

# Beyond SaaS: The Architecture of Service as Software (SaaS<sub>W</sub>) and the Paradigm of Execution-as-a-System

Anteneh Tsegaye Tessema  
*Yaya Systems LLC, Indianapolis, Indiana*

April 2026

## Abstract

The traditional Software-as-a-Service (SaaS) model is reaching a structural inflection point. Enterprises increasingly experience rising integration costs, SaaS sprawl, duplicated workflows, fragmented data surfaces, and escalating human-in-the-loop coordination overhead. This paper formalizes the transition toward Service as Software (SaaS<sub>W</sub>): an architectural paradigm in which autonomous, governed, multi-agent execution systems displace the user interface as the primary locus of operational work.

Rather than arguing that SaaS becomes obsolete, we propose a sharper architectural separation: SaaS recedes into a deterministic Control Plane, while SaaS<sub>W</sub> emerges as the probabilistic but policy-bounded Execution Plane. To make this transition institutionally adoptable, we define the core invariants of SaaS<sub>W</sub> systems, introduce a formal Event Algebra for multi-agent coordination, and operationalize economic value through Expected Execution Yield ( $E(Y)$ ), a risk-adjusted, contract-verifiable performance metric. We further propose a Verifiable Execution Layer (VEL), incorporating cryptographic provenance, causal trace graphs, semantic traceability, and counterfactual replay to address the liability vacuum created by probabilistic execution. Finally, we present a failure-mode taxonomy, production-readiness requirements, and implementation case studies in autonomous ERP and legal operations.

The paper's central claim is that the next enterprise software transition is not merely from graphical interfaces to conversational interfaces, but from interface-centered software to execution-centered systems.

## 1 Introduction: The Decoupling of Interface from Execution

For the past two decades, SaaS democratized enterprise software by moving applications from local installation to cloud-hosted, multi-tenant delivery. This shift reduced procurement friction, improved deployment velocity, and enabled vendors to continuously ship product improvements. Yet the operational philosophy of SaaS remained largely static: the vendor provisions the tool, and the customer provisions the cognitive labor required to operate it.

As enterprise software portfolios expanded, the human user increasingly became the implicit integration layer. Employees copy data between tools, reconcile conflicting states, interpret dashboards, trigger workflows, chase approvals, and resolve exceptions across disconnected systems. In many organizations, the operational burden has moved from software acquisition to software coordination.

The core limitation of SaaS is therefore not hosting, pricing, or interface quality. It is that execution remains externalized to the human operator.

### 1.1 Definition of SaaS<sub>W</sub>

Service as Software (SaaS<sub>W</sub>) is the architectural decoupling of interface from execution. In the SaaS<sub>W</sub> paradigm, the user interface recedes into an observability, governance, and policy-

authoring layer. The human user transitions from a mechanical workflow operator to a strategic commander of constraints, objectives, and escalation rules.

A SaaS system is an asynchronous, event-driven execution substrate in which orchestrated multi-agent systems execute operational workflows autonomously while remaining bounded by deterministic constraints. Such systems may use large language models, small language models, symbolic policies, workflow engines, retrieval systems, rules engines, or domain-specific solvers, but their defining property is not model choice. Their defining property is that **software performs the service rather than merely presenting tools through which humans perform the service.**

## 1.2 What SaaS Is Not

SaaS should not be conflated with three adjacent but distinct categories.

- **Not a Chatbot:** A chatbot is primarily reactive: it waits for user prompts and produces conversational responses. SaaS systems are proactive, event-driven, and goal-oriented. Conversation may be one interface into the system, but it is not the execution substrate.
- **Not Traditional Automation:** Robotic Process Automation (RPA) and rules-based workflow tools are typically deterministic and brittle. They execute pre-authored scripts and fail when confronted with unexpected data, ambiguous context, or shifting process boundaries. SaaS systems instead combine probabilistic interpretation with deterministic governance.
- **Not a Replacement for SaaS:** SaaS remains essential as a system of record, identity boundary, audit surface, policy plane, and human-control interface. SaaS subsumes SaaS operationally, but does not eliminate it architecturally. The legacy SaaS layer becomes the Control Plane through which humans define the permissible envelope of autonomous execution.

## 1.3 Central Thesis

The enterprise software transition now underway is best understood as a migration from Software-as-a-Service to Service-as-Software. In SaaS, software exposes tools and humans perform the service. In SaaS, software executes the service and humans govern the execution.

This transition requires more than agent orchestration. It requires formal constraints, auditable state, verifiable execution, risk-adjusted economics, and institutional liability mechanisms. Without these components, agentic systems remain demonstrations. With them, they become enterprise infrastructure.

# 2 Related Work and Positioning

SaaS builds on several prior streams of work while proposing a distinct enterprise architecture.

## 2.1 Distributed Systems and Event-Sourced Architectures

The use of logical clocks, append-only event logs, event sourcing, and replicated data structures is well established in distributed systems. Lamport clocks provide causal ordering for distributed events, while CRDTs enable eventually consistent state convergence under concurrent writes. SaaS applies these principles to multi-agent enterprise execution, where autonomous agents emit state-mutating proposals that must remain auditable and causally ordered.

## 2.2 Workflow Automation and RPA

RPA systems demonstrated that repetitive human interface actions could be automated, but most RPA architectures remain tightly coupled to brittle UI paths and deterministic scripts. SaaS<sub>W</sub> differs by moving execution to semantic, event-driven, policy-bounded workflows rather than replaying interface actions.

## 2.3 Agentic AI Frameworks

Recent agent frameworks such as AutoGPT-style autonomous agents, LangGraph-style stateful orchestration, and SDK-based agent runtimes have shown that language-model agents can plan, call tools, coordinate across roles, and persist execution state. These systems are important precursors, but SaaS<sub>W</sub> emphasizes the enterprise requirements that sit above agent orchestration: contractual metrics, liability attribution, governance boundaries, counterfactual replay, auditability, and deployment discipline.

## 2.4 AI Risk and Governance Frameworks

AI governance frameworks emphasize safety, transparency, explainability, accountability, security, privacy, and reliability. SaaS<sub>W</sub> adopts these goals but translates them into concrete runtime architecture: confidence-gated transitions, policy precedence, deterministic escalation, immutable traces, counterfactual replay, and control-plane governance.

## 2.5 Distinction from Existing Categories

SaaS<sub>W</sub> is not merely a new label for AI agents, workflow automation, or SaaS plus copilots. Its distinctive contribution is architectural: it separates enterprise systems into a deterministic governance layer and a probabilistic execution layer, then binds them together through formal state semantics, verifiable execution records, and risk-adjusted value accounting.

# 3 Formal Ontology and Architectural Invariants

A probabilistic enterprise system cannot satisfy the safety envelope required for institutional trust without formal grounding. SaaS<sub>W</sub> therefore requires explicit definitions of authority, execution, state, and escalation.

## 3.1 Core Planes

- **Control Plane (SaaS):** The deterministic boundary layer where humans and organizations define constraints, policies, permissions, thresholds, audit rules, approval rights, and rollback procedures.
- **Execution Plane (SaaS<sub>W</sub>):** The asynchronous, event-driven substrate where orchestrated agents execute workflows, propose state transitions, call tools, reconcile data, and escalate exceptions.
- **Verification Plane (VEL):** The audit and forensic layer that records provenance, validates execution integrity, traces causal dependencies, and supports replay after failure.

## 3.2 Core Invariants

### Invariant 1: Policy Precedence

Deterministic constraints from the Control Plane strictly and universally dominate probabilistic agent outputs from the Execution Plane. No agent may override a policy boundary on the basis

of model confidence, historical performance, urgency, or inferred business value. If an execution path violates a policy, the action is blocked or escalated before state mutation.

**Invariant 2: Idempotent Mutability**

Every state-mutating action proposed by the Execution Plane must be idempotent. The system must survive retries, network partitions, duplicate messages, and at-least-once delivery semantics without corrupting authoritative state.

**Invariant 3: Trace-Complete Execution**

Every material decision must be trace-complete. The system must preserve the initiating event, input context, policy version, model version, retrieval context, tool calls, confidence scores, human approvals, and resulting state mutation.

**Invariant 4: Escalation Before Irreversible Harm**

For workflows with material legal, financial, medical, safety, reputational, or compliance impact, uncertainty must trigger escalation before irreversible action. The system may prepare, analyze, recommend, draft, reconcile, simulate, or queue; but it may not autonomously commit high-impact actions outside the approved risk envelope.

## 4 Confidence-Gated State Machines

Rather than requiring mathematical certainty from language-model outputs, SaaS employs **Confidence-Gated Transitions**.

Let an agent propose an action  $a$  with calibrated confidence score  $C(a) \in [0, 1]$ . The system transitions state only if:

$$C(a) \geq \tau$$

where  $\tau$  is the deterministic threshold defined in the Control Plane.

If  $C(a) < \tau$ , execution is suspended and a strict escalation protocol is triggered. The action is routed to human review, policy review, or a secondary verification mechanism before any state mutation occurs.

### 4.1 Calibration Requirements

Confidence scores must not be treated as raw model probabilities. Production systems must calibrate confidence using historical task performance, domain-specific validation sets, tool-call success rates, retrieval quality, policy risk tier, and post-execution audit outcomes.

### 4.2 Risk-Tiered Thresholds

A single global threshold is insufficient. SaaS systems should define thresholds by task class:

Risk Tier	Example Task	Required Behavior
Low	Categorizing support tickets	Autonomous execution allowed above low threshold
Medium	Updating CRM records	Execution allowed with audit trail and rollback
High	Approving vendor payment	Human approval required unless pre-authorized
Critical	Legal commitment or financial filing	Human authority required before commitment

### 4.3 Escalation as a First-Class Event

Human escalation must not be treated as a side channel. It is a first-class event in the execution graph, carrying its own timestamp, actor identity, policy context, decision payload, and effect on system state.

## 5 Formal Event Algebra and State Authority

To prevent state divergence when asynchronous agents act concurrently, SaaS requires formal event semantics.

Let  $E$  be the set of all system events. Each event  $e \in E$  is a tuple:

$$e = \langle id, type, payload, t_{logical}, agent\_id, policy\_version, trace\_id, hash\_prev \rangle$$

where:

- $id$  is a globally unique event identifier.
- $type$  identifies the event class.
- $payload$  contains structured event data.
- $t_{logical}$  is a Lamport timestamp establishing causal ordering.
- $agent\_id$  identifies the agent or human actor that emitted the event.
- $policy\_version$  identifies the governing policy snapshot.
- $trace\_id$  links the event to its causal execution trace.
- $hash\_prev$  links the event to the prior event hash for tamper-evident provenance.

The authoritative system state  $S$  at any point is not a mutable database record but the deterministic projection of an ordered, append-only event log:

$$S = \pi(e_1, e_2, \dots, e_n) \quad \text{subject to} \quad t_{logical}(e_i) \leq t_{logical}(e_{i+1})$$

### 5.1 Concurrent Writes

To handle concurrent writes from distributed agents without coordination locks, state projections should employ CRDTs where appropriate. CRDT-backed projections allow eventual consistency across the execution swarm while maintaining deterministic merge semantics.

### 5.2 Vector Clocks and Partial Ordering

Logical vector clocks establish a valid partial order over agent-emitted events. This prevents silent state divergence while preserving asynchronous throughput. Events that cannot be causally ordered must be explicitly resolved through deterministic merge rules or escalated to the Control Plane.

### 5.3 State Authority Principle

Agents may propose state transitions, but they do not own authoritative state. Authority resides in the event log, projection rules, policy constraints, and approved system-of-record integrations.

## 6 Verifiable Execution Layer (VEL)

The most significant barrier to enterprise SaaS adoption is liability. Traditional logging proves that an action occurred, but often fails to establish why it occurred, which policy permitted it, which model generated it, what evidence informed it, and whether the result was reproducible.

SaaS platforms therefore require a Verifiable Execution Layer (VEL) built on a Forensic Logic Layer with five core capabilities.

### 6.1 Cryptographic Integrity

Every event should be written to an append-only log with hash chaining. Each event hash should commit to the event payload, previous event hash, timestamp, actor identity, policy version, and trace identifier. This produces tamper-evident provenance suitable for internal audit and external review.

### 6.2 Causal Trace Graphs

The VEL should construct directed acyclic graphs that trace outcomes backward through input data, retrieved context, prompts or instructions, model outputs, tool calls, intermediate decisions, policy checks, human approvals, and final mutations. The goal is causal attribution rather than mere chronological logging.

### 6.3 Semantic Traceability

Trace records should preserve semantic reasoning artifacts in a controlled, privacy-aware manner. The system does not need to expose every internal token-level representation, but it must preserve enough structured explanation to determine why a decision was made and whether it complied with policy.

### 6.4 Counterfactual Replay

Upon failure, the VEL reconstructs the precise execution context using version-locked snapshots of data state, model configuration, retrieval corpus, tool schema, policy configuration, and execution environment. The scenario is then replayed deterministically where possible, or statistically replayed over bounded stochastic parameters where deterministic replay is impossible.

Counterfactual replay allows auditors to answer three questions:

1. Would the failure have occurred under the same model and policy configuration?
2. Would the failure have been prevented under a different threshold, policy, tool contract, or data validation layer?
3. Was the root cause model behavior, policy design, data quality, tool failure, human approval, or integration drift?

### 6.5 Liability Mapping

Every material execution trace should map responsibility across actors and system components: Policy owner, Model/runtime owner, Data source owner, Tool/integration owner, Human approver (if applicable), Vendor or platform provider, and Customer administrator. This mapping converts probabilistic execution from a liability vacuum into an adjudicable record.

## 7 Operationalized Metrics: Expected Execution Yield

Seat-based SaaS metrics become less meaningful when software performs work autonomously. Pure outcome-based pricing, however, is often rejected by enterprises because attribution disputes are difficult to resolve. SaaS therefore benefits from a hybrid economic model: baseline platform subscription plus contract-verifiable performance upside.

We define Expected Execution Yield ( $E(Y)$ ) as:

$$E(Y) = \sum_i [\Delta V_i - C_i - (P(E_i) \cdot L_i)]$$

where:

- $\Delta V_i$  is the measurable value generated for task  $i$  above the established human-execution baseline.
- $C_i$  is infrastructure, compute, data, and operational cost attributable to task  $i$ .
- $P(E_i)$  is the calibrated probability of systemic error for task  $i$ .
- $L_i$  is the liability impact or tail-risk cost associated with failure in task  $i$ .

The term  $P(E_i) \cdot L_i$  functions as a risk-adjusted penalty. It ensures that low-probability but catastrophic failures materially reduce expected yield, creating an economic incentive to escalate rather than execute autonomously in high-stakes settings.

### 7.1 Contract-Verifiable Baselines

For  $E(Y)$  to support enterprise pricing or service-level agreements, the human baseline must be contractually defined. Baselines may include historical average completion time, human labor cost per task, error rate under manual execution, rework cost, cycle-time reduction, compliance or audit cost reduction, revenue acceleration, and customer response-time improvement.

### 7.2 Measurement Discipline

SaaS vendors should avoid claiming value from raw throughput alone. A system that executes more tasks but increases downstream rework may have negative yield.  $E(Y)$  must therefore be measured against validated outcomes, not merely activity volume.

## 8 Production-Readiness Requirements

A SaaS system becomes production-ready only when it satisfies operational controls beyond model accuracy.

- **Identity and Access Control:** All agent actions must execute under scoped identities. Agents should not inherit broad administrator privileges. Each tool call should be governed by least privilege, explicit authorization, and policy-scoped credentials.
- **Tool Contracts:** Every tool available to an agent must have a typed contract defining input schema, output schema, side effects, retry semantics, idempotency guarantees, timeout behavior, and rollback strategy. Tools that mutate state require stricter gating than tools that only retrieve information.
- **Environment Separation:** Production SaaS deployments should separate development, staging, simulation, and production environments. Agents must be tested against realistic synthetic and historical event streams before being allowed to operate against live systems.

- **Observability:** A production SaaS platform should expose metrics for task success rate, escalation rate, false autonomy rate, false escalation rate, tool-call failure rate, mean time to recovery, mean time to human approval, policy violation attempts, drift in confidence calibration, cost per successful execution, and risk-adjusted yield.
- **Rollback and Compensation:** Not all actions are reversible. Where rollback is possible, it should be automated and tested. Where rollback is impossible, the system should use compensating transactions, approval gates, or pre-commit simulation.
- **Data Governance:** Production SaaS requires strong data lineage, retention rules, privacy controls, tenant isolation, PII handling, and deletion workflows. Retrieved context should be versioned so that downstream decisions can be audited against the exact information available at execution time.
- **Security Controls:** SaaS systems introduce new attack surfaces. Production systems should implement ingress sanitization, allowlisted tools, constrained execution, adversarial testing, secret isolation, and human approval for sensitive actions.

## 9 Failure Modes and Mitigation Strategies

A resilient SaaS architecture must anticipate systemic failure. The following taxonomy defines critical failure modes and corresponding mitigations.

## 10 Reference Architecture

A production SaaS platform should contain the following layers:

1. **Data Reliability Layer:** Normalizes, validates, deduplicates, enriches, and quarantines incoming data before events enter the execution graph.
2. **Event Backbone:** Provides durable event ingestion, ordering, replay, subscription, and delivery semantics. Supports event sourcing, audit reconstruction, and asynchronous agent execution.
3. **Agent Orchestration Layer:** Coordinates agents, tool calls, memory, retrieval, planning, execution state, human handoffs, and retries. Supports durable execution, interrupted workflow resumption, and explicit state checkpoints.
4. **Policy Engine:** Evaluates proposed actions against deterministic policies, thresholds, permissions, risk tiers, contractual obligations, and customer-specific governance rules.
5. **Verification Layer:** Records causal trace graphs, cryptographic provenance, replay meta-data, evaluation outcomes, and liability mappings.
6. **Human Governance Interface:** Provides dashboards for policy authoring, approval queues, audit review, escalation handling, risk analytics, and execution observability.

## 11 Implementation Case Studies

### 11.1 Autonomous Enterprise Resource Planning

Traditional ERP failures often stem from data ingestion chaos, schema drift, manual reconciliation, and delayed exception handling across heterogeneous systems. A SaaS ERP mitigates



Failure Mode	Description	Mitigation Strategy
<b>Alignment Faking</b>	Agents appear to follow policy during testing but deviate in production.	Runtime policy enforcement, adversarial testing, interpretability monitors, and boundary checks.
<b>Cascading Hallucination</b>	A low-confidence or incorrect output propagates across dependent stages.	Deterministic verification checkpoints; outputs below threshold are held and escalated.
<b>State Divergence</b>	Concurrent agent proposals produce conflicting mutations.	CRDTs, vector clocks, deterministic merge rules, and escalation for unresolved conflicts.
<b>Data Ingestion Chaos</b>	Malformed, adversarial, stale, or schema-drifted inputs corrupt execution.	Data Reliability Layer with schema validation, sanitization, and quarantine workflows.
<b>Tool Misuse</b>	Agent calls a valid tool with invalid intent or unsafe parameters.	Typed tool contracts, policy-scoped tool access, dry-run simulation, and mutation gates.
<b>Over-Autonomy</b>	System autonomously executes actions requiring human judgment.	Risk-tiered autonomy boundaries, approval policies, and post-execution audit sampling.
<b>Under-Autonomy</b>	System escalates too often and fails to deliver productivity gains.	Calibration review, workflow decomposition, better retrieval, and targeted policy relaxation.
<b>Retrieval Poisoning</b>	Agent relies on manipulated or untrusted documents.	Source trust scoring, document quarantine, signed knowledge bases, and citation bounds.
<b>Integration Drift</b>	External APIs, schemas, or permissions change unexpectedly.	Contract tests, integration health checks, versioned adapters, and fallback to escalation.
<b>Silent Cost Explosion</b>	Autonomy increases compute or tool costs beyond the value produced.	Cost budgets, per-task cost attribution, $E(Y)$ monitoring, and execution throttles.

this through a Data Reliability Layer that normalizes noisy ingestion before event-driven agents process financial events.

In a production-grade autonomous ERP, the system operates as a continuous ledger rather than a periodic reporting tool. Financial events are validated, classified, reconciled, and posted according to deterministic policy boundaries. Cryptographic parity is maintained between live ledger state and externally certifiable audit kits. Human finance leaders retain authority over reporting-period certification, material adjustments, and policy changes. This design positions the CFO as a policy authority rather than an operational bottleneck.

## 11.2 Canonical Legal Operating System

Legal systems are adversarial, interpretive, and game-theoretic. A SaaS LegalOS should therefore avoid fully autonomous negotiation, commitment, or adversarial representation. Instead, it should construct a canonical legal data layer by extracting clauses, identifying ambiguity, mapping counterparty positions, and scoring risks against internal playbooks.

The system can autonomously perform analysis, retrieval, comparison, redline preparation, obligation tracking, and risk surfacing. However, human lawyers retain authority over negotiation strategy, legal advice, settlement, commitment, and final approval. This architecture

achieves productivity gains while preserving the accountability structure required in legal contexts.

### 11.3 Customer Operations

A SaaS customer operations system can triage tickets, retrieve account history, draft responses, issue refunds within policy limits, escalate sensitive issues, and update records. The Control Plane defines refund limits, tone policies, escalation triggers, compliance language, and customer segmentation rules. The Execution Plane performs the work. The Verification Layer records why each action was taken and whether the outcome met service-level expectations.

## 12 Evaluation Methodology

SaaS systems should be evaluated across five dimensions:

1. **Task Performance:** Measures whether the system completes the intended task accurately, efficiently, and consistently.
2. **Governance Compliance:** Measures whether every action respects policy constraints, access controls, approval requirements, and risk-tier boundaries.
3. **Trace Completeness:** Measures whether auditors can reconstruct the causal path from initial event to final outcome.
4. **Economic Yield:** Measures whether the system produces positive risk-adjusted  $E(Y)$ , not merely higher activity volume.
5. **Resilience:** Measures how the system behaves under tool failure, network partition, malformed input, adversarial prompt injection, integration drift, stale retrieval context, and human non-response.

## 13 Limitations and Open Questions

The SaaS paradigm introduces unresolved research and implementation challenges:

- Confidence calibration remains difficult, especially across heterogeneous tasks and rapidly changing business contexts.
- Deterministic replay of probabilistic systems is only partially achievable unless model versions, seeds, retrieval corpora, prompts, and tool environments are tightly controlled.
- Liability mapping may become contentious when failures emerge from interactions among model vendors, application developers, customer policies, external tools, and human approvers.
- Excessive governance can eliminate productivity gains, while insufficient governance can create unacceptable risk.

These limitations do not invalidate SaaS. They define the engineering and institutional work required to make SaaS reliable.

## 14 Conclusion

The transition from SaaS to SaaS<sub>w</sub> is a fundamental re-engineering of enterprise architecture. It is not a product category, branding shift, or conversational interface upgrade. It is the migration from software that presents tools to software that executes services.

By decoupling the interface into a governance-oriented Control Plane, formalizing distributed execution through event algebra and deterministic state projections, and operationalizing value through Expected Execution Yield, organizations gain a coherent migration path toward autonomous execution. The Verifiable Execution Layer closes the remaining institutional gap by transforming probabilistic agent behavior into an auditable, replayable, and contractually interpretable execution record.

The immediate frontier is threefold: standardizing VEL interfaces across enterprise platforms; calibrating  $E(Y)$  models to sector-specific liability profiles; and extending Event Algebra to treat human escalation, approval, override, and refusal as first-class causal events.

SaaS<sub>w</sub> will succeed not because agents become unconstrained, but because autonomous execution becomes governable.

## References

- [1] Lamport, L. (1978). “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, 21(7), 558–565.
- [2] Shapiro, M., Preguiça, N., Baquero, N., & Zawirski, M. (2011). “Conflict-Free Replicated Data Types.” *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Lecture Notes in Computer Science, vol. 6976. Springer.
- [3] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). “Re-Act: Synergizing Reasoning and Acting in Language Models.” *International Conference on Learning Representations (ICLR 2023)*.
- [4] National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. U.S. Department of Commerce.
- [5] National Institute of Standards and Technology. (2024). *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile (NIST AI 600-1)*. U.S. Department of Commerce.
- [6] Significant Gravitas. *AutoGPT: Build, Deploy, and Run AI Agents*. Open-source software repository.
- [7] LangChain. *LangGraph Documentation: Durable Execution and Stateful Agent Workflows*.
- [8] OpenAI. *Agents SDK Documentation: Tracing, Guardrails, Handoffs, and Tool Execution*.