Open Agent Trust Stack (OATS): A System Specification for Zero-Trust AI Agent Execution

Version 1.1.0

Jascha Wanger, ThirdKey AI

April 2026

Abstract

As AI systems evolve from assistants into autonomous agents executing consequential actions, the security boundary shifts from model outputs to tool execution. Traditional security paradigms—log aggregation, perimeter defense, post-hoc forensics, and runtime interception of fully-formed actions—cannot adequately protect systems where AI-driven actions are irreversible, execute at machine speed, and originate from potentially compromised orchestration layers. This paper introduces the Open Agent Trust Stack (OATS), an open specification for zero-trust AI agent execution. OATS is built on three architectural convictions. First, allowlist enforcement: rather than intercepting arbitrary actions and deciding which to block, OATS constrains what actions can be expressed through declarative tool contracts, making dangerous actions structurally inexpressible. Second, compile-time enforcement: the Observe-Reason-Gate-Act (ORGA) reasoning loop uses typestate programming so that skipping the policy gate is a type error, not a runtime bug. Third, structural independence: the Gate phase is architecturally isolated from LLM influence. OATS specifies five layers: (1) the ORGA reasoning loop with compile-time phase enforcement, (2) declarative tool contracts with typed parameter validation, (3) a cryptographic identity stack providing bidirectional trust between agents and tools, (4) a formally verifiable policy engine operating on structured inputs, and (5) hash-chained cryptographic audit journals with Ed25519 signatures for tamper-evident forensic reconstruction. OATS is model-agnostic, framework-agnostic, and vendor-neutral. The architecture is informed by operational experience with a production runtime (Symbiont) that has operated autonomously for approximately eight months; however, rigorous empirical evaluation remains ongoing and this version of the specification should be read as an architectural contribution with an accompanying evaluation framework rather than a fully validated system.

1 1. Introduction

1.1 1.1 The Runtime Security Gap

AI agents now execute consequential actions across enterprise systems: querying databases, sending communications, modifying files, invoking cloud services, and managing credentials. Through function calling, plugins, external APIs, and protocol-based tool servers such as the Model Context Protocol (MCP) [1], these agents perform multi-step tasks without human intervention [2], [3].

These actions exhibit five characteristics that existing security paradigms cannot adequately address:

1. **Irreversibility.** Tool executions produce immediate and often permanent effects: database mutations, financial transactions, credential changes, or data exfiltration [4], [5]. Once executed, the damage is done.
2. **Speed.** Agents execute hundreds of tool calls per minute, far exceeding human capacity for real-time review [6]. Multi-step attack chains complete within seconds [7].
3. **Compositional risk.** Individual actions may each satisfy policy while their composition constitutes a violation [8]. Reading a confidential file is permitted; sending email is permitted; doing both in sequence may constitute exfiltration.
4. **Untrusted orchestration.** Prompt injection and indirect instruction attacks mean the model's apparent intent cannot be trusted [9], [10]. Adversarial prompts can be embedded in documents, emails, and images that agents process [11].

5. **Privilege amplification.** Agents routinely operate under static, high-privilege identities misaligned with the principle of least privilege [12].

The gap in the current security landscape lies at the intersection of prevention and context-awareness: no existing system can block actions before execution based on both static policy and accumulated session context while simultaneously constraining what actions can be expressed in the first place. This is the gap that OATS addresses.

This paper makes two contributions: a normative system specification defining the runtime enforcement boundary for autonomous agent execution, and an implementation-grounded evaluation methodology derived from operational experience with a production runtime. The specification is the primary artifact; the evaluation framework (Section 14) is included to make the claims falsifiable and to enable comparable evaluation of future implementations. The contribution is a new runtime security abstraction with testable conformance properties, not a benchmark of one particular system.

1.2 1.2 Design Principles

OATS is built on three architectural convictions, each addressing a structural weakness in current approaches.

Allow-list over deny-list. Current runtime security approaches operate on a deny-list model: the agent formulates an action, a security system intercepts it, evaluates it, and decides whether to allow or block. This requires enumerating dangerous behavior—an enumeration that is incomplete by definition. OATS inverts this model. The agent fills typed parameters defined by a declarative tool contract. The runtime validates parameters against the contract, constructs the invocation from a template, and executes. The agent never generates raw commands or constructs unconstrained API calls. Within the scope of contracted tools, dangerous actions cannot be expressed because the interface does not permit them. Actions that bypass the contract layer entirely (e.g., direct network calls from compromised agent code) require complementary controls such as sandboxing (Section 10).

Compile-time over runtime enforcement. When enforcement correctness is verified only at runtime, a code change that introduces a path bypassing the policy engine goes undetected until that path is exercised. OATS addresses this through the Observe-Reason-Gate-Act (ORGA) cycle, which uses type-level programming (typestates) so that skipping the Gate phase, dispatching tools without reasoning first, or observing results without dispatching are compile-time errors. In a correctly implemented typestate, the type system enforces that every action passes through policy evaluation. This property holds for code paths within the typestate-governed loop; it does not extend to code that circumvents the loop entirely, which is why sandboxing and network isolation provide complementary enforcement.

Structural independence over trust assumptions. When the policy engine shares context, memory, and execution environment with the orchestration layer it governs, an LLM compromised through prompt injection can potentially influence the evaluation of its own actions. In OATS, the Gate phase receives a structured action proposal and evaluates it against policy using a formally verifiable policy engine. The LLM cannot modify, bypass, or influence the Gate's evaluation.

1.3 1.3 Contributions

This specification makes five contributions:

1. **Typestate-enforced reasoning loop.** We specify the ORGA cycle with compile-time phase enforcement, designed to prevent policy evaluation from being skipped, circumvented, or reordered within the loop (Section 5).
2. **Allow-list tool contracts.** We specify a declarative tool contract format that constrains agent-tool interaction to typed, validated parameters, making dangerous actions structurally inexpressible (Section 6).
3. **Layered cryptographic identity.** We specify a bidirectional identity stack providing mutual authentication between agents and tools via domain-anchored cryptographic verification (Section 7).
4. **Hash-chained audit journals.** We specify cryptographically signed, hash-chained event journals for tamper-evident forensic reconstruction (Section 9).
5. **Conformance requirements.** We define minimum requirements for OATS-compliant systems, enabling objective evaluation of implementations (Section 12).

OATS's novelty is not any single component in isolation—typestates, policy engines, cryptographic signatures, audit logs, and sandboxing each have extensive prior art. The contribution is the integration of five layers into a unified runtime security model centered on consequential action execution, with three properties not found in prior work in combination: (a) expressibility constraints that eliminate action categories before policy evaluation, (b) compile-time enforcement that the policy gate executes on every dispatch path within the loop, and (c) bidirectional cryptographic identity binding actions to verified agents and verified tools. The conformance requirements formalize these properties into testable criteria, enabling objective comparison across implementations.

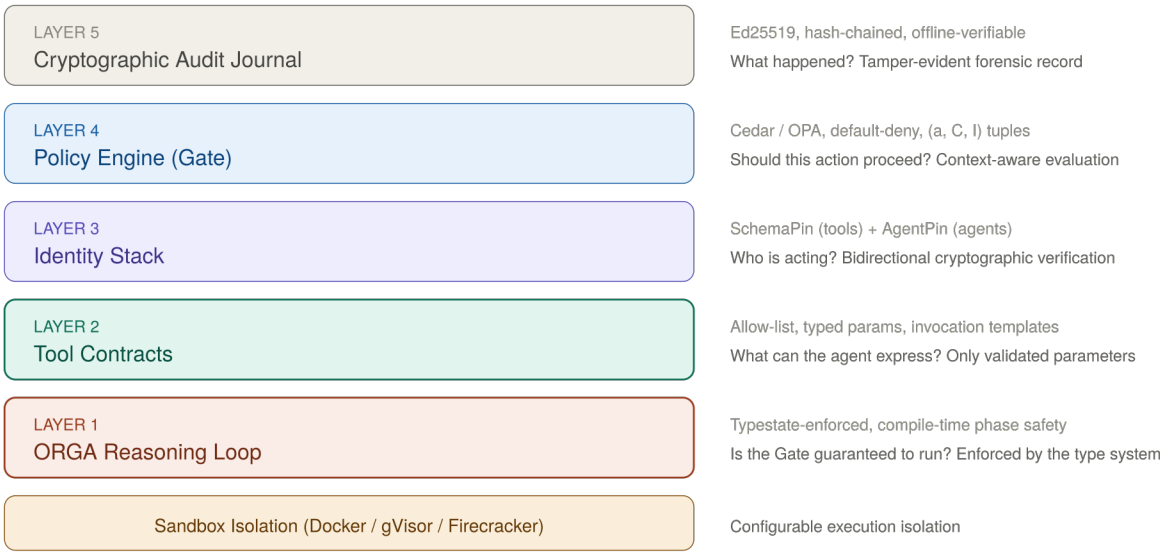


Figure 1: Figure 0. OATS trust stack. Five layers, each addressing a distinct security question. The ORGA loop (Layer 1) enforces that the Gate runs; tool contracts (Layer 2) constrain expressible actions; the identity stack (Layer 3) provides mutual authentication; the policy engine (Layer 4) evaluates authorization; the audit journal (Layer 5) records decisions.

1.4 1.4 Document Structure

Section 2 reviews related work. Section 3 formalizes the problem and defines the system model. Section 4 characterizes the threat landscape. Section 5 presents the ORGA loop architecture. Section 6 specifies the tool contract layer. Section 7 specifies the identity layer. Section 8 specifies the policy enforcement layer. Section 9 specifies the audit layer. Section 10 addresses sandboxing and isolation. Section 11 addresses inter-agent communication. Section 12 defines conformance requirements. Section 13 presents implementation architectures. Section 14 defines the evaluation framework for empirical validation. Section 15 identifies known limitations. Section 16 identifies open research directions. Section 17 concludes.

2 2. Related Work

2.1 2.1 Agent Security Research

The security risks of LLM-based agents have been catalogued by several surveys. Ruan et al. [8] and Wu et al. [4] provide comprehensive threat taxonomies covering prompt injection, tool misuse, and data exfiltration. Su et al. [6] focus on autonomy-induced risks including memory poisoning and deferred decision hazards. Debenedetti et al. [7] introduce AgentDojo for evaluating attacks and defenses against LLM agents, while Ye et al. [5] propose ToolEmu for identifying risky agent failures. These works characterize the problem space but do not propose runtime enforcement architectures. OATS builds on their threat models and contributes a system specification for constraining and evaluating actions before execution.

Gaire et al. [13] systematize security and safety risks in the Model Context Protocol ecosystem, providing a taxonomy of threats to MCP primitives. Their analysis of tool poisoning and indirect prompt injection directly informs OATS's threat model for tool supply chain attacks.

2.2 2.2 Runtime Security Specifications

Errico [14] introduces Autonomous Action Runtime Management (AARM), a system specification for securing AI-driven actions at runtime. AARM formalizes the runtime security gap, proposes an action classification framework distinguishing forbidden, context-dependent deny, context-dependent allow, and context-dependent defer actions, and specifies conformance requirements for pre-execution interception, context accumulation, policy evaluation, and tamper-evident receipts. OATS shares AARM's identification of the action layer as the stable security boundary and incorporates its context-dependent action classification. OATS extends this foundation with compile-time enforcement of the reasoning loop, allow-list tool contracts, concrete cryptographic identity protocols, and multi-tier execution isolation.

2.3 2.3 Industry Frameworks

Google's Cloud CISO perspective [15] advocates defense-in-depth and runtime controls for agents. AWS's Agentic AI Security Scoping Matrix [16] provides a risk assessment framework complementing runtime enforcement with deployment-time scoping. Microsoft's governance framework [17] addresses organizational controls including identity management and approval workflows. Raza et al. [18] present a TRiSM framework for agentic multi-agent systems. These frameworks provide lifecycle governance; OATS focuses specifically on the runtime enforcement layer.

2.4 2.4 Access Control and Policy Languages

OATS's policy enforcement layer builds on established access control research. RBAC and ABAC evaluate permissions against static attributes but lack session-level context accumulation [12]. Capability-based security [19] constrains authority propagation but does not address compositional risks of non-deterministic agents. Policy languages such as OPA [20] and Cedar [21] provide expressive evaluation engines suitable as backends for OATS's policy evaluation component. AWS's independent adoption of Cedar for AgentCore provides supporting evidence for the thesis that formal policy languages belong at the agent execution boundary.

2.5 2.5 Complementary Standards

OATS is complementary to several existing standards. The OWASP Top 10 for LLM Applications [22] catalogs vulnerabilities; OATS provides runtime enforcement mitigating tool misuse, excessive agency, and insecure output handling. The NIST AI RMF [23] provides a risk management framework; OATS provides technical enforcement implementing portions around governance, monitoring, and accountability. MCP [1] defines agent-tool communication; OATS defines how to govern actions flowing through that (or any) tool invocation mechanism.

2.6 2.6 Comparative Positioning

Table 0 positions OATS relative to existing approaches across the security properties specified in this paper. Checkmarks indicate the property is addressed by the approach; dashes indicate it is not.

Table 1: **Table 0.** Comparative positioning. "Partial" indicates the property is architecturally possible but not structurally enforced by the specification. AgentDojo [7] and ToolEmu [5] are evaluation frameworks rather than enforcement runtimes and are not included; they are complementary to OATS rather than alternatives.

Property	Prompt guardrails	Deny-list filter	Sandbox-only	AARM [14]	OATS
Constrains expressible actions	–	–	–	–	Yes
Pre-execution policy gate	–	Yes	–	Yes	Yes
Session context accumulation	–	–	–	Yes	Yes
Cryptographic tool identity	–	–	–	–	Yes
Cryptographic agent identity	–	–	–	–	Yes
Tamper-evident audit journal	–	–	–	Yes	Yes
Compile-time phase enforcement	–	–	–	–	Yes
Gate independent of LLM	–	Partial	N/A	Partial	Yes
Execution isolation	–	–	Yes	–	Yes
Formal conformance criteria	–	–	–	Yes	Yes

3 3. Problem Formalization

3.1 3.1 System Model

Let an AI-enabled application $\mathcal{A}$ consist of:

- An orchestration layer O (agent framework, workflow engine, or application code) that interprets user requests and invokes tools. O includes the LLM, prompt templates, memory systems, and control flow logic. Crucially, O processes untrusted inputs and cannot be assumed to behave as intended [4].
- A set of tools $T = \{t_1, t_2, \ldots, t_n\}$, where each tool t_i exposes operations producing effects on external systems.
- An identity context I comprising four layers: human principal, service identity, agent/session identity, and role/privilege scope.
- An environment E including data stores, APIs, cloud services, and enterprise systems.
- A session context C that accumulates state over the course of an interaction.

3.2 3.2 Action Definition

An action a is a discrete operation the agent requests against a tool:

$$a = (t, op, p, id, ctx, ts)$$

where $t \in T$ is the target tool, op is the specific operation, p is the parameter set, $id \in I$ is the identity context, $ctx \in C$ is the accumulated session context, and ts is the timestamp.

3.3 3.3 Tool Contract Definition

A tool contract κ defines the complete behavioral interface for a tool:

$$\kappa = (name, \Pi, \tau, \sigma_{out}, \mu, \rho)$$

where $\Pi = \{\pi_1, ..., \pi_m\}$ is the set of typed parameter definitions, τ is the invocation template, σ_{out} is the output schema, μ is the policy metadata (resource, action), and ρ is the risk tier.

Each parameter definition $\pi_i = (name_i, type_i, V_i, req_i)$ specifies the parameter name, its type from the type system $\mathcal{T}$, validation constraints V_i, and whether it is required.

The type system $\mathcal{T}$ provides:

$$\mathcal{T} = \{string, integer, boolean, enum, scope_target, url, path, ip_address, cidr, port\}$$

Each type $\tau \in \mathcal{T}$ has an associated validation function $v_\tau : \text{Value} \to \{valid, invalid\}$ and a sanitization function $s_\tau : \text{Value} \to \text{Value}$ that strips dangerous characters.

3.4 3.4 Constrained Action Formulation

Under OATS, the agent does not formulate arbitrary actions. Instead, the agent proposes a *parameterized invocation*:

$$a' = (t, op, p')$$

where $p' = \{(name_i, val_i)\}$ maps parameter names to values. The runtime validates each value:

$$\forall (name_i, val_i) \in p' : \quad v_{type_i}(val_i) = valid$$

If validation succeeds, the runtime constructs the executable action from the template:

$$a_{exec} = \tau(p') \quad \text{where } \tau \text{ is the invocation template}$$

The agent never sees or constructs a_{exec}. This is the allow-list property: the space of expressible actions is constrained to $\{a_{exec} : a_{exec} = \tau(p'), \forall (name_i, val_i) \in p', v_{type_i}(val_i) = valid\}$.

3.5 3.5 Context Accumulation

Session context accumulates across actions:

$$C_n = C_{n-1} \cup \{a_n, o_n, \delta_n\}$$

where C_n is the context after action n, a_n is the action, o_n is its output, and δ_n represents derived signals including data classification, semantic distance from the original request, scope expansion indicators, entity references, and confidence level.

3.6 3.6 Policy Structure

A policy $\pi \in \Pi$ maps an action-context-identity triple to an authorization decision:

$$\pi : (a, C, I) \to \{ALLOW, DENY, MODIFY, STEP_UP, DEFER\}$$

Each policy consists of a match predicate $m(a, C, I) \to \{true, false\}$, a decision d, a priority $p \in \mathbb{N}$, and an optional modification function $f(a) \to a'$ applied when $d = MODIFY$.

3.7 3.7 Security Objectives

An OATS-compliant runtime must ensure that for all actions a:

1. **Structural constraint.** a is expressible only through a valid tool contract κ.
2. **Pre-execution interception.** a is intercepted and evaluated before any effects occur.
3. **Compile-time gate enforcement.** All code paths from action proposal to tool dispatch within the ORGA loop pass through the Gate phase; in typestate implementations, this is enforced at compile time.
4. **Policy compliance.** a satisfies organizational policy Π given context C and identity I.
5. **Context-aware evaluation.** a is evaluated against both static policy and accumulated session context.
6. **Identity verification.** Both the agent invoking a tool and the tool being invoked are cryptographically verified.
7. **Forensic completeness.** Every action, its context, the policy decision, and the execution outcome are recorded in a tamper-evident journal.

4 4. Threat Model

OATS operates on a fundamental assumption: **the AI orchestration layer O cannot be trusted as a security boundary.** The model processes untrusted inputs through opaque reasoning, producing actions that may serve attacker goals rather than user intent.

4.1 4.1 Threat Summary

Table 1 summarizes the primary threats, their attack vectors, and the OATS controls that mitigate them.

Table 2: **Table 1.** Threat summary and OATS mitigations.

Threat	Attack Vector	OATS Control
Prompt injection	User input, documents, tool outputs, images	Tool contracts (structural), policy enforcement, context-dependent deny
Malicious tool outputs	Adversarial tool responses	Post-tool action restrictions, context tracking, output schema validation
Confused deputy	Ambiguous/malicious instructions	Bidirectional identity verification, step-up approval, intent alignment
Over-privileged credentials	Excessive token scopes	Least-privilege enforcement, scoped credentials
Data exfiltration	Action composition	Context accumulation, compositional policies, scope enforcement
Goal hijacking	Injected objectives	Action-level policy, semantic distance tracking
Intent drift	Agent reasoning divergence	Context accumulation, semantic distance threshold, deferral
Memory poisoning	Persistent context manipulation	Provenance tracking, anomaly detection, journal comparison
Tool supply chain	Tampered contracts, spoofed tools	Cryptographic tool integrity verification, TOFU pinning

Threat	Attack Vector	OATS Control
Cross-agent propagation	Multi-agent delegation	Cross-agent context, transitive trust limits, blast-radius containment

4.2 4.2 Attack Lifecycle

Attacks against AI agents typically follow a four-stage lifecycle: (1) injection—attacker embeds malicious instructions in content the agent processes; (2) hijacking—the agent interprets malicious content as legitimate instructions; (3) execution—the agent invokes tools with attacker-controlled parameters; (4) impact—actions produce irreversible effects. OATS intervenes at two points: between stages 2 and 3 (the Gate blocks actions that violate policy), and before stage 3 begins (tool contracts constrain what parameters the agent can express).

Intent drift follows a different lifecycle without explicit injection. The agent's reasoning gradually diverges through plausible-seeming steps. OATS detects this through context accumulation and semantic distance tracking regardless of cause.

4.3 4.3 Trust Assumptions

Trusted: The OATS runtime (ORGA loop, policy engine, tool contract executor, journal, identity verifier); cryptographic primitives and key management; the policy store and authoring process; the underlying infrastructure (OS, network, hardware); the compiler and type system.

Untrusted: The AI model and its outputs; the orchestration layer; user inputs and prompts; tool outputs and retrieved data; external documents, emails, web content, and multi-modal inputs; agent memory and conversation history; tool contracts from unverified publishers.

Partially trusted: Tool implementations (OATS constrains invocation but cannot prevent internal bugs); human approvers (OATS routes step-up authorization but cannot prevent social engineering); verified tool contracts (verified as untampered, but the tool may have vulnerabilities).

4.4 4.4 Out of Scope

OATS addresses runtime action security. The following threats require complementary controls: model training data poisoning or weight manipulation (pre-deployment ML security); denial of service against the OATS runtime (infrastructure availability); physical or infrastructure-level attacks (physical security); social engineering of human approvers (security awareness training); code-level vulnerabilities within tool implementations (application security testing); memory storage security (separate storage controls). OATS is one layer in a defense-in-depth strategy.

5 5. Core Architecture: The ORGA Loop

The ORGA (Observe-Reason-Gate-Act) loop is the core execution engine for OATS-compliant runtimes. It drives a multi-turn cycle between an LLM, a policy gate, and external tools through four mandatory phases.

5.1 5.1 Phase Definitions

Observe. Collect results from previous tool executions. Incorporate tool outputs, error messages, policy denial feedback, and environmental signals into the agent's context. This phase also integrates knowledge retrieval (RAG-enhanced context from vector-backed storage) when available.

Reason. The LLM processes accumulated context and produces proposed actions (tool calls or text responses). The LLM sees tool definitions derived from tool contracts but never sees raw invocation details. The LLM's output is a structured proposal, not an executable action.

Gate. The policy engine evaluates each proposed action. This phase operates entirely outside LLM influence. The Gate receives the proposed action, the accumulated session context, and the agent's identity, and evaluates them against organizational policy. The Gate produces one of five decisions: Allow, Deny, Modify, Step-Up, or

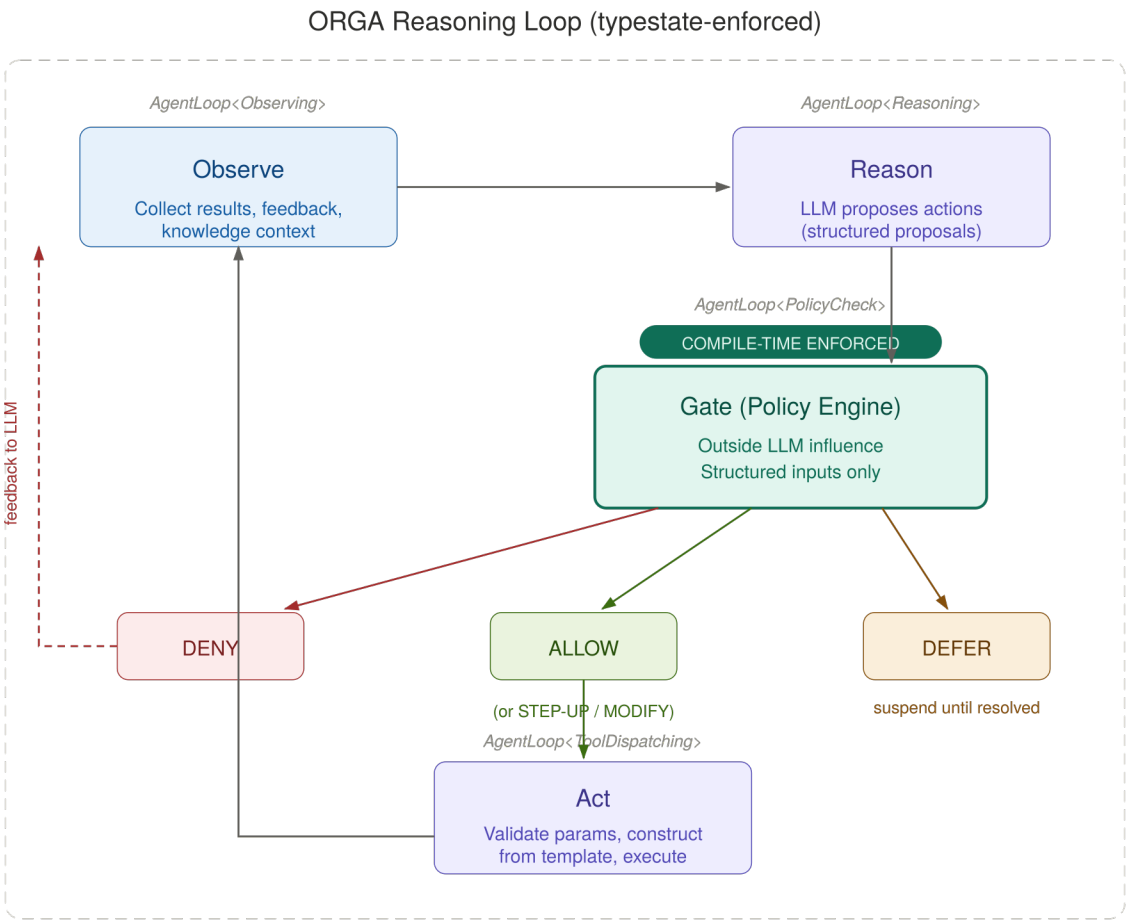


Figure 2: Figure 1. ORGA loop with typestate-enforced phase transitions. The Gate phase is mandatory and structurally independent of the LLM. Denied actions produce feedback observations; approved actions proceed to tool execution.

Defer. Denial reasons are recorded in the audit journal and fed back to the LLM as observations in the next Observe phase.

Act. Approved actions are dispatched to tool executors. The tool contract executor validates parameters against the contract's type system, constructs the invocation from the contract's template, executes with timeout enforcement, captures output in a structured evidence envelope, and records the execution in the audit journal.

5.2 5.2 Typestate Enforcement

Phase transitions MUST be enforced at compile time. Each phase is a distinct type. The loop state machine can only call methods appropriate to its current phase:

$$\text{AgentLoop}\langle\text{Reasoning}\rangle \xrightarrow{produce_output()} \text{AgentLoop}\langle\text{PolicyCheck}\rangle$$

$$\text{AgentLoop}\langle\text{PolicyCheck}\rangle \xrightarrow{check_policy()} \text{AgentLoop}\langle\text{ToolDispatching}\rangle$$

$$\text{AgentLoop}\langle\text{ToolDispatching}\rangle \xrightarrow{dispatch()} \text{AgentLoop}\langle\text{Observing}\rangle$$

$$\text{AgentLoop}\langle\text{Observing}\rangle \xrightarrow{observe()} \text{AgentLoop}\langle\text{Reasoning}\rangle \mid \text{LoopResult}$$

The following are **compile-time errors**, not runtime bugs:

- Transitioning from Reasoning to ToolDispatching (skipping policy check)
- Transitioning from PolicyCheck to Observing (dispatching without execution)
- Transitioning from Reasoning to Observing (skipping both Gate and Act)

Implementations in languages without native typestate support MUST provide equivalent enforcement through runtime checks with 100% path coverage testing and documented verification that all tool dispatch paths pass through the Gate. Such implementations SHOULD acknowledge that runtime enforcement provides weaker assurance than compile-time enforcement and document the residual risk.

5.3 5.3 Dynamic Branching and Termination

The only dynamic branch in the ORGA loop is after Observe: the loop either continues (returning to Reason) or completes (producing a final result). This is a standard pattern match on a concrete type, not dynamic dispatch. All other transitions are strictly linear.

The loop terminates when the LLM produces a final text response, iteration limits are reached, token or time budgets are exhausted, or a circuit breaker trips.

5.4 5.4 Policy Denial Feedback

When the Gate denies an action, the denial reason MUST be fed back to the LLM as an observation. This allows the LLM to adjust its approach. The Gate evaluates each subsequent proposal independently; denials are not negotiable.

5.5 5.5 Scope of Assurance

Typestate enforcement provides a specific, bounded property. To prevent overinterpretation, we state exactly what is and is not covered.

What typestate enforcement covers. Within the ORGA loop, the type system enforces that every transition from action proposal to tool dispatch passes through the Gate phase. In a Rust implementation, this is a compile-time property: any code path that attempts `AgentLoop<Reasoning>` → `AgentLoop<ToolDispatching>` without consuming an intermediate `AgentLoop<PolicyCheck>` is rejected by the compiler. A proof sketch follows.

Proof sketch. Let R, P, D, O denote the Reasoning, PolicyCheck, ToolDispatching, and Observing phases. Each phase is a distinct zero-sized type. The only method consuming R produces P; the only method consuming P

produces D; the only method consuming D produces O; and O produces either R (continue) or a terminal value (complete). Because each method takes `self` by value (consuming the prior state), no valid Rust program can hold two phase values simultaneously or skip a phase. The compiler's ownership and move semantics enforce this without runtime checks. This argument depends on the type signatures being correctly declared; it does not require trust in runtime behavior.

What typestate enforcement does not cover. The property applies only to code paths mediated by the `AgentLoop` runner. It does not provide whole-program non-bypass assurance. Specifically: (a) agent code that invokes tools through a separate code path not mediated by the ORGA runner is unconstrained by the typestate; (b) plugins, FFI calls, or dynamically loaded code may bypass the loop; (c) network-level tool invocations from within the sandbox are not mediated by the type system. These residual risks are addressed by sandboxing (Section 10) and network isolation as defense-in-depth layers, not by the typestate itself. An OATS-compliant deployment SHOULD combine typestate enforcement with at least one complementary isolation mechanism.

5.6 5.6 Gate Independence

The Gate phase is designed to operate outside LLM influence. This section specifies what "outside LLM influence" means concretely and what an implementation must demonstrate.

Structural requirements. The Gate MUST receive input as a typed, serialized data structure (e.g., a Rust struct, a JSON object conforming to a fixed schema) containing: tool name, operation, validated parameters, agent identity, and accumulated session context. The Gate MUST NOT receive natural language strings, LLM reasoning traces, or any content that requires language interpretation. The Gate MUST NOT share mutable memory, mutable references, or writable state with the LLM inference component. The Gate MUST NOT expose a callback, hook, or API that the LLM can invoke to modify Gate behavior during evaluation.

Implementation patterns. Conformant implementations may achieve Gate independence through any of the following mechanisms, listed in decreasing order of isolation strength: (a) separate process with IPC serialization boundary; (b) separate thread with immutable message passing and no shared mutable state; (c) synchronous function call with typed struct input, no closures capturing LLM state, and no interior mutability accessible from the LLM component. Pattern (a) provides the strongest isolation. Pattern (c) is acceptable when the implementation can demonstrate (via code review or static analysis) that no shared mutable path exists.

Verification. Conformance requirement C6 (Section 12) defines the verification procedure: inspect the Gate implementation, verify inputs are typed structs, verify no shared mutable references, verify no dynamic code paths parameterized by LLM output.

6 6. Tool Contract Layer

6.1 6.1 The Allow-List Principle

OATS inverts the conventional sandbox model:

6.2 6.2 Contract Requirements

A tool contract κ MUST define:

1. **Typed parameters.** Each parameter has a declared type from $\mathcal{T}$ with validation constraints. All string-based types MUST reject shell metacharacters (`;|&$\(){}[]<>¡`) by default.
2. **Invocation mechanism.** Command template, HTTP request template, protocol server address, or interactive session definition. The LLM never constructs invocation details.
3. **Output schema.** Expected structure of tool output. The executor validates parsed output before returning results to the agent.
4. **Policy metadata.** Policy resource and action declarations enabling authorization without parsing tool-specific details.
5. **Risk tier.** Risk classification (low, medium, high, critical) informing default policy generation and step-up thresholds.

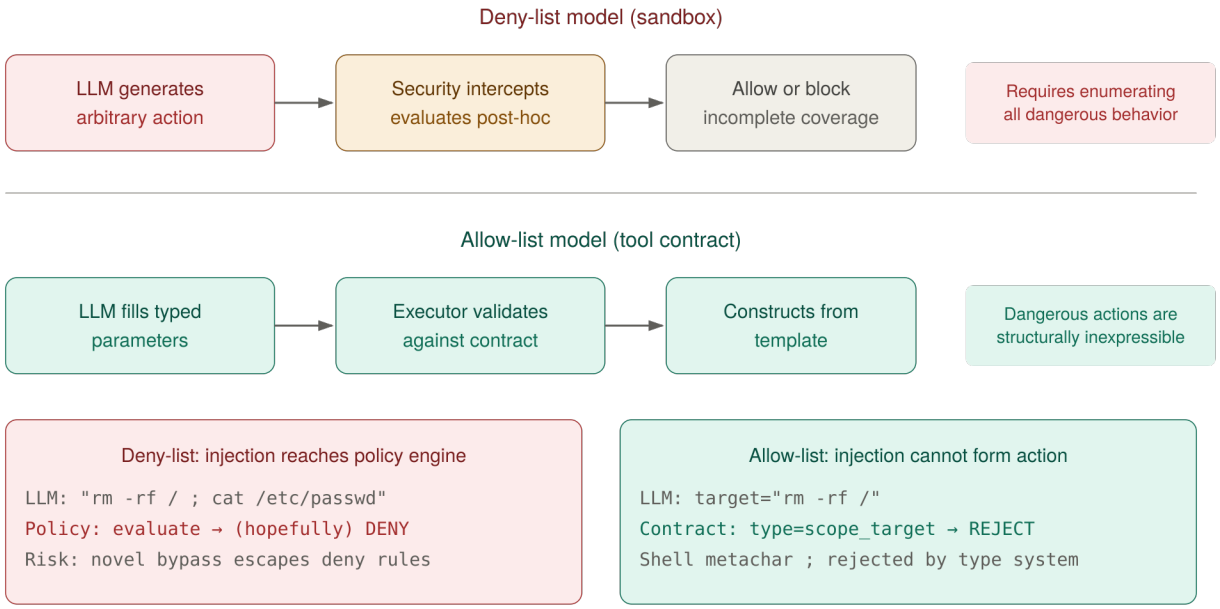


Figure 3: Figure 2. Deny-list vs. allow-list enforcement models. In the allow-list model, the space of expressible actions is constrained by the tool contract before any policy evaluation occurs.

6.3 6.3 Execution Modes

Tool contracts SHOULD support three execution modes sharing a common governance layer:

Table 3: **Table 2.** Tool contract execution modes.

Mode	Description	Governance
Oneshot	Single invocation, return results	Per-invocation Gate evaluation
Session	Running process (PTY), per-interaction validation	Per-interaction Gate evaluation
Browser	Governed browser (CDP/Playwright), scoped navigation	Per-action Gate evaluation

6.4 6.4 Contract Integrity

Tool contracts MUST support cryptographic integrity verification. Signatures MUST cover the entire contract—parameters, validation rules, invocation templates, output schemas, and scope constraints. A contract failing verification MUST be rejected.

6.5 6.5 Schema Generation

Tool contracts SHOULD support automatic generation of protocol-compatible schemas (e.g., MCP `inputSchema` and `outputSchema`) from the contract definition. The LLM understands tool capabilities through generated schemas without the contract format being exposed.

7 7. Identity Layer

7.1 7.1 The Identity Problem

When AI agents interact with tools, services, and other agents, identity is typically self-asserted. An agent claims to be "Scout v2 from Tarnover LLC" with no cryptographic proof. A tool claims to offer a specific interface with

no integrity verification. Self-asserted identity provides no security guarantee.

OATS specifies a two-layer cryptographic identity stack:

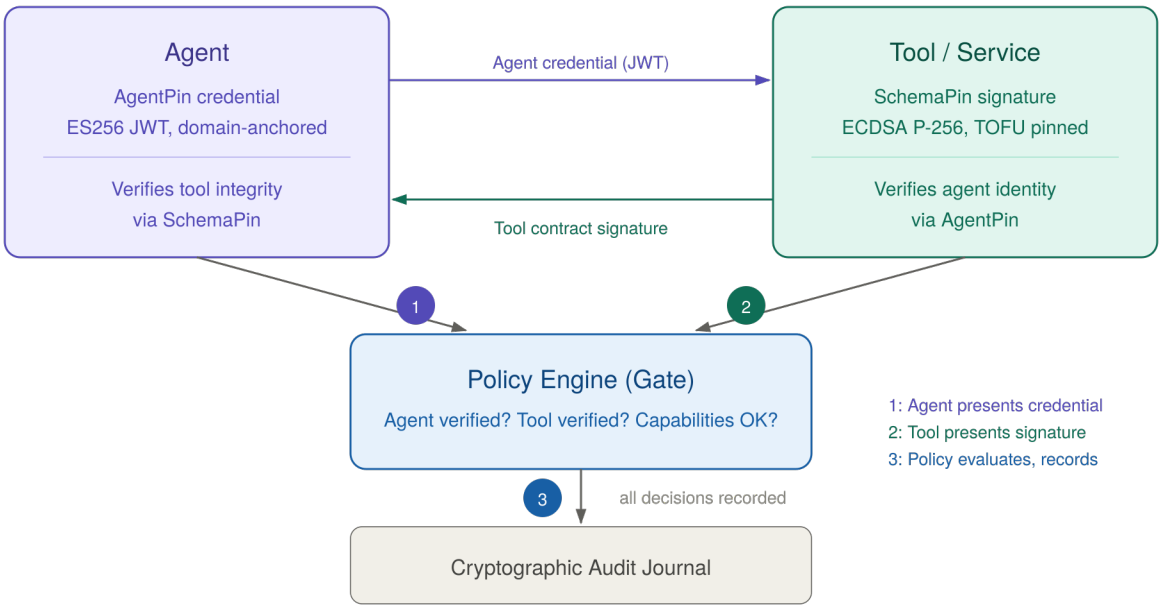


Figure 4: Figure 3. Bidirectional trust model. The agent verifies tool integrity (SchemaPin), the tool verifies agent identity (AgentPin), and the policy engine evaluates authorization. All three steps are recorded in the cryptographic audit journal.

7.2 7.2 Tool Integrity Verification

An OATS-compliant runtime MUST support cryptographic verification of tool contracts. The protocol provides:

- **Domain-anchored discovery.** Publishers host public keys at `/.well-known/` URIs (RFC 8615). No centralized registry.
- **Signature verification.** Contracts signed with ECDSA P-256 (or equivalent). Runtime verifies before registration.
- **TOFU key pinning.** On first encounter, the runtime pins the publisher's key. Subsequent key changes require explicit trust decisions.
- **Revocation support.** Publishers can revoke keys and schemas. Runtime checks revocation status.

7.3 7.3 Agent Identity Verification

An OATS-compliant runtime SHOULD support cryptographic agent identity verification:

- **Domain-anchored identity.** Organizations publish verifiable agent identity documents at `/.well-known/agent-identity.json`.
- **Short-lived credentials.** ES256 JWTs declaring identity, capabilities, and delegation chain.
- **Multi-step verification.** Signature verification, domain binding, capability validation, revocation checking, TOFU key pinning.
- **Delegation chains.** Maker-deployer delegation where the software builder and instance deployer are independently verifiable.
- **Capability scoping.** Credentials declare specific capabilities (e.g., `read:data`, `write:reports`).

7.4 7.4 Bidirectional Trust

The combined verification flow:

1. Agent's runtime verifies tool contract integrity (tool has not been tampered with).
2. Tool's runtime verifies agent identity (agent is authorized to act).
3. Policy engine evaluates whether the verified agent's capabilities authorize use of the verified tool.
4. Both verifications and the policy decision are recorded in the audit journal.

8 8. Policy Enforcement Layer

8.1 8.1 Policy Engine Requirements

An OATS-compliant policy engine evaluates the tuple (a, C, I) and produces an authorization decision. The engine:

- MUST support five decisions: Allow, Deny, Modify, Step-Up, Defer.
- MUST evaluate both static policy and accumulated session context.
- MUST operate outside LLM influence: structured inputs, structured decisions, no natural language processing, no shared mutable state with the LLM.
- SHOULD support a formally verifiable policy language (Cedar [21], OPA [20], or equivalent).
- MUST default to deny.

8.2 8.2 Action Classification

OATS classifies actions into five categories based on how they should be evaluated:

Table 4: **Table 3.** Action classification framework. The "structurally forbidden" category exists only in systems with allow-list tool contracts.

Category	How Identified	Evaluation	Decision
Structurally forbidden	Cannot be expressed via tool contract	None needed	N/A (inexpressible)
Policy-forbidden	Static policy match	Static policy only	DENY
Context-dependent deny	Policy allows, context misaligns	Static + context	DENY
Context-dependent allow	Policy denies, context aligns	Static + context	STEP-UP / ALLOW
Context-dependent defer	Insufficient/conflicting context	Indeterminate	DEFER

8.3 8.3 Context Accumulation

The runtime MUST accumulate session context as an append-only, hash-chained log:

- **Original request.** The user's initial instruction establishing intent.
- **Action history.** Sequence of actions proposed, approved, denied, deferred, and executed.
- **Data classification.** Sensitivity of information accessed. Default: highest configured level when unknown.
- **Tool outputs.** Results from previous actions.
- **Semantic distance.** Drift from original request (see Section 8.4).
- **Identity context.** Verified identities of agent, user, and tools.

8.4 8.4 Semantic Distance Tracking

The runtime SHOULD compute semantic distance between actions and stated intent:

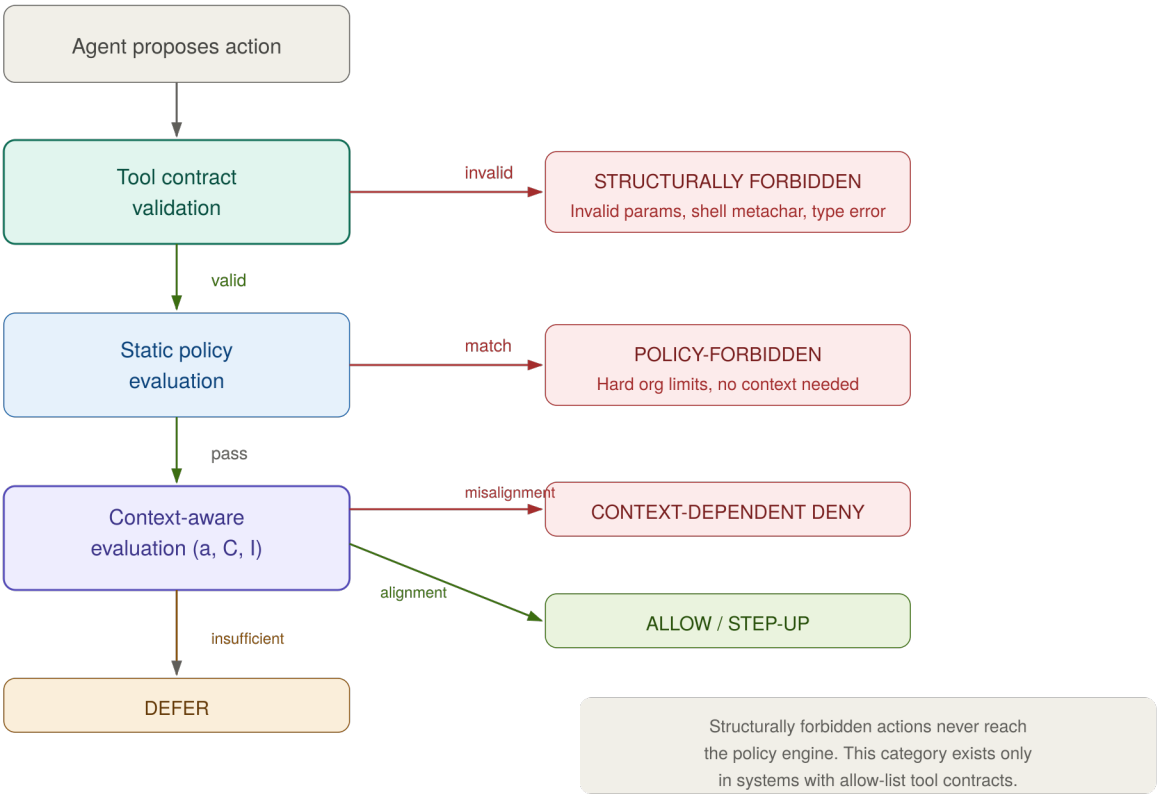


Figure 5: Figure 4. Action classification pipeline. Tool contract validation eliminates structurally forbidden actions before the policy engine is reached. Context-aware evaluation handles the remaining categories.

$$d(r_0, a_n) = 1 - \text{cosine}(\text{embed}(r_0), \text{embed}(a_n))$$

where r_0 is the original request and a_n is the current action. Cumulative drift SHOULD be tracked across sequences. Thresholds are deployment-specific and SHOULD be calibrated empirically.

Semantic distance is a risk signal, not a primary authorization primitive. The hard authorization layer in OATS is the deterministic policy engine (Cedar, OPA, or equivalent) evaluating structured inputs. Semantic distance provides an advisory signal that the policy engine may consume as one input to step-up or defer decisions, but OATS-compliant runtimes SHOULD NOT rely on semantic distance as the sole basis for irreversible denial. This separation ensures that the Gate's authorization decisions remain deterministic and reproducible even when the drift signal is heuristic. The limitations of embedding-based distance tracking are discussed in Section 15.

8.5 8.5 Step-Up Authorization and Deferral

For step-up authorization: execution MUST block until approval; full context MUST be available to approvers; configurable timeouts MUST be enforced (deny on timeout); decisions MUST be recorded in the journal.

For deferral: deferred actions MUST remain paused without effects; the runtime MUST track deferred actions and maintain execution order; cascading deferrals MUST be bounded (deny when limit exceeded); deny on timeout MUST be the default; both deferral and resolution MUST be recorded.

9 9. Audit Layer

9.1 9.1 Journal Architecture

An OATS-compliant runtime MUST maintain a cryptographic audit journal. The journal is the authoritative record of what happened, when, why, and by whose authority.

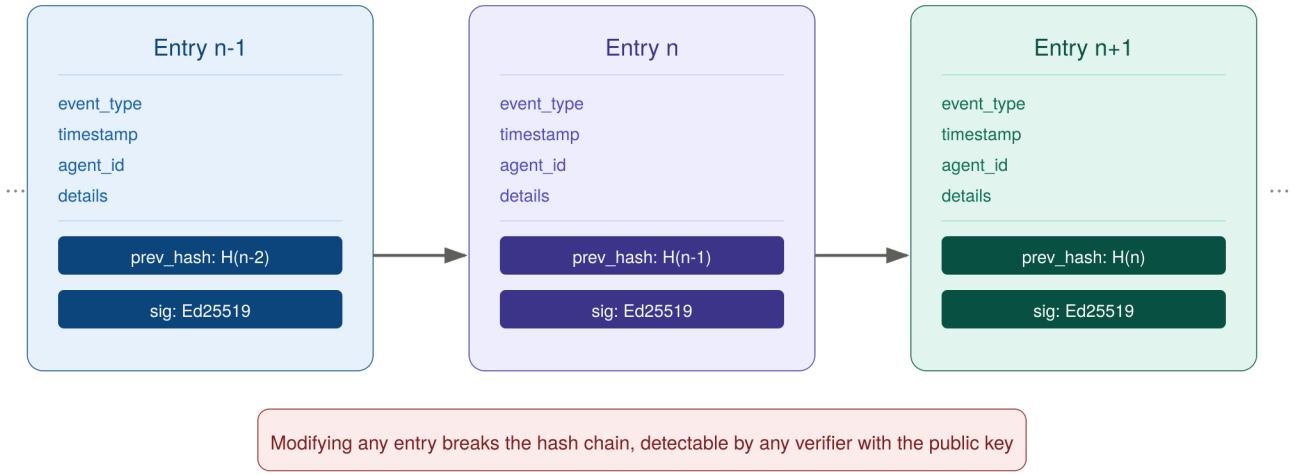


Figure 6: Figure 5. Hash-chained audit journal. Each entry includes the cryptographic hash of the previous entry and an Ed25519 signature, forming a tamper-evident chain verifiable offline.

9.2 9.2 Event Types

Table 5: **Table 4.** Required journal event types.

Event	When	Content
LoopStarted	Loop begins	Configuration, agent identity, original request
ReasoningComplete	After LLM, before Gate	Proposed actions, token usage
PolicyEvaluated	After Gate decision	Decisions, matching policies, reasons
ToolsDispatched	After execution	Tools, parameters, duration, evidence hashes
ObservationsCollected	After results	Observation count, context size
LoopTerminated	Loop ends	Reason, iterations, total usage, duration
RecoveryTriggered	On failure	Strategy, error context

9.3 9.3 Cryptographic Properties

Each entry MUST include an Ed25519 signature (or ECDSA P-256) covering the canonical serialization, a hash chain link to the previous entry, and a cryptographic timestamp. Entries MUST be verifiable offline.

9.4 9.4 Evidence Envelopes

Tool executions MUST produce structured evidence envelopes: tool name and version, validated parameters, constructed invocation, duration and exit status, output hash (SHA-256), authorizing policy decision, and identity at execution time.

9.5 9.5 Compliance Properties

The journal provides infrastructure that can contribute to regulatory compliance, though OATS alone is not sufficient for any regulatory framework. Specifically: the journal can serve as a component of HIPAA audit trails (recording health data access with identity and authorization); SOC2 evidence collection (recording policy enforcement decisions); SOX audit trail requirements (recording attributable financial system actions); and GDPR accountability mechanisms (recording data access patterns). In each case, the journal addresses the technical recording requirement but does not address the organizational, procedural, or legal requirements of the applicable regulation.

10 10. Sandboxing and Isolation

10.1 10.1 Multi-Tier Sandboxing

An OATS-compliant runtime SHOULD support multiple sandboxing tiers:

Table 6: **Table 5.** Sandboxing tiers.

Tier	Mechanism	Isolation Level	Overhead
1	Container (Docker)	Process, filesystem, network	Low
2	User-space kernel (gVisor)	Syscall filtering	Medium
3	MicroVM (Firecracker)	Hardware-level	Medium-High

10.2 10.2 Resource Limits

Agent execution MUST support configurable limits: token budget, time budget, iteration budget, tool call budget, network restrictions, and filesystem restrictions.

10.3 10.3 Circuit Breakers

Tool executions SHOULD be protected by circuit breakers. When a tool fails repeatedly, the circuit trips and subsequent calls are rejected until reset, preventing cascading failures and runaway retry loops.

11 11. Inter-Agent Communication

11.1 11.1 Communication Governance

All inter-agent messages MUST pass through a communication policy gate evaluating authorization rules on communication primitives (ask, delegate, send, parallel, race).

11.2 11.2 Message Security

Messages MUST be cryptographically signed (Ed25519), encrypted (AES-256-GCM), and attributed to verified agent identities.

11.3 11.3 Delegation Constraints

Delegation chains MUST be bounded: maximum depth (configurable), capability narrowing (delegated agents cannot exceed delegating agent's capabilities), and blast-radius containment.

11.4 11.4 Cross-Agent Context

Session context (original request, prior actions, data classifications) SHOULD be propagated to downstream agents, enabling their Gates to evaluate against original intent.

12 12. Conformance Requirements

The requirement language follows RFC 2119: MUST indicates absolute requirements; SHOULD indicates recommendations that may be omitted with documented justification.

12.1 12.1 Conformance Levels

OATS Core (all MUST requirements, C1–C7): Baseline zero-trust agent execution.

OATS Extended (all MUST and SHOULD requirements, C1–C7 + E1–E8): Comprehensive zero-trust with identity, sandboxing, and advanced policy.

12.2 12.2 Core Requirements (MUST)

C1: ORGA Loop Enforcement. The runtime MUST implement the four-phase ORGA loop. The Gate MUST execute before every tool dispatch. In compiled languages, phase transitions MUST be enforced at compile time via typestates. In interpreted languages, equivalent enforcement MUST be provided and documented, with acknowledgment of residual risk per Section 5.5.

Verification: Attempt to construct a code path from Reason to Act bypassing Gate. In a typestate implementation, this MUST be a compile error. In runtime-enforced implementations, this MUST be caught by a verified test suite with 100% tool dispatch path coverage.

C2: Tool Contract Support. The runtime MUST support declarative tool contracts with typed parameter validation. The LLM MUST NOT generate raw tool invocations. All invocations MUST be constructed from validated parameters and contract-defined templates.

Verification: Submit parameters containing shell metacharacters (`;`, `|`, `&`, `` ` ``). Verify rejection. Submit parameters outside declared type constraints. Verify rejection. Verify the LLM never receives raw invocation strings in any code path.

C3: Policy Evaluation. The runtime MUST evaluate actions against policy before execution. The policy engine MUST operate outside LLM influence. MUST support Allow, Deny, Modify, Step-Up, and Defer decisions. Default MUST be deny.

Verification: Configure a DENY policy. Submit a matching action. Verify no effects on the target system. Verify denial recorded in journal with matching policy and reason. Repeat for each decision type.

C4: Context Accumulation. The runtime MUST accumulate session context across actions. Context MUST include original request (when available), action history, and data classification.

Verification: Execute a sequence of three or more actions. Verify the policy engine receives accumulated context for each subsequent action. Verify context includes prior actions and their data classifications.

C5: Cryptographic Audit Journal. The runtime MUST maintain a hash-chained, cryptographically signed audit journal recording all ORGA loop events (Table 4). Entries MUST be verifiable offline.

Verification: Generate journal entries for allowed, denied, deferred, and step-up actions. Verify all fields present, signatures valid, and hash chain intact. Tamper with one entry and verify that chain verification detects the modification.

C6: Gate Independence. The Gate MUST operate on structured inputs only. It MUST NOT process natural language, share mutable state with the LLM, or be influenced by the LLM's reasoning.

Verification: Inspect Gate implementation. Verify inputs are typed structs (tool name, operation, parameters, identity, context), not natural language strings. Verify no shared mutable references between the Gate and the LLM inference component. Verify no dynamic code paths within the Gate that are parameterized by LLM output.

C7: Evidence Envelopes. Tool executions MUST produce structured evidence envelopes with output hashes, execution metadata, and identity binding.

Verification: Execute a tool. Verify the envelope contains tool name, version, validated parameters, constructed invocation, duration, exit status, SHA-256 output hash, authorizing policy decision, and agent/user identity.

12.3 12.3 Extended Requirements (SHOULD)

E1: Tool Integrity Verification. Verify tool contract signatures using domain-anchored cryptographic verification with TOFU key pinning.

E2: Agent Identity Verification. Verify agent identity using domain-anchored ES256 credentials with delegation chain support.

E3: Semantic Distance Tracking. Compute and track semantic distance between actions and stated intent using embedding similarity.

E4: Multi-Tier Sandboxing. Support configurable sandboxing tiers (container, kernel-level, microVM).

E5: Inter-Agent Communication Governance. Enforce authorization policies on inter-agent communication with signed and encrypted messages.

E6: Telemetry Export. Export structured telemetry (OCSF, CEF, or documented custom schemas) with real-time streaming.

E7: Formally Verifiable Policies. Use a policy language enabling static analysis of correctness (Cedar, OPA, or equivalent).

E8: Least-Privilege Credential Scoping. Support just-in-time credential issuance with operation-specific scoping and logged usage.

13 13. Implementation Architectures

OATS does not mandate a specific implementation architecture. The specification intentionally separates conformance properties from implementation details and includes four deployment patterns to reduce dependence on any single implementation. The current specification is informed by one reference implementation (Symbiont);

independent conformance testing across additional implementations is needed to validate that the specification is sufficiently general. Table 6 compares the four reference architectures.

Table 7: **Table 6.** Implementation architecture comparison.

Property	Self-Hosted Runtime	Plugin/Extension	Gateway	Vendor Integration
You control	Everything	Agent code	Network	Policy only
Enforcement	ORGA typestate	Dual-layer	Network proxy	Vendor hooks
Bypass resistance	Very high	High	High	Vendor-dependent
Context richness	Full	Full (inner)	Limited	Vendor-dependent
Tool contracts	Full	Full (outer)	Partial	Vendor-dependent
Identity	Full	Full (outer)	Partial	Vendor-dependent
OATS-conformant	Yes	Yes	Partial	If hooks sufficient

13.1 13.1 Self-Hosted Runtime

The full OATS stack deployed as a single runtime. Provides the strongest available enforcement properties: compile-time phase enforcement, full context visibility, cryptographic identity, and multi-tier isolation. Natural home in systems-level languages with rich type systems.

13.2 13.2 Plugin/Extension Model

For agents running inside third-party platforms. An inner layer (plugin) provides awareness; an outer layer (OATS runtime wrapping the platform via CLI executor or container) provides enforcement. Because the outer ORGA Gate mediates all tool invocations at the process boundary, the inner platform cannot bypass it through normal operation, though side-channel bypasses (e.g., direct network calls from within the sandbox) require complementary network-level controls.

13.3 13.3 Gateway Architecture

For protocol-based tool invocations (MCP, REST). An OATS-compliant gateway intercepts traffic between agents and tools, implementing Gate, context accumulation, and journaling. Provides enforcement without agent modification, at the cost of reduced context visibility.

13.4 13.4 Vendor Integration

For SaaS agents where organizations control no infrastructure. Requires vendor-provided synchronous pre-execution hooks, decision enforcement, context availability, and receipt export. OATS provides the specification for vendor evaluation and contracts.

14 14. Evaluation Framework

The claims made in this specification are architectural: OATS is designed to provide certain security properties through structural enforcement. Converting these design-level claims into empirical evidence requires a systematic evaluation methodology. This section defines the evaluation framework; results will be published separately as they become available.

14.1 14.1 Attack Suite Methodology

To measure whether OATS reduces attack success rates, we define a comparative evaluation against three baselines:

Configuration	Description
Baseline A	No policy enforcement. Agent invokes tools directly.
Baseline B	Deny-list policy. Agent actions intercepted and evaluated against forbidden-action rules.
Baseline C	Prompt-guardrail only. Input/output filtering at the LLM layer, no action-level enforcement.
OATS	Full stack: ORGA loop, tool contracts, policy engine, identity verification, journal.

The attack suite combines existing benchmarks with custom scenarios:

- **AgentDojo** [7]: Dynamic environment evaluating prompt injection attacks and defenses across realistic agent tasks. We measure attack success rate (percentage of injections that produce the attacker's intended tool invocation) and task utility (percentage of legitimate tasks completed successfully).
- **Custom injection suite**: 200+ prompt injection variants (direct, indirect via documents, indirect via tool outputs, multi-turn) targeting tool invocations across 10 tool types. Includes attacks specifically targeting allow-list bypass (e.g., attempting to construct shell commands through parameter concatenation).
- **Compositional exfiltration scenarios**: 50 multi-step sequences where individual actions are policy-compliant but the composition constitutes a violation (e.g., read sensitive data then email externally).

For each configuration, we report: attack success rate, task completion rate (legitimate tasks), false positive rate (legitimate actions blocked), and false negative rate (attack actions allowed).

Fairness methodology. To prevent evaluation bias toward the OATS architecture, the following constraints apply: (a) all configurations use the same task suite, tool set, and underlying LLM; (b) credential scopes are identical across configurations; (c) deny-list policies in Baseline B are tuned using a held-out calibration set, not the test set; (d) all policy thresholds and drift thresholds are fixed before test execution and not adjusted post-hoc; (e) all failures, false negatives, and bypass successes are reported, not only aggregate metrics. These constraints are designed to ensure that measured differences reflect architectural properties rather than configuration advantages.

14.2 14.2 Performance Overhead

Runtime enforcement introduces latency. We define benchmarks for each enforcement layer:

Table 9: **Table 7.** Performance benchmarks for OATS enforcement components. Targets are derived from operational experience and represent design goals, not validated measurements.

Component	Measurement	Target
Static policy evaluation	Per-action latency (p50, p95, p99)	<1ms
Context-dependent evaluation	Per-action latency with accumulated context (10, 50, 100 prior actions)	<5ms
Semantic distance computation	Per-action embedding + cosine similarity	<50ms
Tool contract validation	Parameter type checking + sanitization	<0.5ms
Ed25519 journal signing	Per-entry signature generation	<0.1ms
Hash chain verification	Per-entry hash computation + chain link	<0.05ms
Full ORGA Gate pipeline	End-to-end from action proposal to allow/deny decision	<10ms (p99)
Evidence envelope generation	SHA-256 output hash + envelope construction	<1ms

Benchmarks should be run under realistic agent workloads (10–1000 actions per session) on commodity hardware, with context accumulation growing across the session. Context scalability (memory and evaluation time as a function of session length) should be reported separately.

14.3 14.3 False Positive/Negative Analysis

Security enforcement is useful only if it blocks attacks without blocking legitimate work. We define a methodology for measuring this tradeoff:

- **Task suite**: Run BFCL (Berkeley Function Calling Leaderboard) and SWE-bench Lite task sets through an OATS-compliant runtime. These provide realistic agent workloads with known-correct tool invocation sequences.
- **False positive rate**: Percentage of legitimate tool invocations that the Gate denies or defers. Measured across policy strictness levels (permissive, moderate, strict).
- **False negative rate**: Percentage of attack-injected tool invocations that the Gate allows. Measured using the attack suite from Section 14.1.
- **ROC analysis**: Plot false positive rate vs. true positive rate (attack blocked) across policy threshold configurations. Report area under curve (AUC) for each OATS layer (contracts only, contracts + static policy, contracts + static + context-dependent, full stack).

14.4 14.4 Bypass Resistance Testing

Each enforcement mechanism has a theoretical bypass boundary. We define tests that probe these boundaries:

- **Typestate circumvention**: Attempt to construct Rust code paths from Reasoning to ToolDispatching that skip PolicyCheck. The compiler should reject all such paths. Report: number of bypass attempts, number of compile errors (should be 100%).
- **Tool contract metacharacter injection**: Submit parameters containing shell metacharacters, SQL injection patterns, path traversal sequences, and Unicode homoglyphs to each parameter type. Report: rejection rate per type, any bypasses discovered.
- **Gate influence probing**: Craft LLM outputs designed to influence Gate evaluation (e.g., embedding policy-override instructions in proposed action parameters, attempting to modify shared state). Report: whether any crafted output changes Gate behavior.
- **Sandbox escape**: Attempt network calls, filesystem access, and process spawning from within each sandbox tier. Report: blocked rate per tier, any escapes discovered.
- **Identity spoofing**: Attempt to present forged AgentPin credentials, tampered SchemaPin signatures, and replayed JWTs. Report: verification rejection rate.

14.5 14.5 Ablation Study

To measure the marginal contribution of each OATS layer, we define an ablation removing one layer at a time:

Table 10: **Table 8.** Ablation study configurations.

Configuration	Layers Active	Expected Impact
Full OATS	All 5 layers	Baseline (best security, highest overhead)
No contracts	ORGA + policy + identity + journal	Allows arbitrary action formulation; tests policy-only defense
No identity	ORGA + contracts + policy + journal	Removes mutual auth; tests unverified tool/agent interactions
No context	ORGA + contracts + static policy + journal	Removes context-dependent classifications; tests static-only policy
No journal	ORGA + contracts + policy + identity	Removes audit trail; tests enforcement without forensics
ORGA only	Loop enforcement, permissive policy	Tests whether phase ordering alone provides security value

For each configuration, report attack success rate and task completion rate from the suite defined in Section 14.1. The difference between configurations quantifies the security contribution of each layer.

14.6 14.6 Case Studies

Three detailed scenarios that exercise multiple OATS layers simultaneously:

Scenario 1: Multi-step data exfiltration. An agent is tasked with summarizing Q3 sales for internal leadership. A prompt injection in a retrieved document instructs the agent to email customer PII to an external address. The case study traces the scenario through each ORGA phase, showing: (a) the tool contract constrains the email tool's recipient parameter, (b) context accumulation tracks that PII-classified data was accessed, (c) the context-dependent deny classification blocks the external email, and (d) the journal records the full sequence for forensic reconstruction.

Scenario 2: Tool supply chain attack. An attacker modifies a tool contract to widen parameter validation (e.g., removing scope restrictions on a network scanner). The case study shows: (a) SchemaPin signature verification detects the contract modification, (b) the runtime rejects the tampered contract, and (c) the journal records the verification failure. A variant tests what happens when the attacker also compromises the publisher's signing key (requiring TOFU pin violation detection).

Scenario 3: Intent drift across a long session. An agent is asked to prepare for a client meeting. Over 15 actions, the agent's scope gradually expands from CRM queries to accessing confidential strategy documents. The case study shows: (a) semantic distance increases monotonically across the action sequence, (b) the configured drift threshold triggers a step-up authorization at action 11, and (c) the human approver receives full context including the drift trajectory.

15 15. Limitations

This section identifies known limitations of the OATS specification. These are not future research directions (Section 16) but inherent boundaries of the current architecture.

Typestate scope. Compile-time enforcement of the ORGA loop applies only to code paths within the typestate-governed loop. Agent code that bypasses the loop entirely—for example, by invoking tools through a separate code path not mediated by the ORGA runner—is not caught by the type system. Sandboxing and network isolation provide complementary enforcement but are defense-in-depth layers, not compile-time properties.

Tool contract coverage. The allow-list model governs only tools with declared contracts. Tools without contracts (legacy integrations, dynamically discovered MCP servers, ad-hoc API calls) are outside the allow-list boundary. An OATS-compliant runtime can deny uncontracted tool invocations by default, but this trades functionality for safety and may be impractical in environments with large numbers of tools.

Coverage-safety tradeoff. The allow-list model inherently restricts the agent's action space. Novel legitimate tool uses that were not anticipated when the contract was authored will be rejected until the contract is updated. This creates operational friction proportional to the rate of tool evolution. The severity of this tradeoff has not been quantified empirically.

Semantic distance limitations. Drift detection via embedding similarity depends on the quality of the embedding model and the meaningfulness of cosine distance in the action-intent space. Adversarial embeddings could subvert drift detection by producing actions that are semantically distant from the original intent but close in embedding space. The robustness of semantic distance tracking under adversarial conditions has not been evaluated.

Single reference implementation. The specification is informed by one reference implementation (Symbiont). Multi-implementation conformance testing—building independent OATS-compliant runtimes and verifying interoperability—has not been conducted. The specification may contain implicit assumptions derived from the reference implementation that create unnecessary barriers for alternative implementations.

Regulatory insufficiency. The audit journal provides technical infrastructure for compliance but is not sufficient for any regulatory framework on its own. HIPAA, SOC2, SOX, and GDPR each impose organizational, procedural, and legal requirements that OATS does not address. Claiming OATS compliance should not be conflated with claiming regulatory compliance.

Deferral latency. The DEFER authorization decision suspends action execution until resolution. In time-critical agent workflows (e.g., real-time trading, incident response), deferral latency may be unacceptable. The

specification does not provide guidance on latency-sensitive deferral policies beyond configurable timeouts.

Privacy in cross-agent context. Propagating session context across agent boundaries (Section 11.4) raises privacy and data sovereignty concerns. Context may contain sensitive information from the original user's request, and propagating it to downstream agents in different organizational domains may violate data handling agreements. The specification does not address context redaction or privacy-preserving context propagation.

Non-deterministic evaluation. Context-dependent action classification relies on the policy engine's evaluation of accumulated context. When the policy engine uses semantic similarity or ML-based classification, evaluation results may be non-deterministic across invocations. The specification requires deterministic policy engines (Cedar, OPA) but permits semantic distance as a SHOULD requirement, creating a tension between deterministic authorization and non-deterministic drift signals.

16 16. Research Directions

16.1 16.1 Typestate in Non-Rust Languages

OATS's compile-time enforcement property is most naturally expressed in languages with typestate support (Rust, Haskell, Scala). Providing equivalent enforcement in Python, JavaScript, and Go requires runtime checks with formal path coverage verification, or code generation from a verified specification. The degree of assurance loss when moving from compile-time to runtime enforcement is an open question.

16.2 16.2 Data Flow Through Context Windows

Data may be transformed, summarized, or paraphrased by the LLM before use in subsequent actions. Information-theoretic approaches (taint analysis, embedding watermarking) for tracking lineage through non-deterministic transformations are active research.

16.3 16.3 Multi-Agent Trust Coordination

Maintaining coherent trust chains across organizational boundaries in delegation requires distributed tracing standards, federated receipt verification, and cross-domain policy negotiation.

16.4 16.4 Formal Verification of the ORGA Loop

Typestate enforcement addresses phase ordering within the loop. Mechanized proofs of the entire system—policy engine correctness, context accumulator completeness, journal integrity, and the absence of bypass paths outside the loop—would provide substantially stronger assurance. Such proofs would also help bound the gap between specification-level properties and implementation-level behavior.

16.5 16.5 Approval Fatigue and Deferral Resolution

Balancing security against usability. ML-based approval recommendation, batch approval, and progressive autonomy (reduced approval requirements through demonstrated compliance) are active directions.

16.6 16.6 Vector Embedding Security

When semantic distance tracking uses embeddings, those embeddings become a security surface. Information-theoretic watermarking, steganographic attack detection, and quantization-robust integrity verification are needed.

17 17. Conclusion

OATS specifies what a zero-trust AI agent runtime should do to provide meaningful security properties for autonomous agent execution. The specification is grounded in three architectural convictions, each of which requires empirical validation to confirm that design-level properties translate to measurable security improvements:

Allow-list over deny-list. Constraining what actions can be expressed reduces the attack surface compared to intercepting arbitrary actions and deciding which to block. Tool contracts are designed to make dangerous actions structurally inexpressible within the contracted interface. The degree to which this reduces real-world attack success rates is an empirical question addressed by the evaluation framework in Section 14.

Compile-time over runtime enforcement. Enforcing policy evaluation through the type system provides stronger structural assurance than testing it at runtime, because bypass paths that violate the typestate are rejected by the compiler rather than discovered through testing. This property holds within the typestate-governed code; it does not protect against bypasses that circumvent the loop entirely.

Structural independence over trust assumptions. Architecturally isolating the Gate from LLM influence—by restricting it to structured inputs with no shared mutable state—reduces the risk that a compromised orchestration layer can influence policy evaluation. The strength of this isolation in practice depends on implementation quality and the completeness of the isolation boundary.

The architecture specified here is informed by approximately eight months of autonomous operation in a production runtime (Symbiont, by ThirdKey AI), including rebuilding a codebase using the runtime's own agent infrastructure after a catastrophic loss event. This operational experience has shaped the specification's requirements and identified practical challenges, but it does not constitute a controlled empirical evaluation. Section 14 outlines the evaluation methodology needed to substantiate the specification's claims.

By publishing this specification as an open standard, we aim to establish baseline requirements that enable comparable evaluation of runtime security approaches for autonomous agents. The goal is not to build OATS, but to define what an OATS-compliant system must do, enabling independent implementations to be measured against shared conformance criteria.

17.1 A. Future Directions for Adoption

For implementors: Build independent OATS-compliant runtimes across different language ecosystems. Multi-implementation conformance testing is the most important next step for validating the specification's generality.

For evaluators: Apply the evaluation framework in Section 14 to existing and new agent runtimes. Comparative results across architectures would substantially strengthen or refine the claims made in this specification.

For researchers: Address open challenges in Section 16, particularly typestate portability across language ecosystems, formal verification of runtime properties, and multi-agent trust coordination.

For the community: The specification is open and available at thirdkey.ai/oats. Feedback, critique, and competing proposals are welcome.

18 References

[1] Anthropic, "Model Context Protocol Specification," 2024. https://modelcontextprotocol.io

[2] L. Wang et al., "A Survey on Large Language Model based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.

[3] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," *ICLR*, 2023.

[4] Q. Wu et al., "Security of AI Agents," arXiv:2406.08689, 2024.

[5] Q. Ye et al., "ToolEmu: Identifying Risky Real-World Agent Failures with a Language Model Emulator," *ICLR*, 2024.

[6] H. Su et al., "A Survey on Autonomy-Induced Security Risks in Large Model-Based Agents," arXiv:2506.23844, 2025.

[7] E. Debenedetti et al., "AgentDojo: A Dynamic Environment to Evaluate Attacks and Defenses for LLM Agents," arXiv:2406.13352, 2024.

[8] Y. Ruan et al., "The Emerged Security and Privacy of LLM Agent: A Survey with Case Studies," arXiv:2407.19354, 2024.

[9] S. Perez et al., "Ignore This Title and HackAPrompt: Exposing Systemic Vulnerabilities of LLMs Through a Global Prompt Hacking Competition," *EMNLP*, 2023.

[10] Y. Liu et al., "Formalizing and Benchmarking Prompt Injection Attacks and Defenses," *USENIX Security*, 2024.

[11] K. Greshake et al., "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," *AISec Workshop at ACM CCS*, 2023.

[12] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins University, 2006.

[13] S. Gaire et al., "Systematization of Knowledge: Security and Safety in the Model Context Protocol Ecosystem," arXiv:2512.08290, 2025.

[14] H. Errico, "Autonomous Action Runtime Management (AARM): A System Specification for Securing AI-Driven Actions at Runtime," arXiv:2602.09433v1, 2026.

[15] A. Chuvakin, "Cloud CISO Perspectives: How Google secures AI Agents," *Google Cloud Blog*, June 2025.

[16] D. Reber, "The Agentic AI Security Scoping Matrix: A Framework for Securing Autonomous AI Systems," *AWS Security Blog*, November 2024.

[17] Microsoft, "Governance and security for AI agents across the organization," *Cloud Adoption Framework*, 2024.

[18] S. Raza et al., "TRiSM for Agentic AI: A Review of Trust, Risk, and Security Management in LLM-based Agentic Multi-Agent Systems," arXiv:2506.04133, 2025.

[19] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[20] Open Policy Agent, "OPA: Policy-based control for cloud native environments," 2024. https://www.openpolicyagent.org

[21] Amazon Web Services, "Cedar: A Language for Defining Permissions as Policies," 2023. https://www.cedarpolicy.com

[22] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications," 2024.

[23] National Institute of Standards and Technology, "AI Risk Management Framework (AI RMF 1.0)," 2023.

[24] J. Wanger, "AgentPin Technical Specification v0.2.0," ThirdKey AI, 2026. https://agentpin.org

[25] J. Wanger, "SchemaPin Protocol Specification," ThirdKey AI, 2025. https://schemapin.org

[26] J. Wanger, "ToolClad: Declarative Tool Interface Contracts for Agentic Runtimes v0.5.1," ThirdKey AI, 2026.

[27] J. Wanger, "Symbiont Runtime Architecture," ThirdKey AI, 2026. https://symbiont.dev

---

Open Agent Trust Stack (OATS): Zero-trust agent execution through structural enforcement.

ThirdKey AI — thirdkey.ai | **Symbiont** — symbiont.dev | **SchemaPin** — schemapin.org | **AgentPin** — agentpin.org