Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 1 September 2026

A. Cowles
Quox Ltd

28 February 2026

Agent Orchestration Control Layers (AOCL) Protocol
draft-cowles-aocl-00

Abstract

Agent Orchestration Control Layers (AOCL) is a protocol that standardizes how an orchestrator processes incoming events by passing them through a layered control pipeline. AOCL defines an eleven-layer taxonomy covering ingress normalization, identity scoping, smart routing, policy gating, plan decomposition, context retrieval, prompt shaping, delegation and execution, verification, response assembly, and audit writeback. The protocol is runtime-agnostic and framework-agnostic, producing auditable governance traces as a first-class output. AOCL supports both sequential pipeline and directed acyclic graph (DAG) execution modes, with explicit bypass and branch semantics that mandate audit records for all control-flow deviations.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 September 2026.

Copyright Notice

Table of Contents

1. Introduction

Autonomous AI agents are increasingly deployed in enterprise environments where they perform consequential actions: executing code, accessing databases, managing infrastructure, and interacting with external services. Despite this growing responsibility, there is no widely adopted standard for how agent actions flow through identity verification, policy enforcement, execution delegation, and audit recording.

Without a structured control pipeline, agent systems exhibit several governance failures: actions are taken without policy checks, bypass decisions are unrecorded, identity and permission scoping is ad-hoc, and there is no reproducible trace explaining why a particular action was taken or what evidence supports the result.

The Agent Orchestration Control Layers (AOCL) protocol addresses these problems by defining a layered control pipeline that an orchestrator runs when processing any incoming event. AOCL does not prescribe how agents are built, how tools execute, or how workers are scheduled. Instead, it standardizes the control layers through which work flows, the minimum contract each layer MUST follow, and the audit records that MUST be emitted.

AOCL is designed to be runtime-agnostic and framework-agnostic. It operates alongside the Agent Envelope Exchange (AEE) protocol [AEE], using AEE envelopes as the audit and delegation format without requiring any changes to the AEE specification. Verifiable audit trails MAY additionally be anchored using the VOLT protocol [VOLT].

The protocol defines eleven canonical layers (L0 through L10), a structured context bundle with seven partitions (C0 through C6), and two stack execution modes (pipeline and DAG). All control-flow deviations, including layer bypasses and branch decisions, MUST produce explicit audit records.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document:

Intent A triggering event that initiates AOCL processing. Examples include user messages, monitoring alerts, webhook callbacks, API calls, and scheduled ticks. AOCL does not constrain the event format, but commonly an intent arrives as an AEE [AEE] task envelope.

Orchestrator The component that runs AOCL stacks. It may be a single process or distributed across multiple nodes. The orchestrator is responsible for layer sequencing, context propagation, control-flow decisions, and audit emission.

Layer A named unit of control logic that takes an input state and produces a possibly modified state, decisions and/or actions, and control flags that influence routing (branch, bypass, or halt). Layers can be deterministic code, LLM calls, or hybrids.

Stack A configured sequence or graph of layers used for a class of tasks (e.g., "default", "realtime-alert", "deep-research").

Context Bundle A structured state object passed through AOCL layers, partitioned into seven segments (C0 through C6) that separate concerns such as event metadata, identity, task definition, memory, policy, execution parameters, and audit requirements.

Control Flags Signals emitted by a layer to influence downstream

processing. Standard flags include halt_pipeline, require_hitl, branch_to, and bypass_layers.

Context Delta A patch or merge delta representing changes a layer made to the context bundle. Layers SHOULD emit deltas rather than full context copies to minimize payload size and enable efficient replay.

3. Architecture Overview

AOCL defines a control-plane architecture for agent orchestration. The architecture consists of four primary components: layers, stacks, context bundles, and the orchestrator.

3.1. Layers

A layer is the fundamental unit of control logic. Each layer receives an input context bundle, performs its designated function (normalization, policy checking, planning, etc.), and produces an output consisting of:

* A possibly modified context bundle (expressed as a delta);

* Zero or more decisions with coded reasons;

* Control flags that may alter downstream processing.

Layers are intentionally opaque in their implementation. A layer MAY be implemented as deterministic code, an LLM inference call, a rules engine, a remote service invocation, or any combination thereof. AOCL specifies only the input and output contracts, not the internal mechanism.

3.2. Stacks

A stack is a configured arrangement of layers for a particular class of work. Stacks MAY be expressed as a sequential pipeline (ordered list) or as a directed acyclic graph (DAG) with conditional edges. An orchestrator MAY support multiple named stacks and select among them based on the characteristics of the incoming intent.

3.3. Context Bundles

The context bundle is the shared state object that flows through layers. It is partitioned into seven segments (C0 through C6) to separate concerns and enable selective access. Layers SHOULD emit context deltas (patches) and references/digests rather than re-transmitting large objects. The context bundle SHOULD remain inspectable and replayable.

3.4. Orchestrator Role

The orchestrator is responsible for:

* Receiving incoming intents;
* Selecting the appropriate stack;
* Executing layers in the order defined by the stack;
* Propagating the context bundle between layers;
* Honoring control flags (branch, bypass, halt);
* Emitting audit records for every layer activation and control-flow decision.

3.5. Scope Boundaries

AOCL explicitly does NOT define:

* How agents are built or structured internally;
* How tools execute or what APIs they expose;
* How workers are scheduled or load-balanced;
* Transport bindings (HTTP, WebSocket, NATS, Kafka);
* Cryptographic signature formats;
* A formal expression language for DAG edge conditions;
* A standardized plugin registry for layer implementation references.

These concerns are deferred to runtime implementations or future revisions of this specification.

4. Design Principles

4.1. Layered Control

Work flows through ordered layers with clearly separated responsibilities. Each layer has a single primary concern (e.g., identity scoping, policy enforcement, context retrieval). This separation enables independent evolution, testing, and replacement of individual layers without affecting the overall pipeline structure.

4.2. Branchable and Bypassable

AOCL supports alternate execution paths (branches) and the ability to skip layers (bypasses). However, all branch and bypass decisions MUST be auditable. The orchestrator MUST emit explicit records documenting which layers were bypassed or which branch was taken, who authorized the deviation, and why it was permitted.

4.3. Delta-First Context

Layers SHOULD emit context deltas (patches) and references/digests rather than re-transmitting entire context objects. This principle keeps audit payloads small, enables efficient replay, and allows integrity verification through digest comparison. Large data (e.g., RAG results, file contents) SHOULD be referenced by URI or content hash rather than embedded inline.

4.4. Audit-First

AOCL is designed to produce traces that answer five fundamental governance questions:

1. What happened?

2. Why did it happen?

3. What changed in the context?

4. Who or what approved any bypasses or deviations?

5. What evidence supports the result?

Every layer activation, decision, and control-flow change MUST produce an audit record. Observability is a first-class output of the protocol, not an optional add-on.

5. Context Bundle

The context bundle is the structured state object that flows through all AOCL layers. AOCL does not mandate a single rigid schema, but RECOMMENDS partitioning the bundle into seven segments (C0 through C6) to maintain clear separation of concerns.

5.1. C0: Event

Contains metadata about the triggering event: source identifier, timestamp, channel (chat, webhook, API, schedule), correlation IDs, and any attachments or raw input data. This partition is populated during L0 (ingress normalization) and is generally read-only for subsequent layers.

5.2. C1: Identity and Scope

Contains the authenticated identity context: user and/or organization identifiers, roles, permissions, secret scope boundaries, and redaction rules. This partition is populated during L1 (identity scoping) and is consumed by policy and execution layers to enforce access control.

5.3. C2: Task

Contains the structured task definition: goal statement, constraints, definition of done, priority level, and any decomposition into sub-tasks. This partition is refined during L4 (plan decomposition) and consumed during L7 (delegation and execution).

5.4. C3: Memory and Knowledge

Contains retrieved contextual information: memory notes, retrieval-augmented generation (RAG) results, file references, and bounded summaries. This partition is populated during L5 (context retrieval) and consumed during L6 (shape and rewrite) and L7 (delegation and execution). Large data objects SHOULD be stored as references rather than inline values.

5.5. C4: Policy

Contains active policy constraints: safety and compliance rules, allowed tool lists, model restrictions, content filtering requirements, and human-in-the-loop (HITL) thresholds. This partition is populated during L3 (policy gating) and enforced throughout subsequent layers.

5.6. C5: Execution

Contains runtime execution parameters: budgets (token limits, cost ceilings), timeouts, concurrency limits, the tool registry, model routing preferences, and execution mode (e.g., "restricted-readonly", "full", "analysis-only").

5.7. C6: Audit

Contains audit configuration: log level, required evidence types, compliance checkpoint identifiers, and references to external audit sinks. This partition governs what level of audit detail is emitted by each layer and what evidence MUST accompany the final result.

5.8. Serialization

When serialized, the context bundle SHOULD be represented as a JSON [RFC8259] object with top-level keys corresponding to the partition identifiers (C0 through C6). The following is an illustrative example of a minimal context bundle:

```
{
  "C0": {
    "source": "chat",
    "ts": "2026-01-26T12:00:00Z",
    "channel": "web",
    "corr": "01AOCL_CORR_0001"
  },
  "C1": {
    "user_id": "user-42",
    "org_id": "org-7",
    "roles": ["operator"],
    "redact": ["ssn", "credit_card"]
  },
  "C2": {
    "goal": "Check backup status for production cluster",
    "priority": "high"
  },
  "C3": {},
  "C4": {
    "allowed_tools": ["tool.readonly.*"],
    "model_restrictions": []
  },
  "C5": {
    "timeout_ms": 30000,
    "max_parallel": 4,
    "mode": "restricted-readonly"
  },
  "C6": {
    "log_level": "info",
    "evidence_required": true
  }
}
```

6. Layer Taxonomy

AOCL defines eleven canonical layers (L0 through L10). Implementations are not required to use these exact names, but adopting the shared taxonomy makes stacks portable, debuggable, and interoperable across different orchestrator implementations.

Each layer is described below with its canonical identifier, purpose, input contract, output contract, and key behaviors.

6.1. L0: ingress.normalize

Canonical ID L0.ingress.normalize

Purpose Normalize incoming events from heterogeneous sources (chat

messages, webhooks, emails, monitoring alerts, scheduled ticks) into a uniform internal representation. Assign run identifiers and perform basic input parsing and validation.

Input Contract Raw event data in source-specific format. No assumptions about structure or encoding.

Output Contract Populated C0 (Event) partition with normalized source, timestamp, channel, correlation ID, and parsed attachments. A unique run_id MUST be generated for the processing run.

Key Behaviors

* MUST assign a unique run_id for the processing run.
* MUST populate C0.corr with a correlation identifier (either from the incoming event or newly generated).
* SHOULD validate that the incoming event contains sufficient data to proceed.
* MAY reject malformed events by setting a halt control flag.

6.2. L1: identity.scope

Canonical ID L1.identity.scope

Purpose Resolve and apply identity context, including user and organization identifiers, role assignments, permission sets, secret scope boundaries, and data redaction rules.

Input Contract C0 (Event) partition populated with source and correlation data. Authentication tokens or identity assertions from the incoming event.

Output Contract Populated C1 (Identity and Scope) partition. Identity resolution failures MUST result in a halt control flag or a forced restricted-mode branch.

Key Behaviors

* MUST resolve the acting identity before allowing downstream processing.
* MUST populate redaction rules so that downstream layers can enforce data minimization.
* SHOULD NOT be bypassable in production deployments.

* MAY support multiple identity providers and federation schemes.

6.3. L2: route.smart

Canonical ID L2.route.smart

Purpose Deterministic fast-path router that handles pattern matching, cached answers, known commands, and simple lookups without invoking the full pipeline. This layer provides an early exit for requests that do not require planning, context retrieval, or agent delegation.

Input Contract C0 (Event) and C1 (Identity) partitions populated.

Output Contract If a fast-path match is found: a direct response and halt_pipeline control flag set to true, causing the pipeline to skip to L9 (response assembly). If no match: context passed through unmodified.

Key Behaviors

* SHOULD be fast and deterministic (no LLM calls).
* MAY maintain a cache of previously computed responses.
* MUST set halt_pipeline to true when providing a direct response, to avoid unnecessary downstream processing.
* SHOULD emit a decision record indicating whether a fast-path match was found.

6.4. L3: policy.gate

Canonical ID L3.policy.gate

Purpose Enforce safety and compliance policies. Evaluate tool and model restrictions, content filtering requirements, and human-in-the-loop (HITL) thresholds. This layer acts as the primary governance checkpoint.

Input Contract C0 (Event), C1 (Identity), and C2 (Task) partitions populated (C2 may be partially populated at this stage).

Output Contract Populated C4 (Policy) partition with active constraints. May set control flags including require_hitl, branch_to (e.g., restricted mode), or halt_pipeline (for policy violations).

Key Behaviors

* MUST evaluate all applicable policies before allowing downstream execution.
* MUST emit a decision record with coded reasons (e.g., POLICY_ALLOW, POLICY_DENY, POLICY_RESTRICT).
* SHOULD NOT be bypassable in production deployments.
* MAY trigger HITL requirements by setting the require_hitl control flag.
* SHOULD populate C4.allowed_tools to constrain which tools the execution layer may invoke.

6.5. L4: plan.decompose

Canonical ID L4.plan.decompose

Purpose Convert the incoming intent into structured objectives. Determine whether the task requires single-step or multi-step execution, whether multiple agents are needed, and what the dependency graph looks like.

Input Contract C0 (Event), C1 (Identity), C2 (Task -- partial), and C4 (Policy) partitions populated.

Output Contract Fully populated C2 (Task) partition with structured objectives, sub-task decomposition (if applicable), dependency ordering, and a definition of done.

Key Behaviors

* SHOULD decompose complex requests into discrete, independently verifiable sub-tasks.
* MUST respect policy constraints from C4 when formulating plans (e.g., if tool execution is restricted, plans MUST NOT include tool-dependent steps).
* MAY use LLM inference to analyze intent and generate plans.
* SHOULD emit a decision record describing the decomposition strategy chosen.

6.6. L5: context.retrieve

Canonical ID L5.context.retrieve

Purpose Retrieve relevant memory, files, and knowledge context. Perform retrieval-augmented generation (RAG) lookups, memory searches, file reads, and produce bounded summaries and references.

Input Contract C2 (Task) partition with structured objectives to guide retrieval. C1 (Identity) partition for access control on retrieved data.

Output Contract Populated C3 (Memory and Knowledge) partition with retrieved data, references, and summaries. Large objects SHOULD be stored as references (URIs or content hashes) rather than inline values.

Key Behaviors

* MUST respect identity and permission boundaries from C1 when retrieving data.

* SHOULD produce bounded summaries to avoid context window overflow in downstream LLM calls.

* SHOULD emit references and digests for large retrieved objects rather than embedding them inline.

* MAY query multiple knowledge sources (vector stores, databases, file systems, APIs).

6.7. L6: shape.rewrite

Canonical ID L6.shape.rewrite

Purpose Rewrite and structure the accumulated context into an operational form suitable for execution. This may involve constructing AEE task envelopes, tool invocation plans, prompt templates, or structured schemas.

Input Contract C2 (Task), C3 (Memory), C4 (Policy), and C5 (Execution) partitions populated.

Output Contract Operational artifacts ready for L7 delegation: structured task definitions, tool call specifications, prompt payloads, or AEE task envelopes.

Key Behaviors

* MUST apply redaction rules from C1 to any data included in operational artifacts.

* SHOULD structure output for efficient consumption by the target execution runtime.

* MAY apply prompt engineering, template expansion, or schema mapping.

* SHOULD emit a decision record if the rewrite significantly transforms the original intent.

6.8. L7: delegate.execute

Canonical ID L7.delegate.execute

Purpose Delegate work to agents, tools, or external services. Manage dependencies between sub-tasks, enforce concurrency limits, and collect results. This is the primary execution layer and is intentionally runtime-specific.

Input Contract Operational artifacts from L6. C5 (Execution) partition with budgets, timeouts, concurrency limits, and tool registry.

Output Contract Execution results from delegated work, including agent responses, tool outputs, error reports, and evidence artifacts.

Key Behaviors

* MUST respect budgets and timeouts from C5.

* MUST respect the allowed tool list from C4.

* MUST emit AEE task envelopes when delegating to agents, preserving the correlation ID from C0.

* SHOULD manage concurrent sub-task execution within the limits specified in C5.

* MUST collect and propagate errors in a structured format.

6.9. L8: verify.check

Canonical ID L8.verify.check

Purpose Perform verification, evaluation, and consistency checks on execution results. Validate that evidence requirements are met, outputs are consistent with the task definition, and no policy violations occurred during execution.

Input Contract Execution results from L7. C2 (Task) partition with definition of done. C6 (Audit) partition with evidence requirements.

Output Contract Verification verdict (pass, fail, or partial) with supporting evidence. If verification fails, MAY trigger re-execution (loop back to L7) or halt with an error.

Key Behaviors

* MUST check that required evidence (as specified in C6) is present and valid.
* SHOULD validate output consistency against the task definition in C2.
* MUST emit a verification decision record with pass/fail status and reasons.
* MAY invoke evaluation models or deterministic validation logic.
* MAY request re-execution if results are insufficient, subject to retry limits.

6.10. L9: assemble.respond

Canonical ID L9.assemble.respond

Purpose Assemble the final response from execution results and verification outcomes. Apply final redaction, tone adjustment, and formatting constraints before delivery to the requesting party.

Input Contract Verified execution results from L8 (or direct results from L2 in fast-path cases). C1 (Identity) partition with redaction rules.

Output Contract Final response payload ready for delivery. All redaction rules from C1 MUST have been applied.

Key Behaviors

* MUST apply all redaction rules from C1 before emitting the response.
* SHOULD apply tone and formatting constraints appropriate to the output channel.
* MAY combine results from multiple sub-tasks into a coherent response.

* MUST NOT include sensitive data that was marked for redaction in C1.

6.11. L10: audit.writeback

Canonical ID L10.audit.writeback

Purpose Persist the complete audit trace for the processing run. Write back permitted memory updates (e.g., conversation summaries, learned facts) and emit the final run summary record.

Input Contract Complete context bundle with all accumulated deltas. C6 (Audit) partition with log level and compliance checkpoint requirements.

Output Contract Persisted audit trace (as one or more AEE event envelopes). Optional memory writeback artifacts. An aocl.run.summary event envelope summarizing the complete processing run.

Key Behaviors

* MUST persist the audit trace to the configured audit sink(s).
* MUST emit an aocl.run.summary envelope with timing, layer count, decision summary, and outcome status.
* SHOULD perform memory writeback only for data explicitly permitted by policy (C4).
* MUST NOT fail silently -- audit persistence failures SHOULD be reported as errors.

7. Stack Definition Format

AOCL stacks define the arrangement and configuration of layers for a particular class of work. Two execution modes are supported: pipeline mode and DAG mode.

7.1. Pipeline Mode

Pipeline mode arranges layers in a strict sequential order. Each layer executes after the previous one completes, unless a control flag causes a skip, branch, or halt. This is the simplest execution mode and is RECOMMENDED for most use cases.

A pipeline stack definition MUST include a stack_id, version, mode set to "pipeline", and an ordered array of layer entries. Each layer entry MUST include an id and ref field, and MAY include an enabled flag.

```
{
  "stack_id": "default",
  "version": "0.1",
  "mode": "pipeline",
  "layers": [
    {"id": "L0.ingress.normalize", "ref": "builtin:l0.normalize",
     "enabled": true},
    {"id": "L1.identity.scope",    "ref": "builtin:l1.identity",
     "enabled": true},
    {"id": "L2.route.smart",       "ref": "builtin:l2.router",
     "enabled": true},
    {"id": "L3.policy.gate",       "ref": "builtin:l3.policy",
     "enabled": true},
    {"id": "L4.plan.decompose",    "ref": "builtin:l4.plan",
     "enabled": true},
    {"id": "L5.context.retrieve",  "ref": "builtin:l5.context",
     "enabled": true},
    {"id": "L6.shape.rewrite",     "ref": "builtin:l6.shape",
     "enabled": true},
    {"id": "L7.delegate.execute",  "ref": "builtin:l7.delegate",
     "enabled": true},
    {"id": "L8.verify.check",      "ref": "builtin:l8.verify",
     "enabled": true},
    {"id": "L9.assemble.respond",  "ref": "builtin:l9.respond",
     "enabled": true},
    {"id": "L10.audit.writeback",  "ref": "builtin:l10.audit",
     "enabled": true}
  ],
  "defaults": {
    "bypass_allowed_for_roles": ["admin"],
    "max_parallel_actions": 8,
    "timeout_ms": 60000
  }
}
```

The "ref" field is an implementation pointer (plugin ID, module path, container image, etc.). AOCL does not define ref resolution semantics; it standardizes only the stack structure and layer input/output contracts.

7.2. DAG Mode

DAG mode arranges layers as nodes in a directed acyclic graph with conditional edges. This mode supports conditional routing, parallel branches, fast-path exits, and restricted execution paths.

A DAG stack definition MUST include a stack_id, version, mode set to "dag", an array of nodes, and an array of edges. Each edge MUST include "from" and "to" fields, and MAY include a "when" condition.

```
{
  "stack_id": "default-dag",
  "version": "0.1",
  "mode": "dag",
  "nodes": [
    {"id": "L0.ingress.normalize",
     "ref": "builtin:l0.normalize"},
    {"id": "L1.identity.scope",
     "ref": "builtin:l1.identity"},
    {"id": "L2.route.smart",
     "ref": "builtin:l2.router"},
    {"id": "L3.policy.gate",
     "ref": "builtin:l3.policy"},
    {"id": "L5.context.retrieve",
     "ref": "builtin:l5.context"},
    {"id": "L7.delegate.execute",
     "ref": "builtin:l7.delegate"},
    {"id": "L9.assemble.respond",
     "ref": "builtin:l9.respond"},
    {"id": "L10.audit.writeback",
     "ref": "builtin:l10.audit"},
    {"id": "BR.realtime_alert",
     "ref": "builtin:branch.alert_fast"},
    {"id": "BR.restricted_mode",
     "ref": "builtin:branch.restricted"}
  ],
  "edges": [
    {"from": "L0.ingress.normalize",
     "to": "L1.identity.scope"},
    {"from": "L1.identity.scope",
     "to": "L2.route.smart"},
    {"from": "L2.route.smart",
     "to": "L9.assemble.respond",
     "when": "control.halt_pipeline == true"},
    {"from": "L2.route.smart",
     "to": "L3.policy.gate",
     "when": "control.halt_pipeline != true"},
    {"from": "L3.policy.gate",
```

```
        "to": "BR.restricted_mode",
        "when": "control.require_hitl == true"},
       {"from": "L3.policy.gate",
        "to": "L5.context.retrieve",
        "when": "control.require_hitl != true"},
       {"from": "L5.context.retrieve",
        "to": "L7.delegate.execute"},
       {"from": "L7.delegate.execute",
        "to": "L9.assemble.respond"},
       {"from": "L9.assemble.respond",
        "to": "L10.audit.writeback"}
     ],
     "defaults": {
       "max_parallel_actions": 8,
       "timeout_ms": 60000
     }
   }
```

The "when" field is expressed as a string condition. In this version of the specification, implementations MAY treat it as simple boolean expressions over "context" and "control" namespaces. A formal expression language is deferred to future revisions.

Branch nodes (prefixed with "BR.") MAY represent sub-stacks that contain their own layer sequences. The resolution of branch node internals is implementation-defined.

7.3. Variant Stacks

Implementations are encouraged to define variant stacks tailored to specific workload classes. The following variants are RECOMMENDED:

realtime-alert (Speed-First) Prioritizes latency over thoroughness. Typical flow: normalize, identity, policy, delegate, respond, audit. Skips heavy planning and context retrieval unless explicitly required.

deep-research (Evidence-First) Prioritizes evidence quality and verification. Adds stronger context retrieval (multi-source), always-on verification and evaluation, and tighter evidence requirements in the audit partition.

restricted-textonly (Safety-First) Disables tool execution entirely. The policy gate forces text-only mode, and delegation becomes analysis-only or defers to human-in-the-loop decision making.

8. AEE Binding

AOCL is designed to work alongside the Agent Envelope Exchange (AEE) protocol [AEE]. AEE standardizes the message envelope for agent-to-agent and human-to-agent exchange, providing identity, intent, and causality semantics. AOCL standardizes the layered control pipeline that an orchestrator runs to decide what happens.

This binding defines how AOCL uses AEE without requiring any changes to the AEE envelope specification. AOCL operates entirely through intent conventions and payload schemas within existing AEE envelope types.

8.1. Envelope Types Used

AOCL uses existing AEE envelope type categories:

task Emitted when AOCL delegates work to an agent or worker.

result Received when an agent or worker returns completed work.

event Emitted when AOCL produces an audit record (layer enter, exit, or decision).

stream Optionally emitted for incremental traces or long-running layer output.

error Emitted when a layer or delegated work fails in a structured way.

AOCL does not require any new AEE type values.

8.2. Intent Namespace

AOCL defines the following intent namespace for layer-level audit and control. All intents use the "aocl." prefix:

Intent	Description
aocl.stack.select	Stack selection decision
aocl.layer.enter	Layer activation (entry)
aocl.layer.exit	Layer completion (exit)
aocl.layer.decision	Layer decision with coded reasons
aocl.context.patch	Context bundle modification (delta)
aocl.control.branch	Branch decision (alternate path taken)
aocl.control.bypass	Layer bypass decision
aocl.verify.result	Verification outcome
aocl.run.summary	Complete run summary

Table 1

For business-level delegation, AOCL does not impose intent names. Delegated tasks use the caller's intent (e.g., "ops.backup.status.check") or a derived intent convention appropriate to the domain.

8.3. Correlation Strategy

All envelopes generated during processing of a single work item MUST share the same "corr" (correlation) value as the originating request envelope. This enables grep-able traces across layer events, agent tasks and results, verification records, and final responses.

AOCL supports two reply-chaining strategies for the "reply_to" field:

Strategy A: Root-linked (RECOMMENDED) All AOCL layer events set reply_to to the originating request envelope ID. All delegated tasks also set reply_to to the originating request. This is the simplest strategy and is RECOMMENDED for initial implementations. Ordering is inferred from timestamps.

Strategy B: Span-linked Each layer event sets reply_to to the

previous layer event, forming a strict causal chain. Delegated tasks set reply_to to the layer decision event that emitted the task. This strategy provides explicit causal ordering suitable for APM-like trace visualization, at the cost of more complex envelope management.

Implementations SHOULD choose one strategy and apply it consistently within a deployment. Mixing strategies within a single processing run is NOT RECOMMENDED.

8.4. Payload Patterns

AOCL layer envelopes SHOULD NOT contain the full context bundle. Instead, payloads SHOULD include only the following fields:

run_id Orchestration run identifier.

layer Object containing "id" and "version" of the layer.

decisions Array of objects, each with "code" and "reason" fields.

delta Context patch or merge delta (small, not full context).

refs Array of string references to large data objects.

digests Map of digest names to digest values for integrity verification and replay support.

control Control flags (halt_pipeline, require_hitl, branch_to, bypass_layers).

timing_ms Optional layer execution duration for observability.

The following example shows an AOCL layer decision emitted as an AEE event envelope:

```
{
  "v": "1",
  "id": "01AOCL_LAYER_EVENT_0001",
  "ts": "2026-01-26T12:00:01Z",
  "type": "event",
  "from": "agent.orchestrator",
  "to": "log.aocl",
  "intent": "aocl.layer.decision",
  "corr": "01AOCL_CORR_0001",
  "reply_to": "01ORIGIN_TASK_0001",
  "trace": null,
  "priority": "normal",
  "requires": null,
  "payload": {
    "run_id": "RUN-01AOCL-0001",
    "layer": {
      "id": "L3.policy.gate",
      "version": "0.1"
    },
    "decisions": [
      {
        "code": "POLICY_ALLOW",
        "reason": "No restricted content; tools allowed: read-only"
      }
    ],
    "delta": {
      "C4.Policy.allowed_tools": ["tool.readonly.*"],
      "C5.Execution.mode": "restricted-readonly"
    },
    "refs": [],
    "digests": {
      "context_in": "sha256:abc123...",
      "context_out": "sha256:def456..."
    },
    "control": {
      "halt_pipeline": false
    }
  },
  "sig": null
}
```

The following example shows an AOCL delegation emitted as an AEE task envelope:

```
{
  "v": "1",
  "id": "01AOCL_DELEGATE_TASK_0001",
  "ts": "2026-01-26T12:00:03Z",
  "type": "task",
  "from": "agent.orchestrator",
  "to": "agent.backup_auditor",
  "intent": "ops.backup.status.check",
  "corr": "01AOCL_CORR_0001",
  "reply_to": "01ORIGIN_TASK_0001",
  "trace": null,
  "priority": "high",
  "requires": {
    "timeout_ms": 30000,
    "evidence": true
  },
  "payload": {
    "cluster_ref": "inv://clusters/node.lan",
    "window": "24h",
    "context_refs": [
      "mem://run/RUN-01AOCL-0001/context-snapshot"
    ]
  },
  "sig": null
}
```

Note that the "to" field is a logical sink identifier (e.g., "log.aocl", "agent.backup_auditor") that MAY map to a file, message queue topic, streaming subject, or any other transport endpoint. The AOCL semantics are carried entirely in the intent and payload fields, while the AEE envelope structure remains stable and unchanged.

9. Bypass and Branch Decisions

AOCL supports two forms of control-flow deviation: bypasses (skipping one or more layers) and branches (taking an alternate execution path). Both forms are permitted but MUST be explicitly audited.

9.1. Bypass Requirements

When a layer bypass is requested or applied, the orchestrator MUST emit an AEE event envelope with intent "aocl.control.bypass" containing:

* The identity of the requester (from C1);

* The list of layers that were skipped;

* Whether the bypass was allowed or denied;

* The policy or rule that permitted the bypass.

Stacks SHOULD declare bypass policy in their configuration. The following example shows a bypass policy declaration:

```
{
  "bypass_policy": {
    "allowed_roles": ["admin"],
    "never_bypass": [
      "L1.identity.scope",
      "L3.policy.gate"
    ],
    "audit_required": true
  }
}
```

Implementations SHOULD NOT allow bypass of the identity layer (L1.identity.scope) or the policy layer (L3.policy.gate) in production deployments. If bypass of these layers is permitted (e.g., in development or testing environments), the orchestrator MUST still emit a bypass audit record.

9.2. Branch Requirements

When the orchestrator takes a branch (alternate execution path), it MUST emit an AEE event envelope with intent "aocl.control.branch" containing:

* The layer or node that initiated the branch;

* The target branch destination;

* The reason the branch was taken (e.g., policy restriction, fast-path match, HITL requirement).

Branch decisions are typically emitted by the smart router (L2) or the policy gate (L3), but any layer MAY produce branch control flags if the stack configuration permits it.

9.3. Audit Completeness

The combination of bypass and branch audit records, together with layer enter/exit events, MUST provide a complete and reconstructable picture of which layers were executed, which were skipped, and which alternate paths were taken for every processing run. An observer with access to the audit trail MUST be able to reconstruct the exact execution path without any implicit or hidden state transitions.

10. Security Considerations

AOCL is a control-plane protocol that governs how AI agent operations flow through identity, policy, execution, and audit stages. The security properties of a deployment depend critically on the integrity of each layer and the trustworthiness of the orchestrator. This section describes the primary security considerations.

10.1. Policy Enforcement Integrity

The policy gate (L3.policy.gate) is the primary governance checkpoint. If an attacker can manipulate, bypass, or disable this layer, they can cause the orchestrator to execute actions that violate organizational policies. Implementations MUST ensure that the policy layer cannot be circumvented through context manipulation, layer reconfiguration, or control-flag injection.

Policy rules SHOULD be loaded from a trusted, integrity- protected source. Changes to policy configuration SHOULD require authenticated authorization and SHOULD themselves be audited.

10.2. Bypass Audit Trail Requirements

Layer bypasses represent a significant security-relevant event. The bypass audit trail is the primary mechanism for detecting unauthorized control-flow deviations. Implementations MUST ensure that:

* Bypass audit records cannot be suppressed, deleted, or modified after emission;

* Bypass audit records are written to a separate, append- only audit sink when possible;

* Bypass of identity (L1) and policy (L3) layers is prohibited in production unless explicitly authorized by a privileged role and documented in the bypass policy configuration;

* All bypass records include the authenticated identity of the requester, not merely a self-asserted identity.

10.3. Identity Trust Model

The identity layer (L1.identity.scope) populates the C1 partition that all subsequent layers rely on for access control decisions. If the identity layer produces incorrect or forged identity information, all downstream policy enforcement and access control becomes unreliable.

Implementations MUST use authenticated identity sources (e.g., verified tokens, mutual TLS, signed assertions) rather than self-reported identity claims. The identity layer SHOULD validate identity assertions against a trusted identity provider before populating C1.

When AOCL operates in a multi-tenant environment, implementations MUST ensure that identity scoping prevents cross-tenant data access through context bundle isolation or equivalent mechanisms.

10.4. Layer Isolation

Layers SHOULD be isolated from one another to prevent a compromised or malfunctioning layer from corrupting the context bundle or control flags of other layers. The orchestrator SHOULD validate context bundle modifications against expected schemas and reject unexpected mutations.

When layers are implemented as external services or plugins, the orchestrator SHOULD authenticate layer implementations and validate their outputs. A malicious layer implementation could inject false decisions, corrupt the context bundle, or emit misleading audit records.

The delegation layer (L7.delegate.execute) poses particular isolation challenges because it invokes external agents and tools. Implementations SHOULD sandbox delegated execution and validate returned results against expected schemas before incorporating them into the context bundle.

10.5. Denial of Service through Layer Abuse

An attacker who can submit intents to the orchestrator may attempt to cause denial of service by:

* Submitting events that trigger expensive layers (e.g., large-scale context retrieval in L5 or unbounded delegation in L7);

* Crafting inputs that cause verification loops between L7 and L8;

* Exploiting DAG edge conditions to create long or resource-intensive execution paths;

* Generating excessive audit records to overwhelm the audit sink.

Implementations SHOULD enforce resource budgets at the stack level (timeout_ms, max_parallel_actions) and at individual layer level. The orchestrator SHOULD implement rate limiting on incoming intents and circuit breakers on layers that exhibit abnormal latency or error rates. Verification loops (L7 to L8 retry cycles) MUST be bounded by a configurable maximum retry count.

10.6. Context Bundle Confidentiality

The context bundle may contain sensitive information across multiple partitions: authentication credentials in C1, proprietary task details in C2, retrieved knowledge in C3, and policy rules in C4. Implementations MUST protect the context bundle from unauthorized access during transmission between layers and at rest.

When layers are distributed across network boundaries, the context bundle MUST be transmitted over encrypted channels (e.g., TLS). When layers are implemented by third-party services, the orchestrator SHOULD apply data minimization by providing only the context partitions that a layer requires, rather than the full bundle.

Audit records emitted as AEE envelopes contain context deltas that may include sensitive data. Implementations SHOULD apply redaction rules from C1 to audit payloads before persisting them, or ensure that audit sinks have equivalent access protections to the data sources referenced in the context bundle.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

12.2. Informative References

[AEE] Cowles, A., "Agent Envelope Exchange (AEE)", Work in Progress, Internet-Draft, draft-cowles-aee-00, 2026, <https://github.com/AdaminX/AEE-Agent-Envelope-Exchange>.

[VOLT] Cowles, A., "Verifiable Operations Ledger and Trace (VOLT)", Work in Progress, Internet-Draft, draft-cowles-volt-00, 2026, <https://github.com/AdaminX/VOLT-Protocol>.

Acknowledgements

The author wishes to thank the early adopters and reviewers of the AOCL specification within the Quox ecosystem, whose feedback on real-world agent orchestration challenges shaped the design of this control-layer protocol.

Author’s Address

Adam Cowles
Quox Ltd
London
United Kingdom
Email: adam@quox.ai
URI: https://quox.ai