising. But Most People Don't Understand Why," LinkedIn, 2026. [Original LinkedIn post](#)
- [2] GitHub, "Spec Kit Documentation," accessed August 14, 2026. [GitHub Spec Kit documentation](#)
- [3] Kiro, "Kiro and the Future of AI Spec-Driven Software Development," July 15, 2025. [Kiro article](#)
- [4] Kiro, "What Are Specs?" accessed August 14, 2026. [Kiro Specs documentation](#)
- [5] Cucumber, "Behaviour-Driven Development," accessed August 14, 2026. [Cucumber BDD documentation](#)
- [6] INCOSE, "MBSE Initiative Working Group," accessed August 14, 2026. [INCOSE MBSE Initiative](#)
- [7] INCOSE, "Model-Based Practices - Systems Engineering Vision 2035," accessed August 14, 2026. [INCOSE Vision 2035](#)
- [8] National Institute of Standards and Technology, "Digital Twins," accessed August 14, 2026. [NIST Digital Twins](#)
- [9] National Institute of Standards and Technology, "Digital Twins for Advanced Manufacturing," April 15, 2024. [NIST Advanced Manufacturing](#)
- [10] 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, software development, behavioral simulation, testing, automation, DevOps, DevSecOps, SRE, platform engineering, and intelligent software delivery. 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