AOS Runtime

Executive Summary

What it is

AOS is a Python runtime that sits between AI agents and the systems they operate on. It enforces what agents are allowed to do, gates destructive actions on human approval, produces a tamper-evident audit trail of every action, and limits how much each session can spend.

It runs as a library inside an application or as a standalone server with an operator dashboard. PostgreSQL is the only required infrastructure. Distributed under MIT.

The problem it solves

AI agents are increasingly powerful and increasingly trusted with consequential work. Most agent frameworks — LangGraph, AutoGen, CrewAI, OpenAI's Agents SDK — give agents tools to call and freedom to call them. They do not enforce who can call what, do not require human approval before destructive actions, and do not produce forensic audit trails when something goes wrong.

Industry evidence from 2026 documents this gap concretely. Gravitee's *State of AI Agent Security 2026* survey of 919 executives and practitioners found that for agent-to-agent interactions, teams rely heavily on insecure or shared methods for authentication, including API Keys at **45.6%** and Generic Tokens at **44.4%**, while secure standards like mTLS are utilized by only **17.8%**. Only **21.9%** of teams treat AI agents as independent, identity-bearing entities, and **27.2%** of technical teams have reverted to custom, hardcoded logic to manage authorization. **88%** of organizations reported confirmed or suspected AI agent security incidents in the last twelve months; in healthcare the rate reaches **92.7%**. **82%** of executives report confidence that their existing policies protect against unauthorized agent actions, while only **14.4%** of organizations send agents to production with full security or IT approval.

The academic literature reaches the same conclusion. A March 2026 paper on pre-action authorization for autonomous agents states that *"the gap between what AI agents can do and what they should do is an authorization problem, not an alignment problem. Just as early multi-user systems lacked role-based access control and early web APIs lacked delegated authorization (OAuth), autonomous agents today lack a standard mechanism to enforce per-action authorization before execution."*

AOS addresses four operational realities that no major agent framework handles natively:

1. **Unauthorized actions** — an agent given broad tool access can use tools it should not. AOS enforces capability narrowing so an

agent's child can never have more authority than its parent, and an agent without the capability for a tool cannot call it regardless of what the language model decides.
2. **Autonomous destructive actions** — when an agent decides to delete, send, modify, or execute, most frameworks just do it. AOS classifies tools by reversibility; irreversible tools cannot fire without an operator approving the exact tool, exact arguments, and visible blast radius.
3. **Silent forensic gaps** — when something goes wrong, most frameworks have no record of what the agent saw, decided, and did. AOS records every action, tool call, memory operation, and state transition in a SHA-256 hash chain partitioned per session, with anchor objects in independently-credentialed object storage so tampering is detectable.
4. **Invisible costs** — most frameworks let LLM costs accumulate per task with discovery at the end of the billing cycle. AOS records every model call with input/output tokens and computed USD cost; configurable session ceilings raise an exception before the call rather than after.

What it does

Capability narrowing. Every agent carries a signed JWT capability token listing exactly which tools it can call. Child agents receive the intersection of the parent's grants and the child's request — never more. Verification fires before every tool call, every memory operation, and every spawn. There is no ambient authority and no escape hatch short of a direct database connection, which agents never receive.

Mandatory approval for destructive actions. Tools registered as `reversible=False` cannot execute autonomously. When an agent attempts one, AOS takes a checkpoint, inserts a pending review row with the tool name and arguments, and blocks the agent. An operator reviews via the dashboard or API and approves or rejects. Approvals are single-use and bound to the specific (tool, argument-hash) pair — repeat calls require fresh approvals.

Tamper-evident audit chain. Every action — spawn, tool call, memory write, state transition, approval request, capability denial — is one row in a SHA-256 hash chain partitioned per session. Anchor objects written every 1000 rows to a separately-credentialed object store make mid-chain tampering detectable even by an attacker with PostgreSQL write access.

Crash recovery. Composite checkpoints capture LangGraph thread state, agent control block, and working memory atomically. Scheduler workers heartbeat their claimed agents; an orphan reaper reclaims any agent whose worker has gone silent and re-queues it for resume from the last checkpoint. State-machine validation in PostgreSQL prevents invalid transitions like terminated-to-runnable.

Cost ledger. Every LLM call is recorded with model, tokens, and computed USD cost. A configurable session ceiling raises an exception before the call rather than after. Costs roll up by agent, by session, and by day.

Memory lifecycle governance. Working memory entries carry importance scores and decay rates. Agents can pin critical facts against eviction, explicitly forget stale ones, or trigger sweeps at natural checkpoints. Cross-session experiential memory accumulates per agent class — every future instance of a class shares lessons learned by prior instances, with confidence rising on positive reinforcement and falling on negative.

Knowledge base substrate. Operator-loaded documents indexed via pgvector with hybrid vector + keyword search, scoped retrieval through metadata filtering, stable chunk citations that the dashboard links back to specific passages, and incremental ingestion that skips unchanged documents.

LLM backend abstraction. Five backends supported (Anthropic API, AWS Bedrock, OpenAI, local Ollama, Claude Code CLI shim) with per-agent-class data-egress envelopes. An agent class registered with `allowed_backends={"ollama"}` cannot accidentally route to Anthropic regardless of operator configuration — regulator-grade evidence that data didn't leave the deployment.

Prompt injection defense. Tool results are wrapped in untrusted-content delimiters so the language model treats tool output as data rather than instructions. Output is scanned for injection-suspicious patterns; detections prepend security warnings and write audit events. The system prompt teaches every agent class to follow only the operator's task and never instructions found in tool results.

MCP integration. Agents reach external knowledge bases, internal APIs, and consumer services through the Model Context Protocol. Each MCP tool inherits the same capability, audit, and approval pipeline. Conversely, AOS itself exposes 11 governance tools as an MCP server so Claude Code, Cursor, Cline, and similar IDE-integrated agents can dispatch governed swarms with one configuration change.

What it does not do

- **Provide AI models, prompts, or business logic.** AOS wraps agents; it doesn't replace them. Bring your own LangGraph (or other) agent definitions.
- **Run mid-inference preemption.** The scheduler can preempt between LangGraph node boundaries, not inside an LLM call. Documented trade-off.
- **Protect against fully-compromised infrastructure.** If an attacker holds both PostgreSQL and audit-anchor credentials, the chain can be rewritten. Documented out-of-scope; operators who care must route audit to an external append-only service.
- **Detect reasoning-redirection injection.** Adversarial content designed to make the language model reach a specific conclusion — rather than call a specific tool — is not detected. The judge-model defense pattern is deferred to a future release.
- **Authenticate end users.** AOS authenticates API callers via bearer tokens; user identity, single sign-on, and role-based access at the application layer are the host application's responsibility.

Customer profile

The customer profile is anyone running AI agents that take real actions on systems where unsupervised destructive operations are unacceptable:

- **Critical infrastructure operations.** Read-only diagnostic agents on production systems where state changes need sign-off and audit. IBM i shops, Kubernetes operators, database administrators, network engineers, security incident responders.
- **Regulated industries.** Financial services, healthcare, legal, insurance — environments where "we used AI" is acceptable only if "here is the cryptographic record of every action it took, who authorized state changes, and how much each session cost" comes with it.
- **Multi-agent platforms.** Teams running concurrent agent fleets where coordination, capability boundaries, and cost attribution matter at the per-session level.
- **Development tooling.** Claude Code and similar IDE-integrated agents that spawn sub-agents for code review, bug investigation, or release readiness — workflows where the parent session shouldn't grant the child everything it has.

Where it stands

Production-shaped alpha. Core security primitives correctly implemented, anti-hallucination plumbing more rigorous than typical agent frameworks, prompt injection defenses for direct attacks now in place. Honest threat model documenting what's covered and what isn't. Validated through internal dogfood — the runtime's own architectural decisions have been informed by its own swarm classes. Has not yet been hardened by external deployment.

What's left for first-customer readiness: deployment runbook, response and disclosure SLAs, multi-key auth, operator UX polish, performance characterization under realistic load. None of these are runtime gaps; they are operational gaps that close through real deployment rather than further code.

Position in the landscape

The agent runtime market in 2026 has crowded categories and one open gap.

- **Durable execution** — Temporal, DBOS, Restate, Inngest. Mature, well-funded, broad framework support. AOS does not compete on durability; it includes durability as one property among many.
- **Agent memory** — Letta, Mem0, Zep. Specialists with deeper retrieval features. AOS includes competent memory primitives in service of governance, not as the product.
- **Capability-based authorization for AI agents** — documented as a structural gap by multiple 2026 sources. The Gravitee survey, the academic pre-action authorization paper, and an audit of 30 popular open-source AI agent projects all reach the same conclusion. The Grantex audit of 30 popular AI agent projects published in March 2026 found that *"nearly all of them rely on unscoped API keys with no per-agent identity, no user consent, and no revocation mechanism."*

Sharp positioning: **a capability-authorization runtime for AI agents, with built-in human approval gates and tamper-evident audit, designed for systems where destructive actions cannot be autonomous.**

Thirty-second pitch

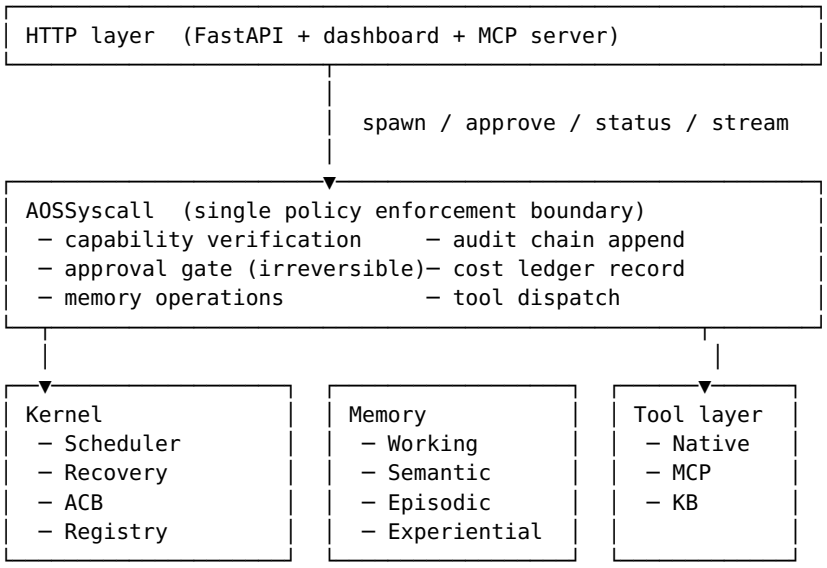
AI agents can now do real work on real systems, but every framework treats authorization as a best practice. AOS treats it as a runtime property. An agent governed by AOS cannot execute a destructive action without an operator approving the exact tool and arguments, cannot exceed its parent's authority, cannot exhaust the LLM budget, cannot follow injection-shaped instructions in untrusted content, and produces a cryptographic audit trail of every step. It runs as a library next to an existing PostgreSQL or as a standalone service. Open source, MIT licensed, ships today.

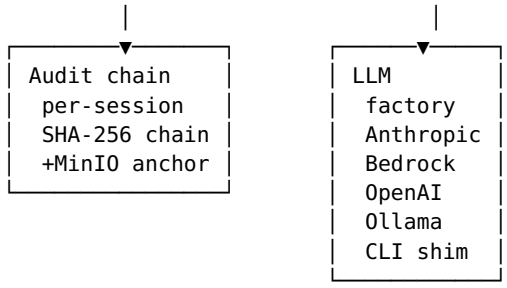
Technical Overview

The technical overview that follows describes the implementation as it stands in the codebase. None of the technical claims in this section are statistical or sourced from external research — they describe the system's actual architecture, files, and behavior, all of which are verifiable directly from the code.

System architecture

AOS is a Python runtime with five top-level subsystems plus a thin HTTP layer. Every operation an agent performs crosses a single syscall interface — `AOSSyscall` — which is the policy enforcement choke point. Agents never hold a database connection, never call tools directly, and never invoke LangGraph directly. The architecture is deliberately analogous to a Unix kernel: agents are processes, the syscall interface is the kernel API, capability tokens are the security context, and the audit log is the system log.





PostgreSQL backs every persistent component: agent processes, working memory, checkpoints, audit log, cost events, capability tokens (verified statelessly via JWT), tool registry metadata, and the knowledge base. pgvector extends PostgreSQL with vector similarity search for memory recall and KB retrieval. Optional MinIO or S3-compatible object storage holds audit chain anchors. No other infrastructure is required.

The syscall interface

Every agent operation crosses `AOSSyscall`. This is the load-bearing architectural decision: by funneling all behavior through one boundary, the runtime can enforce capability checks, audit appends, and approval gates uniformly without trusting the agent to follow conventions.

The syscall surface is a Unix-shaped API:

Syscall	Purpose
`spawn(class, task, grants)`	Fork a child agent with narrowed capability
`wait(child_id)`	Block until child reaches zombie state
`exit(result)`	Terminate this agent
`tool_call(name, args)`	Capability-checked tool invocation
`mem_read(key)`	Read working memory
`mem_write(key, value)`	Write working/semantic/episodic memory
`mem_recall(query)`	Semantic recall across memory types
`mem_retain(key, importance)`	Pin against eviction
`mem_forget(key)`	Explicit deletion
`mem_evict()`	Bulk sweep below threshold
`experience_write/recall/reinforce/weaken`	Cross-session learning
`record_llm_cost(model, tokens)`	Cost accounting
`yield_cpu()`	Voluntary scheduling yield
`audit_mark(event_type, payload)`	Application audit event

Capability verification fires before every operation. Audit append fires after every operation. The agent's code path cannot reach the underlying database, the LangGraph executor, the tool dispatcher, or the audit chain except through this interface.

Capability tokens

Every agent carries a signed JWT capability token. The token's claims are:

- `agent_id` — the holder
- `parent_agent_id` — null for root agents
- `tools` — list of tool names this agent may invoke
- `memory_namespaces` — paths this agent may read/write under
- `memory_ops` — `["read"]`, `["read", "write"]`, etc.
- `delegation_depth` — 0 for root, increments on each spawn
- `delegation_max` — depth ceiling, clamped to a hard runtime limit
- `max_child_agents` — fan-out limit
- `token_budget` — LLM token ceiling for this agent's lifetime
- `jti` — unique token id for audit correlation
- `exp` — expiration timestamp

Delegation is intersection-narrowing: when a parent calls `spawn(requested_grants)`, the child's grants become `parent.tools ∩ requested.tools`, with `delegation_depth = parent.depth + 1`. Empty intersections raise `InsufficientAuthority`. The child token is signed with the same secret as the parent and verified statelessly on every operation — capability tokens never leave their original signing service.

The signing service is HS256 with a secret per AOS deployment. Per-agent-class data-egress envelopes are enforced separately via the registry's `allowed_backends` field, which the syscall consults at LLM construction time — a token alone cannot determine whether an agent class is permitted to send data to a particular backend.

Audit chain

Every audit event is one row with these fields: `id` (BIGSERIAL), `session_id`, `agent_id`, `event_type`, `payload` (JSONB), `prev_hash`, `chain_seq` (per-session monotonic), `row_hash`, `created_at`. The `row_hash` is `SHA-256(prev_hash || chain_seq || event_type || canonical_json(payload) || created_at)`. The `prev_hash` of `chain_seq=N` equals the `row_hash` of `chain_seq=N-1` for the same session.

Per-session chains avoid serialization contention across concurrent sessions. Each session's chain serializes via `pg_advisory_xact_lock(hashtextextended(session_id, 0))` so writes within a session are linearized, but writes across sessions are concurrent.

Tamper evidence comes from anchor objects: every 1000 rows, the runtime writes the latest chain_seq, row_hash, and metadata to MinIO or an S3-compatible bucket with credentials independent from the PostgreSQL credentials. An attacker with PostgreSQL write access alone can rewrite rows but cannot rewrite the anchor; verification walks the chain forward from the previous anchor and compares against the next anchor. Mismatch detects mid-chain tampering.

The runtime also tracks anchor write failures and surfaces them in `/api/v1/health`. Silent anchor degradation — anchors not being written successfully — is the single most likely failure mode of the tamper-evidence guarantee, so visibility into it is first-class.

Approval gate for irreversible tools

Tools are registered with `reversible=True` (read-only or idempotent) or `reversible=False` (destructive). The split is application-defined; the runtime enforces it.

When an agent calls a tool registered as `reversible=False`:

1. Capability verification (as for any tool).
2. Composite checkpoint: LangGraph thread state, ACB snapshot, working memory snapshot, atomically.
3. Check for a standing approval: the runtime looks up `approved_tool_call:{tool_name}:{args_hash}` in working memory.
4. If found: consume (delete) the approval row, dispatch the tool, write `TOOL_CALL` and `TOOL_RESULT` audit events, return.
5. If not found: insert a pending review row with `(agent_id, session_id, tool_name, args, args_hash)`, raise `PendingApproval`. The scheduler catches `PendingApproval` and transitions the agent to `BLOCKED(human_review)`. An operator reviews via the dashboard, approves or rejects. Approval writes the matching working memory key and unblocks the agent. The agent's LangGraph node re-executes on resume — checkpointing handles this cleanly — and finds the approval on the second pass.

Approvals are single-use: a second invocation of the same `(tool, args)` requires a new approval. The args_hash is computed as `SHA-256(canonical_json(args))` so different argument permutations produce different approval keys.

The MCP server exposes the approval queue and approve/reject operations. Critically, the runtime cannot enforce that MCP clients surface approvals to humans rather than auto-firing — that's a contract with the client, documented as an anti-pattern. Claude Code respects the contract by design.

Scheduler

The scheduler is an asyncio event loop driving multiple concurrent agents through their LangGraph graphs. Concurrency is bounded by `AOS_MAX_SLOTS` (default 8). Agents compete for slots by priority (0 = highest, 100 = lowest) with FIFO within a priority band.

The dispatch flow:

1. The dequeue function uses `FOR UPDATE SKIP LOCKED` against `agent_processes WHERE state='runnable' ORDER BY priority, created_at`. Multiple worker instances can dequeue concurrently without conflict.
2. The dequeued agent's row is updated to `state='running'`, `worker_id=<this worker>`, `last_heartbeat=now()`.
3. The worker constructs an `AOSSyscall` for the agent and invokes one step of the agent's LangGraph graph.
4. State transitions are gated by a state-machine SQL function. Invalid transitions (e.g., `terminated → runnable`) match zero rows.

5. After the step, the agent transitions back to `runnable` (continue), `blocked` (waiting on tool/child/approval), `zombie` (terminal), or `suspended` (preempted).

Workers heartbeat their claimed agents every few seconds. An orphan reaper detects agents whose worker has gone silent past a threshold and re-queues them. Combined with the composite checkpoint, this means an agent whose worker crashed mid-step resumes cleanly from the last checkpoint without operator intervention.

Crash recovery

Three components captured atomically per checkpoint:

1. **LangGraph thread state** — the `AsyncPostgresSaver` checkpointer writes graph state to the `checkpoints` table.
2. **Agent control block** — current state, token budget consumed, delegation depth, namespace.
3. **Working memory snapshot** — all key-value pairs in the agent's namespace.

A composite checkpoint is the unit of recovery. On orphan detection or explicit suspend-and-resume, the scheduler restores all three from the most recent successful checkpoint. The combination — durable graph state plus durable agent metadata plus durable working memory — means resume is correct regardless of which component the crash interrupted.

Pre-irreversible-tool checkpoints fire automatically before any `reversible=False` tool call, so even if the runtime crashes between the approval check and the tool dispatch, replay is replay-safe.

Memory layer

Four memory types, all reachable via the syscall:

Working memory — key/value, agent-scoped, ephemeral. Per-key TTL, importance scoring, decay-based eviction. The substrate for short-term agent state.

Semantic memory — domain facts indexed by category and topic, vector-embedded for similarity search. Application-owned tables joined via the post-write hook pattern; agents see them via `mem_recall`.

Episodic memory — session log entries with structured outcomes and lessons. Vector-embedded; recallable via `mem_recall` filtered by type.

Experiential memory — cross-session learned behaviors per agent class. Every instance of `BugHuntSwarm` shares experiences written by all prior instances. Confidence rises with `experience_reinforce` calls, falls with `experience_weaken`. Stale patterns fade out without manual curation.

The decay model for working memory uses `decay_score = importance + access_frequency_bonus - age_hours * decay_rate`. Entries at `importance >= 0.9` (pinned via `mem_retain`) are never evicted. Entries with explicit TTLs expire naturally. The decay model and eviction threshold are operator-configurable per `DecayPolicy`.

Knowledge base

Operator-loaded documents indexed in the `aos_documents` table. Each row carries `source_uri`, `chunk_index`, `content`, `metadata` (JSONB), `embedding` (pgvector), and `content_hash` (SHA-256 of content for incremental ingestion).

Two indexes drive retrieval:

- HNSW index on `embedding` for cosine similarity (vector search)
- GIN index on `tsv` (generated tsvector from content) for keyword ranking via `ts_rank_cd`

Hybrid search composes both: `score = vector_weight * vector_similarity + keyword_weight * ts_rank` with operator-tunable weights. Default 0.7/0.3 favors semantic similarity but the keyword component substantially improves retrieval on identifier-heavy queries (codes, names, version strings).

Metadata filtering at search time lets one corpus serve multiple scoped tools. `kb_search` with `metadata_filter={"scope": "runbooks"}` returns only runbook chunks; with `source_uri_prefix="kb://policies/"` returns only policies. Operators register scoped tool variants (`kb_search_runbooks`, `kb_search_policies`) as one-line wrappers.

Markdown-aware chunking splits documents on H2/H3 headings rather than blank lines, with chunk overlap and heading-path metadata. Chunk citations resolve to stable anchors (`#heading:section-slug` or `#chunk:N`) that the dashboard links back to the source passage.

Incremental ingestion compares content hashes to skip unchanged chunks. Re-running the loader on a 10,000-document corpus where one document changed re-embeds one chunk, not 10,000.

LLM backend abstraction

Five backends, selected at LLM-construction time via the factory:

- **Anthropic API** — direct via `langchain_anthropic`. Requires `ANTHROPIC_API_KEY`.
- **AWS Bedrock** — via `langchain_aws`. Requires AWS credentials.
- **OpenAI** — via `langchain_openai`. Requires `OPENAI_API_KEY`.
- **Ollama** — local inference via `langchain_ollama`. No external dependency.
- **CLI shim** — subprocess to a local Claude Code binary. Inherits the developer's mounted session.

Per-agent-class data-egress envelopes restrict which backends a class may use. An agent class registered with `allowed_backends={"ollama", "cli"}` cannot route to Anthropic regardless of operator configuration. Mismatches raise `BackendNotPermittedForAgentClass` at construction. The factory also validates `requires_tool_calling` against the backend's tool-calling capability — an agent class that requires structured tool calls cannot run on a model that lacks tool support.

Anti-hallucination architecture

The swarm classes (`BugHuntSwarm`, `CodeReviewSwarm`, etc.) inherit from `BaseSwarmAgent`, which compiles a four-node LangGraph: `reason → execute_tools → reason → ... → validate_proposal → finalize`. Seven layers prevent hallucinated or ungrounded outputs:

1. **Schema-bound tool calls.** When the backend supports `bind_tools`, the LLM is constrained at the API level to emit calls matching the registered Pydantic schema. Malformed JSON is rejected at the model layer.
2. **Pydantic validation.** Every emitted `emit_proposal` is parsed against the swarm's proposal schema. Type mismatches, missing required fields, or invalid enum values reject the proposal and return a structured error to the LLM for correction.
3. **Audit-row evidence verification.** Every proposal must cite audit row IDs in its `evidence` field. The runtime queries `agent_audit_log` to verify each cited ID exists in this agent's session. Fabricated IDs reject the proposal.
4. **Tool result evidence stamps.** Every tool result is prefixed with `[evidence: audit#N]` matching its TOOL_RESULT row. The LLM cites these IDs; the runtime verifies them.
5. **Loop guard.** Identical `(tool, args)` pairs occurring twice in the message history are refused at dispatch with a synthetic error message telling the model to break the loop.
6. **Force-investigate gate.** A non-trivial proposal (non-empty actions, non-low severity) with zero `[evidence: audit#N]` stamps in the message history is rejected. The agent must use a read or KB tool before claiming findings.
7. **Abandonment after repeated refusals.** After two force-investigate refusals, the agent terminates with `APP_AGENT_ABANDONED_UNGROUNDED` rather than burning more budget on a model that refuses to investigate.

These layers compose. A proposal must be schema-valid, must cite real audit rows, must be backed by tools the agent actually called, and must follow at least one investigation step. Together they make hallucinated proposals structurally impossible to ship.

Prompt injection defense

After the latest sprint, two layers protect against direct injection:

Untrusted-content delimiter. Every tool result is wrapped in `<untrusted_tool_output tool="..." source_kind="native|mcp" path="...">...content...</untrusted_tool_output>`. The system prompt teaches every swarm class to treat delimited content as data, never instructions. Injection-shaped content inside the delimiter has structural standing as data, not as a directive. The `[evidence: audit#N]` stamp remains the first line, preserving the citation pattern.

Pattern detection. Tool outputs are scanned for known injection patterns: literal `<tool_call>` tags, JSON-fence tool-call shapes, role-prefix markers (`System:`, `Assistant:`), and "ignore previous instructions" phrases. Detections prepend a `SECURITY_WARNING` block to the wrapped content and write an `APP_INJECTION_PATTERN_DETECTED` audit event. The agent is not blocked — false positives are inevitable — but the warning gives the LLM explicit instruction to not follow embedded directives, and the audit event surfaces forensically.

What's not yet defended: reasoning-redirection injection (the judge-model pattern, deferred), KB content scanning at ingest (deferred), multi-turn injection that drifts over many tool calls (no good general defense).

Multi-persona swarms

A swarm class can declare a list of `Persona` objects. When a parent of such a class is spawned, the runtime fans out: one child per persona via `aos.spawn`, each child running the same compiled graph with a persona-overlay system prompt that narrows its perspective. Children investigate independently, emit structured proposals via `emit_proposal`, and zombie. The parent's `await_children` node collects their proposals and runs an aggregation reason node that synthesizes one consolidated proposal citing the union of children's evidence.

Failure modes are configurable:

- `best_effort` — finalize with whatever children succeeded; mention failures in notes
- `all_or_nothing` — any child failure yields no parent proposal
- `min_threshold` — succeed if at least N children returned valid proposals

This is the substrate for the LLM backend decisions document — five perspectives on the same architectural question, each grounded in real evidence, aggregated into one decision artifact.

MCP integration

AOS plays both sides of the Model Context Protocol.

As an MCP client. The `mcp_bridge` in `aos/tools/` lets agents call tools registered on remote MCP servers. Each external MCP tool registers in the runtime's tool registry with explicit reversibility classification, inheriting the same capability check, audit append, and (for irreversible tools) approval gate as native handlers. External knowledge bases, internal APIs, and consumer services all become available without bespoke integration code.

As an MCP server. The `aos.mcp_server` package exposes 11 governance tools — `spawn_swarm`, `wait_for_swarm`, `get_swarm_status`, `stream_swarm_events`, `list_pending_approvals`, `get_approval_details`, `approve_action`, `reject_action`, `list_swarm_classes`, `get_audit_trail`, `get_session_cost` — through standard MCP transports (stdio for desktop clients like Claude Code, HTTP/SSE for remote clients). Adding AOS to a Claude Code session is one configuration change. Sub-agents spawned through AOS get capability narrowing, audit, and approval gates without modification to Claude Code itself.

The MCP server is stateless — every tool delegates to the AOS REST API. Auth flows through bearer token forwarding. The trust boundary is between AOS and the MCP server; the MCP client sits inside that boundary on the operator's workstation.

Cost ledger

Every LLM call writes a row to `agent_cost_events` with model, input tokens, output tokens, computed USD cost, agent ID, session ID, and optional context label. The cost computation uses `ModelPriceTable` — a registry of per-million-token prices that operators can extend or override.

`CostBudget(max_cost_usd)` configured per session causes `record_llm_cost` to raise `CostBudgetExceeded` before the call would push the session over the ceiling. The budget check fires before the LLM API call, not after — by the time `record_llm_cost` is called, the API request has already happened, but the next call will raise before its API request is made.

For free-priced models (Ollama, locally-run inference), `price_source` markers distinguish “actually free” from “synthetic price” so compliance reporting doesn’t quote fictional dollars. A complementary `AOS_MAX_SESSION_TOKENS` env var enforces a token-count ceiling independent of cost.

Threat model

The threat model document specifies in scope and out of scope explicitly with file and migration references. Briefly:

In scope:

- Capability narrowing prevents agent-to-agent privilege escalation
- Per-session audit chain detects PostgreSQL-write tampering when MinIO is independently credentialed
- Mandatory approval prevents autonomous destructive actions when tools are registered correctly
- Cost ceilings prevent runaway LLM spend
- Per-agent-class backend constraints prevent data egress to unintended providers
- Untrusted-content delimiters and pattern detection raise the floor against direct prompt injection

Out of scope, documented:

- Fully-compromised infrastructure operator with both PostgreSQL and audit-anchor write access
- Reasoning-redirection injection (the judge-model pattern is the standard defense; deferred)
- KB content poisoning at ingest time (operators are the trust boundary)
- Multi-turn injection drifting across many tool results
- Adversarially-crafted patterns that evade the regex detectors
- LLM hallucination of correct-looking but wrong tool arguments where capabilities permit (the seven-layer evidence stack catches most cases but cannot prove absence)

Observability

`/api/v1/health` returns: `auth_mode`, `reasoning_backend_status`, `audit_anchors` (success and failure counts), database pool status, scheduler heartbeat counts, and pending approval queue depth.

The dashboard renders agents grouped by session, the swarm tree of parent-child relationships, audit timelines per session, the pending approval queue with proposed actions and blast radius, and per-

session cost rollups. Server-Sent Events stream live agent state changes so operators see updates without refreshing.

Audit-event filtering by event type, session, agent, or time range is built in. Operators can replay any session's full audit trail to reconstruct what happened.

Testing

The test suite covers the syscall interface, capability delegation, audit chain integrity, scheduler state transitions, recovery composites, cost ledger, memory lifecycle, swarm graph behaviors, MCP tool wiring, prompt injection patterns, and the knowledge base. Tests use fakes for the database pool and embedder so they run without infrastructure dependencies. A separate `tests/integration/` directory holds tests gated behind real PostgreSQL + pgvector for end-to-end verification.

Deployment

Single Postgres connection, single Python process for the API server, optional separate process for the MCP server. Docker Compose stack ships with the runtime. Bind-mounts mean code changes take effect on container restart without rebuild.

`pip install aos-runtime` installs the core. Optional extras: `[server]` for the FastAPI dashboard, `[embeddings]` for sentence-transformers, `[audit-anchors]` for MinIO/S3 anchor support, `[mcp]` for the MCP server, `[ollama]` for local inference, `[all]` for everything.

Migrations run automatically at startup, gated by a Postgres advisory lock so multiple instances starting concurrently don't conflict. Migration files are forward-only and idempotent. Rollback is operator responsibility (point-in-time PostgreSQL recovery).

What's next

Items deferred from previous sprints but architecturally bounded:

- Judge-model pattern for reasoning-redirection injection defense
- KB content scanning at ingest with operator review queue
- Multi-key auth with per-key audit attribution
- OIDC/mTLS for production deployments
- Performance characterization under realistic concurrency
- Operator UX hardening through deployed use
- First external customer integration

The runtime is no longer the bottleneck. The remaining work is operational hardening through deployment rather than further code.

Sources

External claims in the executive summary are drawn from these published sources. Every percentage and external statistic above can be verified against them:

- **Gravitee, *State of AI Agent Security 2026 Report*** (survey of 919 executives and practitioners). Key findings cited: 45.6% of teams use shared API keys for agent-to-agent auth; 44.4% use generic tokens; 17.8% use mTLS; only 21.9% treat agents as independent identity-bearing entities; 27.2% have reverted to custom hardcoded authorization logic; 88% reported confirmed or suspected AI agent security incidents in the prior year; 92.7% in healthcare; 80.9% past planning into testing or production; only 14.4% deploy with full security/IT approval; 82% of executives confident their policies prevent unauthorized agent actions. Primary URLs: https://www.gravitee.io/state-of-ai-agent-security and https://www.gravitee.io/blog/state-of-ai-agent-security-2026-report-when-adoption-outpaces-control. Full PDF: https://www.gravitee.io/hubfs/Downloadable%20Resource/state_of_ai_agent_security_report_pdf_2026.pdf.

- **Grantex Research, *State of AI Agent Security 2026*** (audit of 30 popular open-source AI agent projects, March 2026). Cited finding: “nearly all rely on unscoped API keys with no per-agent identity, no user consent, and no revocation mechanism.” URL: https://grantex.dev/report/state-of-agent-security-2026.

- **VentureBeat / Fountain City coverage** of the Gravitee survey corroborate the 88% incident rate, 21.9% identity-bearing rate, and 14.4% security-approval rate. URLs: https://venturebeat.com/security/most-enterprises-cant-stop-stage-three-ai-agent-threats-venturebeat-survey-finds; https://fountaincity.tech/resources/blog/ai-agent-security-enterprise-guide/.

- **Pre-action authorization paper, arXiv:2603.20953**, March 2026. Cited for the structural authorization-gap framing: “the gap between what AI agents can do and what they should do is an authorization problem, not an alignment problem.”

For any reader doing due diligence: the Gravitee report PDF and the academic paper are the authoritative sources. The other links are corroborative reporting on those primary sources. If you publish the executive summary externally, link directly to the Gravitee report and the arXiv paper rather than to the secondary coverage.