WHITEPAPER — 2025

Software Signal Engineering

A Probabilistic Framework for Adaptive Validation
in Distributed Systems

Massimo Forno

Technical Founder & CEO — Quantik Mind GmbH

Version 1.0 · 2025 · Switzerland

quantikmind.com

Validated on a 120-microservice banking architecture · 2,175 test scenarios

ABSTRACT

This paper introduces Software Signal Engineering — a discipline for adaptive validation in distributed software systems. It presents theoretical foundations drawing from Bayesian probability theory, information theory, adaptive machine learning, and quantum-inspired computational principles, and describes the architecture of Quantik Mind as its first industrial implementation. Empirical validation was conducted on a production-grade financial system comprising 120+ microservices and 2,175 functional test scenarios, using two years of historical execution data. Results demonstrated 77.1% reduction in redundant test execution, 95.5% coverage of dynamically modeled high-impact risk, and 4.4x faster detection of critical failures against a deterministic full regression baseline. The paper argues that deterministic validation discipline is structurally mismatched to probabilistic distributed systems, and that signal-driven adaptive selection represents the necessary evolution of software quality practice.

Keywords: software testing, probabilistic risk modeling, adaptive test selection, distributed systems, quantum-inspired algorithms, Bayesian inference, adaptive machine learning, observability, CI/CD optimization, signal engineering

Table of Contents

1. Introduction and Problem Statement

1.1 The Deterministic Model

For most of software engineering history, quality assurance operated under a deterministic model: define a test suite, execute it uniformly, measure coverage, repeat. This model emerged in an era of monolithic, tightly coupled systems where behavior was largely predictable and risk was relatively static. Within those constraints, the model was rational. Full regression provided a repeatable baseline. Coverage metrics offered a proxy for confidence. The cost of exhaustive execution was acceptable given the simplicity of underlying architectures.

1.2 The Architectural Shift

Modern distributed systems have invalidated those constraints. Microservice architectures introduce hundreds of independent services with dynamic, non-linear interdependencies. A change in one service propagates risk through dependency chains that no static test suite was designed to model. Runtime state, deployment patterns, and traffic distribution continuously reshape where failure is most likely to emerge.

In this environment, risk is not a static property of the codebase. It is a dynamic field — redistributed with every commit, every deployment, every anomaly in runtime telemetry. The fundamental assumption of deterministic validation — that executing the same suite produces equivalent information on each run — is no longer valid.

1.3 The Three Failure Modes

Applying a deterministic discipline to a probabilistic system produces three compounding failure modes that accumulate silently over time:

> **Execution waste.** Between 60% and 80% of tests in a full regression suite carry no meaningful probability of revealing new information at any given execution. They validate what was already known. The compute, time, and engineering attention consumed is structurally irrecoverable.
>
> **Feedback latency.** Long pipelines delay the detection of real failures. In high-velocity environments, feedback latency compounds — each delayed detection increases the surface area of dependent commits that may inherit the defect.
>
> **False confidence.** A green dashboard reporting 100% coverage communicates certainty that the system does not possess. Risk has not been eliminated. It has been redistributed into areas the static suite was not designed to observe.

This paper presents Software Signal Engineering as the structural response to these failure modes — not an optimization of deterministic practice, but a replacement of the model under which it operates.

2. Theoretical Foundations

Software Signal Engineering draws from four distinct theoretical domains. Each contributes a layer of the discipline and maps to a concrete computational operation in the selection engine.

2.1 Bayesian Probability Theory

The core of Software Signal Engineering is a Bayesian model of risk. Each test in the validation suite carries a prior probability P(t) of revealing a failure — derived from historical execution results, failure distributions, and service-level volatility scores.

This prior is continuously updated as new evidence arrives: code changes, runtime signals, deployment events, and previous execution outcomes. The posterior probability — the updated belief about where risk is concentrated after observing new evidence — drives selection.

The discipline does not ask *'which tests should we run?'* It asks: *What is the probability that risk remains unvalidated in this area of the system, given everything we currently observe?*

2.2 Information Theory

Selection decisions are governed by expected information gain. A test is worth executing if its execution is likely to reduce uncertainty about the system's current state. Tests that are historically stable — the 'always green' population — carry near-zero information value under most conditions and are suppressed accordingly. Tests at high-risk boundaries, or covering recently modified services, carry high information value and are prioritized.

The selection threshold parameter τ controls the coverage-efficiency tradeoff: higher τ values yield more conservative selection; lower values pursue more aggressive reduction. The default $\tau = 0.70$ reflects the empirically validated balance between waste elimination and risk coverage.

2.3 Adaptive Machine Learning

The selection engine continuously refines its probabilistic models through a supervised learning model trained on historical execution outcomes. As execution cycles accumulate, the model refines feature weights, volatility scores, and entanglement graph edge strengths based on observed outcomes.

This adaptive layer ensures that the system does not converge on a fixed selection pattern. It learns which signals are genuinely predictive of failure in a specific architecture, and progressively reduces dependence on generic priors in favor of architecture-specific knowledge.

2.4 Quantum Mechanics as Conceptual Framework

Quantum mechanical principles provide a vocabulary for reasoning about the probabilistic, interdependent nature of distributed systems. This is not a literal application of quantum physics to software — it is a computational analogy that maps naturally onto the mathematical structures required for adaptive validation. The three principles — superposition, entanglement, and uncertainty — each correspond to a distinct computational phase in the selection engine, described in Section 4.

3. The Probability Distribution Model

3.1 Initial Distribution Construction

The initial risk probability distribution is constructed from historical execution data. For each test t in the validation suite, an initial risk score P(t) is computed as a weighted combination of the following factors:

Historical failure rate	The proportion of past executions in which the test detected a failure. This is the primary prior signal.
Failure recency	Recent failures are weighted more heavily using an exponential decay function with configurable half-life. A failure from two weeks ago contributes more than one from six months prior.
Business criticality	Tests covering high-criticality services receive a prior boost proportional to the declared business impact of that service.
Flakiness penalty	Tests with high outcome variance receive a confidence penalty that reduces their effective information value.
Cold-start fallback	When no historical data exists, uninformative priors are applied, ensuring equal initial consideration until evidence accumulates.

3.2 Continuous Recalibration

The distribution is recalibrated at every trigger event — a code commit delivered via webhook, a deployment, a runtime anomaly, or a scheduled execution cycle. Each recalibration incorporates all available signals to produce an updated posterior reflecting the current state of the system, not its historical average.

Commits are ingested dynamically through a webhook integration, enabling the engine to update the risk distribution in real time as code changes land — without requiring manual configuration or pipeline restarts.

The recalibration process is computationally lightweight. Selection computation on a 2,000+ test suite completes in under 20 seconds, making it suitable for inline execution within CI/CD pipelines without introducing meaningful latency.

4. Quantum Principles as Computational Framework

The three quantum principles each map to a specific phase of the selection engine. Together, they constitute the Quantum Selection Engine — the core computational architecture of Quantik Mind.

4.1 Superposition — Risk-Based Probabilistic Scoring

In quantum mechanics, a system exists in multiple states simultaneously until observed. The act of observation collapses the superposition into a definite state.

In Software Signal Engineering, every test exists in a superposition of 'necessary' and 'redundant' until the risk distribution is evaluated. The superposition phase scores each test against the current probability distribution. Tests above threshold τ are promoted to the selection set. Tests below are deferred — their execution postponed until the distribution shifts sufficiently to make them relevant again.

The selection is not permanent. A test suppressed in one cycle may be activated in the next if a signal shift elevates its risk score. The engine continuously re-evaluates all tests against the current distribution.

4.2 Entanglement — Adaptive Dependency-Aware Recalibration

In quantum mechanics, entangled particles influence each other regardless of physical separation. In distributed systems, services influence each other through dependency chains that may span multiple architectural hops.

The entanglement graph is a weighted directed graph where edges represent historical co-failure correlations between services. Critically, this graph is not static: it is continuously updated by the adaptive ML layer as new co-failure patterns emerge and historical correlations decay. The graph learns the true dependency topology of the system from observed behavior, not from declared architecture.

When superposition scoring identifies a service as high-risk, its entanglement relationships propagate risk elevation to connected services — even if those services were not directly modified by the triggering change. This mechanism detects cross-service failure propagation: a class of defects that deterministic full regression, by design, cannot prioritize.

4.3 Uncertainty — Exploration and Confidence Building

The uncertainty operator addresses a known failure mode of any adaptive selection system: convergence on a fixed selection pattern that leaves low-historical-risk areas permanently unobserved.

The uncertainty phase elevates risk scores for under-observed areas — services not tested recently, tests with insufficient execution history, and architectural areas that have never recorded a failure. This ensures periodic coverage across the full probability landscape, preventing the model from becoming blind to emerging risk in historically stable regions.

5. Observability as Signal Input

Static test history alone is insufficient for real-time risk modeling. Software Signal Engineering treats the live observability stack as a continuous signal source that modifies the probability distribution between execution cycles.

5.1 Signal Sources

Metrics	Infrastructure and application metrics — CPU saturation, memory pressure, error rates, latency percentiles per service. Anomalous values trigger risk elevation for affected services and their entangled dependencies.
Distributed tracing	Trace analysis across service boundaries reveals latency anomalies and error propagation patterns, providing higher-resolution risk attribution than aggregate metrics alone.
Logging	Structured log streams analyzed for error signatures and anomaly patterns. Novel signatures not present in historical data receive elevated scores due to high uncertainty value.
Commit webhook	Code change surface ingested in real time via webhook. Modified files are mapped to affected services and tests. Fix commits receive higher boosts than feature commits, reflecting elevated regression risk associated with defect corrections.

5.2 Signal Integration

Signals are ingested at selection time through a configurable lookback window (default: 5 minutes). Each signal type contributes a weighted modifier to the base risk score. Integration is additive — multiple concurrent signals compound, reflecting the real-world relationship between simultaneous system stresses and failure probability.

The observability integration is tooling-agnostic. Any monitoring, logging, or tracing stack that exposes standard interfaces is compatible. Specific tooling details are described in the empirical validation environment (Section 7.1).

6. The Selection Engine — Architecture and Flow

6.1 System Architecture

The Quantik Mind selection engine operates as a stateless API layer integrated into the CI/CD pipeline. It intercepts the execution decision before tests are dispatched to the test runner, computes the selection set, and returns a prioritized list with full decision transparency.

The engine is framework-agnostic — compatible with any test runner that accepts a test list via API. It operates in two modes: **shadow mode**, where selection is computed but the full suite executes (used during calibration); and **enforced mode**, where only selected tests execute.

6.2 Selection Flow

Step 1: Context Ingestion. Commit metadata delivered via webhook, observability signals, and the current test library are loaded. Service change magnitudes and volatility scores are computed from the diff surface and recent telemetry.

Step 2: Distribution Recalibration. The risk probability distribution is updated using Bayesian inference across all active signals. Historical priors are combined with current evidence to produce the posterior distribution.

Step 3: Superposition Scoring. Each test receives a final risk score combining historical priors, signal modifiers, commit boosts, and flakiness penalties.

Step 4: Entanglement Expansion. Tests for services entangled with high-risk services are evaluated for inclusion, even if their direct risk scores fall below τ. The entanglement graph is dynamically updated by the adaptive ML layer.

Step 5: Uncertainty Adjustment. Under-observed areas receive score elevation to maintain exploration coverage across the full test landscape.

Step 6: Always-Green Filtering. Tests with consistently stable outcomes and no active risk signals are filtered, reducing noise without compromising coverage.

Step 7: Output and Attribution. The selected test list is returned via API with full decision attribution: quantum origin (superposition / entanglement / uncertainty), risk score, and human-readable selection reason.

6.3 Decision Transparency

Every selection decision is fully auditable through the decision-insight endpoint. The response returns per-test attribution showing which signals contributed to inclusion or exclusion, with confidence scores for each decision. This satisfies traceability requirements in regulated environments — financial services, healthcare, and other sectors where validation decisions must be documented and defensible.

7. Empirical Validation

7.1 Experimental Environment

Validation was conducted on a production-grade distributed financial transaction system replicating enterprise-scale architectural complexity. The environment was selected to maximize the challenge surface — high service interdependency, dynamic load patterns, and a historical failure distribution typical of complex financial architectures.

Architecture	120+ microservices deployed on Google Kubernetes Engine (GKE)
Test suite	2,175 functional test scenarios with full service-level attribution
Observability	Prometheus (metrics), OpenTelemetry (tracing), Loki (logging)
Historical dataset	Two years of execution history — multi-cycle failure distribution across all services
Baseline	Deterministic full regression — 2,175 tests executed uniformly
Engine config	Quantik Mind selection engine, $\tau = 0.70$ (balanced mode)

7.2 Quantitative Results

77.1%	**95.5%**	**4.4×**
Reduction in redundant execution	Coverage of high-impact risk	Faster detection of critical failures

Table 1. Results against deterministic full regression baseline.

7.3 Result Interpretation

The 77.1% reduction figure represents tests correctly identified as carrying insufficient probability of revealing new information given the current system state. These tests were not discarded — they were deferred, remaining eligible for reactivation when distribution shifts warranted it.

The 95.5% risk coverage figure does not represent traditional code coverage. It represents coverage of dynamically modeled high-impact risk — the proportion of the risk probability mass addressed by the selected test set. Critically, this figure understates the actual quality advantage: by concentrating execution on high-risk areas rather than diluting it uniformly across the suite, the discipline surfaces defects that traditional full regression structurally cannot detect. Failures that would remain hidden in the noise of exhaustive execution

become visible precisely because execution is focused where uncertainty is highest.

The 4.4x acceleration in critical failure detection reflects concentrated execution: high-risk tests execute first and complete faster because low-information tests that would have preceded them in a sorted regression run are removed from the execution path.

8. Implications for Engineering Organizations

8.1 Operational Impact

Pipeline velocity. A 77% reduction in executed tests translates directly to pipeline time reduction. For organizations running multiple pipelines per hour, this compounds into hours of recovered engineering capacity daily.

Infrastructure cost. Cloud compute for CI/CD is billed by execution time. A 77% reduction in test execution reduces compute costs proportionally — typically representing 30–50% of total CI/CD infrastructure spend.

Carbon footprint. Compute reduction translates directly to CO_2 reduction. For a single project running daily pipeline cycles, Software Signal Engineering reduces emissions by approximately 150 kg of CO_2 annually — a measurable ESG contribution for organizations with sustainability commitments.

Defect detection quality. 4.4x faster critical failure detection means defects are surfaced before dependent commits accumulate. The blast radius of undetected failures shrinks. Engineering confidence in pipeline output increases because the signal-to-noise ratio of results improves.

8.2 Organizational Readiness

Software Signal Engineering requires a minimum level of architectural and operational maturity. Organizations that meet the following criteria are candidates for adoption:

— A defined test suite with service-level attribution — each test mapped to the service(s) it covers.

— An active observability stack exposing metrics, tracing, and logging through standard interfaces.

— CI/CD pipeline with API integration capability and webhook support.

— Willingness to operate in shadow mode during the calibration period, typically 2–4 weeks.

Organizations meeting these criteria typically achieve production-grade selection quality within 4–6 weeks of integration, as the model accumulates sufficient execution history to refine its probability distributions.

9. Conclusion

Software Signal Engineering is a response to a structural mismatch: a deterministic discipline applied to probabilistic systems. That mismatch has compounded quietly across the industry for years — in wasted compute, delayed feedback, and misplaced confidence.

The discipline and the framework presented in this paper address that mismatch at the root. By modeling risk as a dynamic probability field, incorporating real-time observability signals, applying adaptive machine learning to refine entanglement relationships, and using quantum-inspired computational principles for selection, Software Signal Engineering transforms validation from an execution ritual into a decision science.

The empirical results from the banking architecture validation are not projections. They are measured outcomes from a production-grade system under real operational conditions: 77.1% waste elimination, 95.5% risk coverage, 4.4x detection acceleration — against two years of historical baseline data.

These results represent the baseline performance of a model trained on a single architecture over a finite history. As the entanglement graph matures, as probability distributions accumulate evidence, as observability coverage expands, selection quality improves continuously.

"The next generation of engineering leaders will not measure coverage.
They will measure signal."

Software Signal Engineering is an open discipline. Its principles belong to every engineering organization willing to think beyond the deterministic frame. Its first industrial implementation is Quantik Mind.

Massimo Forno — Technical Founder & CEO

Quantik Mind GmbH · Switzerland · quantikmind.com

Founders of Software Signal Engineering