

The Deterministic Control Plane for Traditional and Modern Software Engineering

Building trust through executable behavior models and stakeholder promises

Anchor Traditional Software Engineering and Modern Software Engineering to agreed behavior as non deterministic AI expands across applications, SDLC processes, and infrastructure.

By Marc Hornbeek, a.k.a. DevOps_the_Gray

Originator of Simulation-Driven Software Engineering

October 2026

www.PrimaModus.com

Executive Summary

Traditional Software Engineering and Modern Software Engineering share the same obligation: preserve stakeholder promises from intent through production. An agreed executable behavior model should anchor a trusted deterministic control plane for that work. Specifications, tests, processes, and artifacts provide evidence whose meaning must remain connected to the promised behavior. Trust depends on that connection and on controls that act when the evidence is insufficient or behavior departs from the agreement.

Modern Software Engineering makes this obligation more urgent because non deterministic AI can develop and operate applications, create and operate SDLC processes, and create and operate supporting infrastructure. This pyramid of interdependent systems increases the ways in which behavior can vary and promises can be lost. The deterministic control plane anchors those variations to explicit expectations, authority, and evidence. Its rules must distinguish mechanically checkable obligations from outcomes requiring statistical evaluation or accountable human judgment.

This paper follows an illustrative payment service to show why Traditional Software Engineering already needed this control plane and why Modern Software Engineering increases its importance. It explains how executable behavior models support trust across the lifecycle and how to govern non deterministic actions without assuming identical AI outputs. SDSE and BSN provide one implementation of these principles. The case and outcomes are hypothetical teaching examples.

Contents

1 Preserve promises in Traditional Software Engineering	3
2 Control non deterministic Modern Software Engineering	4
3 Build trust through executable behavioral promises	5
4 Keep the control plane authoritative across the lifecycle	6
5 Apply deterministic control to non deterministic actions	7
6 Protect trust in the control plane	7
7 Use SDSE and BSN across both engineering approaches	8
8 Lead adoption in Traditional and Modern Software Engineering	9
Conclusion	9
References and About the Author	10

List of Figures

Figure 1 Deterministic control of interdependent non deterministic systems	4
Figure 2 A trusted control plane for Traditional and Modern Software Engineering	6

1 Preserve promises in Traditional Software Engineering

A payment team releases a change after the expected reviews and automated checks pass. During a network interruption, the payment provider accepts a charge, but its response never reaches the application. The application retries while infrastructure recovery restarts a worker. A customer receives two charges for one purchase, although the release records show that the required engineering activities were completed.

Begin Traditional Software Engineering with the customer promise: one payment request must create at most one charge, including during timeout, retry, and restart. Finance needs a reconcilable transaction record, and operations must distinguish failure from an unknown outcome. Security requires the authorization boundary to survive recovery. These behaviors define what stakeholders must be able to trust, regardless of how the implementation is produced.

These expectations explain what the system must do under relevant conditions. Give each promise a stable identifier and identify the stakeholders authorized to agree its meaning. Record the assumptions that make the promise achievable, including the payment provider's duplicate-request handling. If those assumptions remain uncertain, expose the uncertainty before approving a design or release.

Traditional Software Engineering often governs delivery through a relationship chain from specification to test, process, and artifact. Each translation can preserve local correctness while obscuring the original behavioral promise. A developer may interpret timeout as failure while an operator interprets it as an unknown result. Specifications and tests can capture behavior accurately, but their existence and correlation alone do not establish the behavior stakeholders need to trust.

Make the deterministic control plane explicit in Traditional Software Engineering. A successful build establishes a completed activity, and a passing test supports the assertions exercised under its conditions. Link those results directly to named promises and identify what remains unverified. Without that discipline, trust can migrate toward completed artifacts while unacceptable behavior remains hidden until deployment.

An agreed executable behavior model provides the common reference for this control plane. It describes participants, states, interactions, and expected outcomes that stakeholders can challenge before implementation. The deterministic rules retain the agreed meaning as designs and artifacts evolve. Modern Software Engineering inherits this need and adds non deterministic actors to the activities being governed.

2 Control non deterministic Modern Software Engineering

Modern Software Engineering can place non deterministic AI at three interdependent levels: the application, the SDLC processes that change it, and the infrastructure supporting both. AI models may generate and operate applications, create and operate delivery workflows, and create and operate infrastructure. Their outputs and action choices can vary with context and between runs. Map this entire dependency pyramid when deciding which outcomes stakeholders can trust.

The payment promise reaches all three levels of Modern Software Engineering. An application agent may choose a retry, a delivery agent may change its implementation or tests, and an infrastructure agent may restart the worker. Each action affects the same financial outcome. The deterministic control plane must preserve identity, require relevant evidence, and constrain recovery so that local decisions respect the agreed behavior.

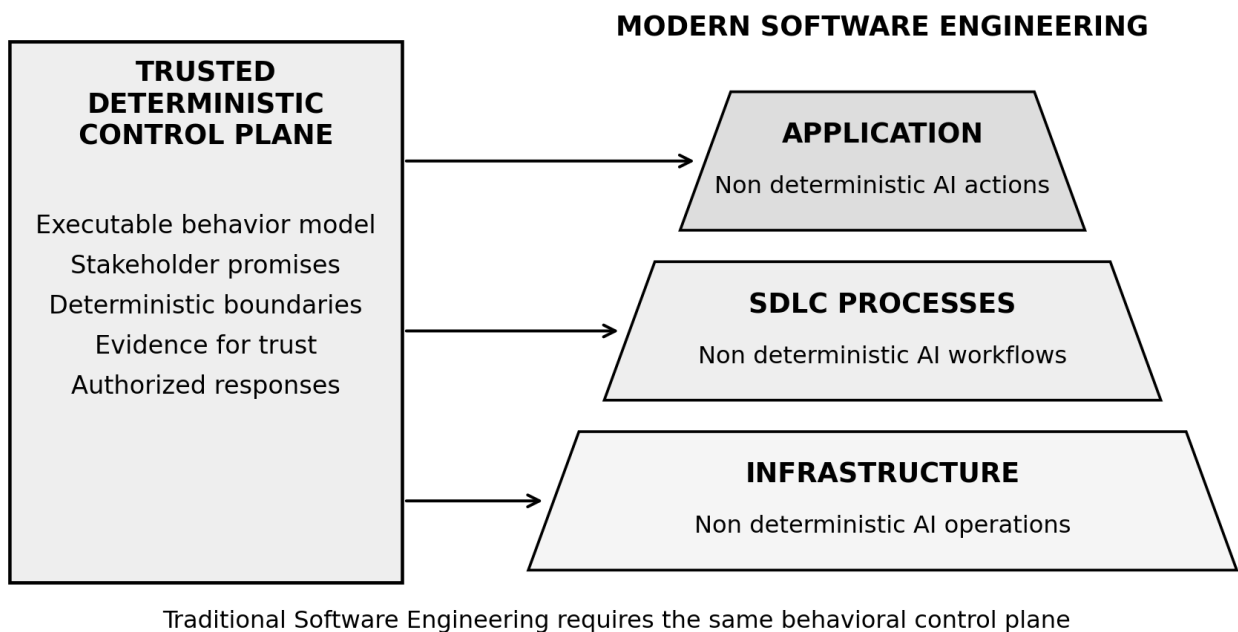


Figure 1 Deterministic control of interdependent non deterministic systems

Assign behavioral obligations at each level and connect them to the trusted control plane. Specify the state that recovery must preserve, the evidence required before retry changes progress, and the authority for each action. Enforce those deterministic boundaries where tools and systems act. Without these connections, non deterministic choices that appear acceptable locally can combine into an unacceptable customer outcome.

Traditional Software Engineering can produce the same failure through human-written logic and conventional automation. Distributed timing and external dependencies already introduce uncertainty even when local code is deterministic. Modern Software Engineering adds non deterministic AI at multiple levels and increases the pace of change. A trusted deterministic control plane is therefore a continuing engineering requirement whose importance has grown.

3 Build trust through executable behavioral promises

Build trust by translating the payment promise into conditions stakeholders can inspect and engineers can evaluate. In Traditional Software Engineering, those conditions guide human decisions and conventional automation. In Modern Software Engineering, they also constrain non deterministic AI actions. Define durable payment identity, conflicting-request handling, record retention, and state transitions that preserve the promise during uncertainty.

For promise P-PAY-01, the modeled rule is that all attempts belonging to one payment identity refer to the same payment obligation. At most one accepted charge may be associated with that identity. A delayed or missing response leaves the outcome unresolved until the authoritative provider record establishes it. Repeated requests must return or reconcile the existing outcome within the stated operating assumptions.

Make the conditions visible. The provider must support the required duplicate-request semantics, and the application must retain the identity long enough to cover the retry and reconciliation windows. Validate those dependency contracts through integration evidence. A model that assumes support the provider does not supply will produce reassuring simulations of an infeasible design.

Use executable scenarios to examine normal completion, timeout after acceptance, concurrent retries, restart during reconciliation, and delayed provider responses. Include a conflicting amount under an existing payment identity and a recovery action that loses local state. Observe both the transaction result and the state changes across participants. If the scenarios only check returned messages, they can miss an unauthorized financial side effect.

Bring customer representatives, finance, development, operations, security, and platform ownership into the review where their responsibilities are affected. Resolve disagreements about unknown results, customer communication, and recovery authority before accepting the baseline. Designate who owns model integrity and who approves material changes. Agreement gives the model organizational authority; execution helps reveal consequences of that agreement.

Keep the model small enough to understand and precise enough to challenge. Record exclusions and operating limits alongside the promise, including dependencies and scenarios still unverified. Version the accepted baseline and link it to the next engineering commitment. An incomplete model can guide a bounded decision when its limits are visible, while hidden omissions can turn a green result into misplaced confidence.

4 Keep the control plane authoritative across the lifecycle

Use the accepted behavior model as the authoritative reference for the control plane at each consequential commitment. Traditional Software Engineering requires architecture, design, implementation, and integration evidence connected to the same promises. Modern Software Engineering requires the same connections when non deterministic AI performs or modifies those activities. Transaction identity, reconciliation responsibility, and failure handling must retain their meaning throughout.

Build and packaging processes should preserve links among the change, control plane baseline, and behavioral evidence. Release decisions must identify affected promises and establish whether the evidence supports trust in the proposed change. Deployment controls preserve state compatibility and recovery expectations. In Modern Software Engineering, a non deterministic recommendation to release or recover remains subject to these deterministic obligations.

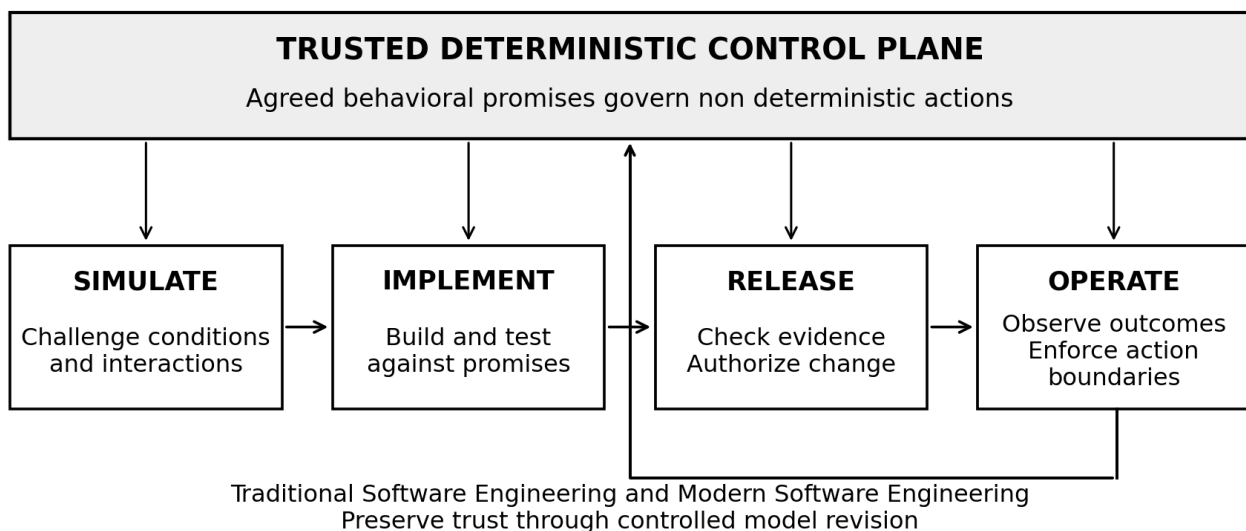


Figure 2 A trusted control plane for Traditional and Modern Software Engineering

Compare outcomes using defined equivalence rules. The model and implementation may have different internal events or timing details while satisfying the same payment obligation. Check charge counts, payment identities, allowed state transitions, reconciliation results, and prohibited side effects. Requiring identical internal traces can reject valid designs without improving confidence in the customer promise.

Connect monitoring to those same observations. Reconcile application events with the provider's authoritative transaction records so that instrumentation errors do not hide duplicates. Define who responds when the result is unknown, the evidence is incomplete, or a violation is detected. Findings return through review to a corrected implementation, improved controls, or an authorized change to the behavioral baseline.

5 Apply deterministic control to non deterministic actions

A deterministic control plane supplies explicit rules and decision mechanisms anchored in agreed behavior. In Traditional Software Engineering, it governs human work and conventional automation; in Modern Software Engineering, it also governs non deterministic AI actions. The executable behavior model defines acceptable outcomes, while connected verification and enforcement make that agreement operational. Trust requires those connections at the actual action boundaries.

Deterministic control means that mechanically checkable obligations produce consistent decisions for equivalent evidence. Non deterministic AI may suggest different recovery plans, but each plan must satisfy the same authorization and transaction-integrity rules. Enforce critical boundaries through tool permissions, interfaces, and protected state transitions. This anchors trust in observable compliance with the promise while allowing variation in how an authorized outcome is achieved.

Some expectations require statistical evaluation or domain judgment. Assess variable AI performance using representative scenarios, repeated trials, and approved thresholds; preserve the evaluation method and relevant system versions. NIST emphasizes intended-use evidence, realistic evaluation conditions, and ongoing assessment of deployed behavior [1]. Record uncertainty when available evidence cannot support the requested decision.

Use deterministic checks for charge identity and prohibited duplicate effects, while evaluating an AI support assistant's explanation quality through an approved rubric and review process. A model-based evaluator can assist, with its variability and calibration recorded. Anthropic describes code-based, model-based, and human graders with different strengths and limitations [2]. A reproducible evaluation policy makes decisions accountable without making every underlying judgment deterministic.

6 Protect trust in the control plane

Protect trust in the control plane by separating proposed changes from authority to approve behavioral obligations. Traditional Software Engineering needs this separation when teams modify their own acceptance criteria. Modern Software Engineering makes it especially important when non deterministic AI can generate code, tests, processes, and candidate model revisions. Protect critical checks and evidence so the actor being assessed cannot silently redefine success.

Ask an independent reviewer to challenge failure sequences and assumptions. Inspect generated tests against the promise and verify that they can observe the actual financial effect. Retain immutable links to the approved model version, implementation, environment, and evaluation result. Shared assumptions can survive across model, code, and tests, so agreement among generated artifacts needs independent evidence.

7 Use SDSE and BSN across both engineering approaches

Simulation-Driven Software Engineering (SDSE) provides one lifecycle discipline for establishing this trusted control plane in Traditional Software Engineering and Modern Software Engineering. It develops intent into a candidate specification and challenges expected behavior through simulation before further commitments. Findings inform the validated specification, design, implementation, and integration. Release and operational feedback maintain the connection to stakeholder promises as systems and evidence change [3, 4].

Behavior Simulation Notation (BSN) supplies an explicit representation of the behavior being explored. Participants, interfaces, operating conditions, rules, interactions, and scenarios give engineers a way to examine consequences before committing to an implementation. The BSN Workbench supports this modeling and simulation activity. Stakeholder review and controlled baselines establish which modeled promises are authoritative for the next decision.

For the payment example, represent the customer request, payment service, provider, transaction store, and recovery controller. Exercise acceptance followed by response loss, overlapping retries, and worker restart while reconciliation is incomplete. Use the simulation findings to refine the ownership of identity and the handling of unknown outcomes. The resulting baseline becomes a reference for acceptance expectations, developer tests, and integration evidence.

Use BSN to examine the interactions of non deterministic agents across the Modern Software Engineering pyramid. Model pipeline-change authority, protected checks, recovery permissions, and delegation limits. Examine conflicts between application, delivery, and infrastructure agents acting on shared state. The deterministic control plane keeps their permitted actions conditional on the stakeholder promises they collectively support.

Increase model fidelity when a decision requires more realistic timing, dependencies, or operating conditions. Record which assumptions have been checked against prototypes, real interfaces, or production evidence. Finite simulations cover the behaviors and scenarios represented by the model; they do not establish universal correctness. An observed counterexample is evidence to investigate, with the affected commitment reconsidered.

Adopt SDSE and BSN alongside effective development practices. TDD can guide implementation and refactoring, while ATDD can establish shared acceptance examples. Policy enforcement and runtime monitoring carry approved expectations into actual delivery and operation. The practical contribution is continuity of behavioral meaning across these activities, with each producing evidence appropriate to its scope [4].

8 Lead adoption in Traditional and Modern Software Engineering

Choose one consequential promise and an upcoming engineering decision for the control plane pilot. A Traditional Software Engineering team can start with duplicate prevention during retry and recovery. A Modern Software Engineering team should include the non deterministic AI actions that can affect those outcomes. Name the stakeholders, model owner, assumptions, and evidence required to establish trust before the change progresses.

Give the team time to challenge the model before rewarding faster artifact production. Ask finance to examine reconciliation, operators to examine interrupted recovery, and security to examine authority across tool routes. Implement the resulting rules through the existing delivery system and protect the acceptance baseline. A pilot limited to drawing a model will leave the production risk unchanged.

Measure whether the approach improves decisions. Track specification gaps discovered before implementation, consequential interaction defects found before release, and the effort needed to maintain model and evidence links. Observe production violations and unresolved outcomes against a defined prior baseline. These measures should support learning without implying that every improvement was caused by modeling.

Expand where the pilot justifies the cost. Reuse common definitions and evidence contracts while allowing teams to choose suitable implementation tools. Review the model when provider behavior, infrastructure, AI components, or stakeholder obligations change materially. A shared behavioral reference remains useful only while its meaning, assumptions, and evidence reflect the system being governed.

Conclusion

The illustrative payment team now handles response loss as an unknown outcome requiring reconciliation. Retries preserve the payment identity, recovery retains the necessary state, and release assessment examines evidence for the same agreed promise. Operators compare application observations with provider records and can interrupt unsafe recovery. Customer communication reflects the actual transaction state.

Traditional Software Engineering always needed a trusted deterministic control plane anchored in stakeholder behavior promises. Modern Software Engineering intensifies that need by introducing non deterministic AI into applications, SDLC processes, and infrastructure. Agreed executable behavior models preserve the expected outcomes across that pyramid, with independent evidence and enforcement at action boundaries. Trust grows when engineering decisions remain accountable to the behaviors people depend on.

References

[1] National Institute of Standards and Technology. Artificial Intelligence Risk Management Framework 1.0, 2023. Section 3, AI Risks and Trustworthiness. <https://airc.nist.gov/airmf-resources/airmf/3-sec-characteristics/>

[2] Anthropic. Demystifying evals for AI agents. January 9, 2026. <https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents>

[3] Marc Hornbeek, Debashis Bhattacharyya, and Jitin Dhanani. Simulation-Driven Software Engineering. 2026.

[4] Marc Hornbeek. Connecting Simulation and Test-Driven Development: Comparing SDSE and BSN with TDD and ATDD. PrimaModus white paper, October 2026. Author-supplied reference.

Learn more and participate

Visit www.PrimaModus.com for information about SDSE, Behavior Simulation Notation, and the BSN Workbench. Senior engineers, architects, security practitioners, SREs, platform engineers, and agentic AI teams interested in evaluating these practices can contact mhornbeek@engineeringdevops.com. Trial participants can contribute realistic scenarios and structured feedback to help shape the notation and workbench.

About the Author

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

Prepared with assistance from OpenAI ChatGPT in drafting, source research, and diagram development. Final editorial and technical review rests with the author.