Cognous Agent Control Plane: Runtime Control, Replay, and Public-Safe Governance for AI-Agent Workflows

André de Lima
Cognous
Seattle, Washington
andredelima@cogno.us

July 2026

Abstract

AI agents increasingly operate through tools, APIs, databases, files, and enterprise systems. This creates a runtime control problem that ordinary chatbot frameworks do not solve: before an agent acts, an enterprise needs to know what the agent proposed, whether the action was authorized, which policy allowed, blocked, or escalated it, what external sources were relied upon, whether the decision path can be inspected, and whether the run can be reconstructed after the fact. Cognous Agent Control Plane is a minimal public reference implementation for runtime control and replay of AI-agent workflows. It records action proposals, policy decisions, policy evaluation traces, blocked actions, authority records, reliance records, replay bundles, signed replay bundles, policy configuration artifacts, JSON-schema-governed records, semantic validation reports, redacted exports, command-line validation workflows, tool-adapter execution results, and filesystem-persisted run artifacts. This whitepaper specifies the public reference architecture, current implementation boundary, object model, policy-gate semantics, lightweight policy configuration layer, audit/replay workflow, schema role, semantic validation boundary, redaction model, tool-adapter execution pattern, persistence adapter, test status, and extension path. It is intentionally scoped: it is not an agent framework, not a model runtime, not a complete enterprise governance platform, not production key management, and not a substitute for application-level authorization. Its contribution is narrower and operational: a small, inspectable control layer that makes agent runs easier to authorize, audit, validate, redact, persist, sign, and replay.

1 Introduction

AI-agent systems are moving from conversational assistance toward delegated execution. Agents now call tools, retrieve records, draft messages, update systems, invoke APIs, and coordinate workflows across enterprise boundaries. This shift changes the governance problem. It is no longer sufficient to inspect final text. The material question becomes whether the agent's proposed actions were governed before they affected external systems.

A deployed agent run typically contains several distinct events:

1. a task is framed;
2. lightweight policy configuration is applied;
3. an agent proposes an action;

4. a policy decision allows, blocks, or escalates the action;
5. the policy decision may produce an evaluation trace;
6. authority is checked for privileged action types;
7. the agent may rely on a tool, file, database, model output, or user input;
8. a final output or operational result is produced;
9. the run may need to be audited, semantically validated, signed, persisted, redacted, or replayed.

Most agent frameworks focus on orchestration. Most model runtimes focus on inference. Most logging systems record events after they occur. Agent Control Plane occupies a narrower layer: it records the governance surface around proposed agent actions before those actions are treated as executable.

The guiding design principle is:

> No material agent action should occur without a record of what was proposed, what policy decided, how that decision was reached, what authority existed, what reliance was introduced, and how the run can be inspected later.

The public reference implementation exposes a small set of public objects and utilities: `Frame`, `ActionProposal`, `PolicyDecision`, `PolicyRuleEvaluation`, `PolicyEvaluationTrace`, `BlockedAction`, `AuthorityRecord`, `RelianceRecord`, `RunRecord`, `ReplayBundle`, `SignedReplayBundle`, `PolicyConfig`, `ValidationIssue`, `ValidationReport`, `RedactionConfig`, `ToolExecutionResult`, policy-configuration helpers, validation helpers, redaction helpers, a command-line interface, JSON schemas, signing utilities, tool-adapter execution helpers, and a filesystem persistence adapter.

2 Problem Statement

Enterprise AI-agent adoption creates a runtime control gap. The system may know that an agent produced a result, but it may not know, in a structured and replayable way, what the agent attempted to do along the way.

This gap has several operational forms.

2.1 Proposed Versus Executed Actions

An agent may propose an action that is not executed. Without explicit action-proposal records, blocked or rejected behavior disappears from the audit trail. The absence of a bad outcome is not the same as evidence that a risky action was blocked.

2.2 Authority Ambiguity

Agents operate under delegated authority. Some actions are harmless reads. Others are writes, sends, deletes, purchases, updates, or escalations. A system needs an explicit record of which actor had which scoped authority at the time of decision.

2.3 Opaque Policy Decisions

A policy decision that merely says `allow`, `block`, or `escalate` may be insufficient for audit review. Reviewers often need to know which rule matched, which rules did not match, and whether the decision fingerprint corresponds to the recorded trace.

2.4 Opaque Reliance

Agent outputs often depend on external context: retrieved documents, databases, tool responses, user input, prior model outputs, or API results. If reliance is not recorded, later review cannot distinguish unsupported generation from source-grounded execution.

2.5 Unreplayable Runs

When a run cannot be replayed or reconstructed, investigation becomes forensic guesswork. Auditors need a portable bundle containing the frame, actions, decisions, policy traces, authority records, reliance records, blocked actions, and final output.

2.6 Configuration Drift

A policy gate is only useful if its applied configuration is explicit. Allowed tools, blocked tools, default action posture, and required authority scopes must be represented in a repeatable way rather than scattered through application code.

2.7 Schema Ambiguity

A public reference layer requires public object boundaries. JSON schemas provide a stable contract for run records, action proposals, policy decisions, policy traces, authority records, reliance records, validation reports, redaction configuration, tool execution results, replay bundles, and signed replay bundles.

2.8 Unsafe Export

Run records and replay bundles may contain sensitive targets, payloads, reasons, or final outputs. A control layer should support redacted exports that preserve governance structure while removing sensitive content.

3 Design Goals

Agent Control Plane is designed around ten goals.

1. **Pre-action recording.** Proposed actions are recorded before execution.
2. **Deterministic policy gating.** Identical decision inputs produce the same policy result and deterministic fingerprint.
3. **Traceable policy decisions.** Policy decisions can include ordered evaluation traces.
4. **Blocked-action visibility.** Blocked actions create first-class records.
5. **Authority scoping.** Authority is scoped by run, actor, permission, source, and optional expiry.
6. **Reliance tracking.** External-source dependence is recorded as part of the run.
7. **Replayability.** Completed runs can be exported as replay bundles.
8. **Configuration portability.** Lightweight policy configuration can be represented outside Python code.
9. **Safe export.** Records can be redacted and signed for controlled downstream review.
10. **Safe extensibility.** Validation, persistence, adapters, and future replay tooling can be added without turning the package into a full enterprise platform.

4 Non-Goals

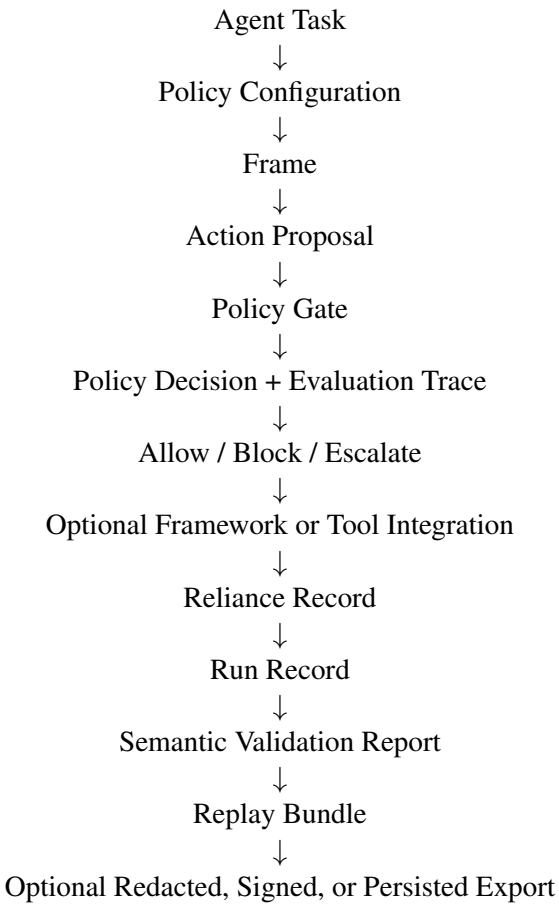
The architecture is deliberately bounded. Agent Control Plane is not:

- an agent framework;
- a model runtime;
- a planner;
- a dashboard;
- a full policy authoring system;
- a complete compliance product;
- a production key-management system;
- a security boundary by itself;
- a substitute for application-level authorization;
- a guarantee that model outputs are true, safe, or complete.

It enforces only the policy rules and authority records supplied to it. If the configured policies are incomplete, overly permissive, or incorrect, the control plane will faithfully record and apply those policies, but it will not infer the missing governance intent.

5 System Overview

The runtime pattern is:



The system sits beside an agent. The agent still performs planning and generation. The control plane records, evaluates, validates, and packages the governance-relevant events around that agent's proposed actions.

6 Core Object Model

6.1 Frame

A `Frame` is the execution context for a run. It records the task, actor, environment, allowed tools, blocked tools, policy version, and creation time. The frame is immutable for the duration of a run.

Field	Meaning
`frame_id`	unique frame identifier
`task`	task description
`actor`	agent, user, or service identifier
`environment`	deployment context
`allowed_tools`	tools explicitly permitted
`blocked_tools`	tools explicitly blocked
`policy_version`	active policy version
`created_at`	creation timestamp

6.2 ActionProposal

An `ActionProposal` records a proposed tool or system action before execution. It includes the tool name, action type, target, payload, proposed time, and optional reason.

6.3 PolicyDecision

A `PolicyDecision` records the outcome of policy evaluation. The result is one of:

- `allow`;
- `block`;
- `escalate`.

The decision also includes the policy name, reason, timestamp, deterministic fingerprint, and optional trace identifier.

6.4 PolicyRuleEvaluation

A `PolicyRuleEvaluation` records the outcome of one rule check in the policy chain. It includes the rule name, whether the rule matched, the local result, and a human-readable reason.

6.5 PolicyEvaluationTrace

A `PolicyEvaluationTrace` records the ordered rule path for a decision. It includes the trace identifier, run identifier, action identifier, evaluation time, rules evaluated, final result, and deterministic fingerprint.

6.6 BlockedAction

A `BlockedAction` is created when a policy decision returns `block`. This ensures that blocked behavior is visible as a first-class event rather than disappearing from the run history.

6.7 AuthorityRecord

An `AuthorityRecord` grants scoped permission to an actor for a run. Authority records may expire. The policy gate considers only active authority records matching the current run and actor.

6.8 RelianceRecord

A `RelianceRecord` records dependence on an external source, such as a tool, database, file, API, user input, model output, or other source. Reliance records allow later review to identify what the agent depended on.

6.9 RunRecord

A `RunRecord` aggregates the frame, action proposals, policy decisions, policy evaluation traces, authority records, reliance records, blocked actions, final output, replay bundle identifier, and completion status for a single run.

6.10 ReplayBundle

A `ReplayBundle` packages the run into a self-contained object suitable for inspection, semantic validation, signing, redaction, persistence, or downstream audit review. It includes policy evaluation traces as part of the replayable governance surface.

6.11 SignedReplayBundle

A `SignedReplayBundle` stores a replay bundle alongside optional export-integrity metadata. The current reference implementation uses HMAC-SHA256 over deterministic JSON serialization of the replay bundle.

6.12 ValidationIssue and ValidationReport

A `ValidationIssue` records a warning or error discovered during semantic validation. A `ValidationReport` aggregates issues and marks the checked object as valid only when no error-severity issues are present.

6.13 RedactionConfig

A `RedactionConfig` controls privacy-safe export behavior. It can redact action payloads, action targets, final output, action reasons, policy decision reasons, blocked-action reasons, and trace-rule reasons while preserving identifiers and structural relationships.

6.14 ToolExecutionResult

A `ToolExecutionResult` records the outcome of attempting to execute an allowed action through a tool adapter. It includes the action identifier, run identifier, execution flag, result payload, and optional error.

7 Policy Gate Semantics

The policy gate evaluates action proposals against a frame and active authority records. The current reference implementation applies a deliberately small rule set.

No.	Condition	Result	Rationale
1	tool is explicitly blocked	block	explicit block list takes priority
2	tool is not in allowed tools	escalate	unknown tools require review
3	action type is `external_send` and no matching authority exists	block	outbound actions require explicit authority
4	action type is `read` and tool is allowed	allow	allowed read operations are permitted
5	action type is `write` and write authority exists	allow	write requires scoped authority
6	no rule matches	escalate	unknown cases are not silently allowed

The gate is deterministic with respect to result and fingerprint. The `decision_id` and `decided_at` fields are generated per evaluation and are therefore not intended to be stable across repeated evaluations.

7.1 Deterministic Fingerprint

The deterministic fingerprint is computed from canonical decision inputs, including:

- tool name;
- action type;
- target;
- sorted allowed tools;
- sorted blocked tools;
- policy version;
- sorted active authority scopes.

This yields a stable digest for identical decision inputs. The digest supports comparison and audit workflows, but it is not a cryptographic proof of correctness.

8 Policy Evaluation Traces

The public reference implementation supports `evaluate_with_trace()`, which returns both a `PolicyDecision` and a `PolicyEvaluationTrace`. The trace records the ordered rule path used to reach the decision.

Rules are traced in order:

1. `blocked_tool_policy`;
2. `unknown_tool_policy`;

3. external_send_authority_policy;

4. read_allowed_tool_policy;

5. write_authority_policy;

6. default_escalation_policy.

A trace gives reviewers more than the final decision. It shows which rule matched, which earlier rules did not match, what result was produced, and why. RunRecorder records policy traces by default and includes them in both RunRecord and ReplayBundle exports.

9 Lightweight Policy Configuration

The public reference implementation includes a lightweight JSON-backed policy configuration helper. This is not a full policy language. It is a small portability mechanism for expressing the basic policy inputs that create a frame and, where applicable, authority records.

9.1 PolicyConfig

A PolicyConfig contains:

Field	Meaning
policy_version	version string for the policy set
allowed_tools	list of tools permitted in the frame
blocked_tools	list of tools explicitly blocked in the frame
authority_required	mapping from action type to required authority scopes
default_action	fallback posture: allow, block, or escalate

The purpose of PolicyConfig is to make simple policy posture portable across examples and integrations. It should not be mistaken for a production policy DSL.

9.2 frame_from_policy_config()

The helper frame_from_policy_config() creates a Frame from a validated PolicyConfig. It maps the policy version, allowed tools, and blocked tools into the frame while requiring the caller to supply the task, actor, and environment.

This separates stable policy posture from run-specific context:

- policy configuration supplies allowed_tools, blocked_tools, and policy_version;
- the application supplies task, actor, and environment;
- the resulting Frame becomes the immutable execution context for that run.

9.3 authority_records_from_policy_config()

The helper authority_records_from_policy_config() creates AuthorityRecord objects from the authority_required mapping. The caller supplies the run identifier, actor, source, and optional expiry. This provides a simple bridge from configuration to runtime authority records without introducing a full policy language.

10 Authority and Reliance

Authority and reliance are distinct.

10.1 Authority

Authority answers the question:

> Was this actor permitted to perform this class of action in this run?

Authority records are scoped. A permission granted to one actor does not automatically apply to another actor. A permission granted in one run does not automatically apply to another run. Expired authority is ignored.

10.2 Reliance

Reliance answers the question:

> What external source did the agent depend on while producing or executing this run?

A reliance record does not grant permission. It records dependence. This distinction prevents a common failure: treating source access as if it were authorization.

11 Run Recording Workflow

The `RunRecorder` is the primary user-facing orchestration object. A typical workflow is:

1. start a run;
2. add authority records;
3. propose an action;
4. evaluate the action and record a policy trace;
5. create a blocked-action record if blocked;
6. execute only if allowed;
7. record reliance if a source or tool was used;
8. complete the run;
9. export the run record;
10. generate a replay bundle.

The run recorder enforces action/run consistency. A foreign action from another run is rejected. An action that was not proposed through the recorder is also rejected. This prevents accidental evaluation of untracked actions.

12 Framework Integration and Tool Adapter Execution

The repository includes two public-safe integration patterns.

12.1 Mock Framework Integration Demo

The mock framework integration demo is intentionally not a real LangChain, OpenAI Agents, or vendor-specific adapter. It is a public-safe pattern showing how Agent Control Plane can sit beside an agent or framework without becoming the framework.

The mock integration demonstrates the following sequence:

1. a mock agent proposes a list of intended actions;
2. each proposed action is recorded through `RunRecorder`;
3. the policy gate evaluates each proposal;
4. only allowed actions are treated as executable;
5. reliance is recorded for allowed tool use;
6. the run is completed;
7. a run record is exported;
8. a replay bundle is generated.

The purpose of the demo is architectural clarity, not framework coverage.

12.2 Tool Adapter Execution API

The `ToolAdapter` protocol and `execute_with_control()` helper provide a minimal tool-execution boundary.

The execution pattern is:

1. create an action proposal;
2. evaluate the proposal through the policy gate;
3. if the result is not `allow`, do not execute the adapter;
4. if the result is `allow` and no adapter is supplied, return a non-executed `ToolExecutionResult`;
5. if an adapter is supplied, execute it;
6. record reliance after successful tool execution;
7. return the policy decision and execution result.

Adapter exceptions are captured in `ToolExecutionResult.error`. This keeps the public pattern small and inspectable without turning Agent Control Plane into an agent framework.

13 Replay Bundles

A replay bundle is a portable representation of a completed run. It includes the frame, action proposals, policy decisions, policy evaluation traces, authority records, reliance records, blocked actions, and final output.

Replay bundles serve several use cases:

- post-run audit;
- incident review;
- debugging;
- policy regression testing;
- controlled export;
- downstream validation;
- redaction;
- signing and integrity verification;
- filesystem persistence.

A replay bundle does not guarantee that the original external world can be reconstructed. Tool outputs, database states, and files may change after the run. The bundle records the governance surface and run artifacts available to the control plane.

14 JSON Schemas and Public Conformance

The repository includes JSON Schema Draft 2020-12 files for the public record objects. These schemas define the public interchange boundary for records such as `Frame`, `ActionProposal`, `PolicyDecision`, `PolicyRuleEvaluation`, `PolicyEvaluationTrace`, `AuthorityRecord`, `RelianceRecord`, `ValidationIssue`, `ValidationReport`, `RedactionConfig`, `ToolExecutionResult`, `RunRecord`, `ReplayBundle`, and `SignedReplayBundle`.

The schemas serve four purposes:

1. **Interchange.** They allow external systems to understand the shape of exported records.
2. **Validation.** They support structural validation of records outside the Python package.
3. **Documentation.** They make the public object boundary explicit.
4. **Conformance.** They provide a basis for future compatibility tests across implementations.

The schema boundary is deliberately limited. JSON schemas validate structure, required fields, field types, and allowed values where specified. They do not prove that an agent's output is correct, that a policy is sufficient, that an action was safe, or that an enterprise compliance obligation has been satisfied.

15 Semantic Validation

The current reference implementation supports semantic validation reports for exported run records and replay bundles. These checks go beyond Pydantic parsing by verifying internal record consistency.

Validation checks include:

- every embedded action uses the correct run identifier;
- every policy decision uses the correct run identifier;
- decisions reference known actions;
- blocked actions reference known actions;
- blocked actions correspond to block decisions;
- reliance records reference known actions when a reference is present;
- authority records use the correct run identifier;
- authority actor mismatch with the frame actor produces a warning;
- policy traces use the correct run identifier;
- policy traces reference known actions;
- policy decision trace identifiers reference known traces;
- trace fingerprints match associated decision fingerprints when linked;
- completed runs without final output produce warnings.

A `ValidationReport` contains `ValidationIssue` records. Error-severity issues make the report invalid. Warning-severity issues indicate review concerns without necessarily invalidating the record.

This remains implementation-level validation. It does not prove factual correctness, compliance satisfaction, policy sufficiency, or safety of the underlying agent behavior.

16 Redacted Exports

Run records and replay bundles may contain sensitive targets, payloads, reasons, or final outputs. Redaction allows structural review without exposing sensitive content.

`RedactionConfig` supports:

- action payload redaction;
- action target redaction;
- final-output redaction;
- action-reason redaction;
- policy-decision-reason redaction;
- blocked-action-reason redaction;

- trace-rule-reason redaction.

Redaction returns model copies and does not mutate originals. Identifiers, timestamps, decision results, policy names, fingerprints, trace relationships, and structural relationships are preserved.

Fingerprints are not recomputed after redaction. The fingerprint corresponds to the original decision inputs, not the redacted export.

17 Signed Replay Bundles

Signed replay bundles provide optional export integrity metadata. The reference implementation uses HMAC-SHA256 over deterministic JSON serialization of the replay bundle.

The signing workflow is:

1. serialize the replay bundle deterministically;
2. compute HMAC-SHA256 using a caller-supplied secret;
3. attach signature metadata;
4. verify later by recomputing the signature and comparing digests.

This is useful for detecting modification after export. It is not production key management. It is not a full security boundary. A real deployment must decide how secrets are generated, stored, rotated, scoped, and protected.

18 Persistence

The reference implementation includes a minimal persistence adapter interface and a filesystem adapter.

The abstract interface supports:

- saving run records;
- loading run records;
- saving replay bundles;
- loading replay bundles.

The filesystem adapter stores records as JSON files under a local directory. In the reference implementation, run records and replay bundles are written to separate subdirectories, typically corresponding to local paths such as:

- `runs/{run_id}.json`;
- `replay_bundles/{replay_bundle_id}.json`.

This is suitable for tests, examples, demos, and small integrations. It is not a production storage architecture. Database adapters, object-store adapters, encryption at rest, retention policies, tenant isolation, and enterprise audit storage are out of scope for the minimal public reference implementation.

19 Command-Line Interface

The `acp` command-line interface provides small utilities for exported records.

19.1 Implemented Commands

The current public reference implementation includes:

- `acp validate-run path/to/run_record.json`;
- `acp validate-replay path/to/replay_bundle.json`;
- `acp redact-run input.json --out output.json`;
- `acp redact-replay input.json --out output.json`;
- `acp sign-replay path/to/replay_bundle.json --secret SECRET --out path/to/signed.js`
- `acp verify-signed-replay path/to/signed.json --secret SECRET`.

The redaction commands support optional flags for targets, final output, reasons, and replacement text. The validation commands use semantic validation reports rather than merely loading the models.

These commands are intended for local validation, examples, and lightweight workflows. They are not a hosted service, daemon, dashboard, or enterprise control surface.

20 Reference Implementation Status

The public reference implementation is intentionally small. It is meant to be inspectable, testable, and safe to publish.

Capability	Status	Notes
Core object model	Implemented	Pydantic v2 models
Policy gate	Implemented	deterministic result and fingerprint
Policy evaluation traces	Implemented	ordered rule-path records
Run recorder	Implemented	in-memory run accumulation
Blocked-action records	Implemented	auto-created on block
Authority records	Implemented	scoped by run and actor
Reliance records	Implemented	external-source tracking
Replay bundles	Implemented	portable run snapshot
Signed replay bundles	Implemented	HMAC-SHA256 export integrity metadata
PolicyConfig	Implemented	lightweight JSON-backed config
Policy authority helper	Implemented	config-to-authority records
Semantic validation reports	Implemented	internal consistency checks
Redacted exports	Implemented	payloads, targets, outputs, reasons
Redacted export CLI	Implemented	`redact-run`, `redact-replay`
Tool adapter execution	Implemented	execute only after allow decisions
Filesystem persistence	Implemented	local JSON storage
CLI validation/signing	Implemented	local command-line utility
Mock integration demo	Implemented	framework-pattern demonstration
Full framework adapters	Extension	out of current minimal scope
Hosted dashboard	Extension	out of current minimal scope
Production key management	Extension	out of current minimal scope

21 Test Suite and Continuous Integration

The repository includes a test suite covering the public reference behavior. Tests are not part of the architecture itself, but they are part of the public trust surface. They show which claims are exercised by executable checks.

The test suite covers areas such as:

- allow, block, and escalate decisions;
- authority requirements for outbound actions;
- frame immutability;
- blocked-action record creation;
- reliance record creation;
- replay bundle generation;
- policy evaluation trace generation;

- trace storage in run records and replay bundles;
- deterministic fingerprint behavior;
- JSON export round trips;
- policy configuration loading;
- authority-record creation from policy configuration;
- signing and signature verification;
- semantic validation reports;
- redaction helpers;
- tool adapter execution;
- filesystem persistence round trips;
- CLI validation, redaction, signing, and verification commands.

A GitHub Actions workflow runs the test suite across supported Python versions. This gives the public reference implementation a basic compatibility and regression signal without implying production certification.

22 Security Considerations

Agent Control Plane improves observability and control, but it should not be treated as a complete security product.

Important limitations include:

- the policy gate enforces supplied rules, not unstated intent;
- policy traces explain rule-path behavior, not policy sufficiency;
- validation checks internal consistency, not factual truth or compliance satisfaction;
- signing depends on caller-managed secrets;
- redaction must be configured correctly;
- filesystem persistence may need encryption and retention controls in real deployments;
- tool adapters must be integrated before execution, not after;
- adapter execution does not replace application authorization;
- production deployments still require authentication, authorization, logging, monitoring, and incident response.

The correct security posture is layered. Agent Control Plane supplies structured runtime records, gate decisions, traces, validation reports, and export utilities. It should be combined with application authorization, infrastructure security, model-risk controls, policy review, and operational monitoring.

23 Implementation Boundary

The public reference implementation is intentionally thin. It provides enough structure to demonstrate the category and support real integration experiments. It does not reveal or depend on private Cognous platform internals.

This boundary is strategic as well as technical. Public software should expose the integration contract: what developers can call, what records they can inspect, and what artifacts they can export. Private platform internals should remain private unless separately reviewed and deliberately published.

The public boundary is therefore:

Public Reference Layer	Private Product Layer
Action proposals	product-specific reasoning systems
Policy decisions	proprietary policy-generation methods
Policy traces	proprietary explainability and review tooling
Blocked actions	internal platform control logic
Authority records	customer-specific authority models
Reliance records	private retrieval and ranking systems
Replay bundles	proprietary replay and analysis tooling
Validation reports	enterprise compliance workflows
Redacted exports	internal data-classification systems
Signed exports	production key-management design
JSON schemas	enterprise conformance tooling
Filesystem persistence	production storage architecture
Tool adapter pattern	product-specific execution orchestration
Mock integration demo	product-specific framework integrations

24 Example Use Cases

24.1 Customer-Service Agent

A customer-service agent reads a CRM record and drafts an email. Reading the CRM is allowed. Sending an email is blocked unless explicit outbound authority exists. The run record captures both the allowed read and the blocked send. The policy trace distinguishes a tool block from a missing-authority block.

24.2 Internal Research Assistant

An internal assistant retrieves files and produces a summary. Each file reliance is recorded. The replay bundle allows a reviewer to inspect which sources the assistant used.

24.3 Workflow Automation

An agent proposes a database update. The policy gate escalates because write authority is missing. The escalated decision and trace are recorded before the update occurs.

24.4 Audit Export

A completed run is semantically validated, redacted, signed, and exported. The reviewer receives structure and decision metadata without sensitive payloads.

24.5 Controlled Tool Execution

An application uses a tool adapter to gate execution. The adapter is called only when the policy decision is `allow`. Successful execution creates a reliance record, and adapter exceptions are captured in a tool execution result.

25 Evaluation Criteria

The usefulness of Agent Control Plane should be evaluated against operational criteria rather than benchmark scores.

A deployment should ask:

1. Are proposed actions recorded before execution?
2. Are blocked actions visible?
3. Are policy decisions deterministic with respect to decision inputs?
4. Are policy evaluation traces available for review?
5. Are authority records scoped and expiring where appropriate?
6. Are external reliance points captured?
7. Can a completed run be exported as a replay bundle?
8. Can exported artifacts be semantically validated?
9. Can sensitive content be redacted while preserving structure?
10. Can replay bundles be signed and verified?
11. Can simple policy posture be loaded from configuration?
12. Can authority records be derived from configuration?
13. Can records be persisted locally through the reference adapter?
14. Can tool execution be gated without turning the control layer into an agent framework?
15. Can the control layer sit beside existing agent code without becoming the framework?

26 Roadmap

The public roadmap should remain implementation-focused and avoid overclaiming.

Implemented MVP extensions include:

- policy evaluation traces;
- semantic validation reports;
- redaction profiles and redacted export CLI commands;
- signed replay bundles;

- lightweight policy configuration;
- authority-record generation from policy configuration;
- filesystem persistence;
- tool adapter execution;
- mock framework integration examples.

Near-term extensions include:

- richer policy configuration;
- additional framework examples;
- additional persistence adapters;
- conformance tests for public schemas;
- richer tool-adapter examples;
- optional replay-viewer tooling.

Longer-term work may include dashboards, hosted policy services, enterprise integrations, and formal compliance workflows. Those are product-layer concerns and should remain separate from the minimal public reference implementation.

27 Conclusion

AI-agent governance cannot be limited to final-output review. As agents gain access to tools, APIs, files, and enterprise systems, governance must move closer to the moment of action. Cognous Agent Control Plane provides a minimal public reference architecture for that shift. It records proposed actions before execution, evaluates those actions through a policy gate, records policy evaluation traces, records authority and reliance, creates blocked-action records, packages runs into replay bundles, supports semantic validation reports, supports redacted exports, supports signed replay bundles, exposes a lightweight policy-configuration helper, publishes JSON schemas for the public record boundary, provides tool-adapter execution helpers, and includes a filesystem persistence adapter and command-line utilities for local workflows.

The contribution is deliberately narrow. Agent Control Plane does not solve all AI governance problems. It does not run agents, train models, author complex policies, or guarantee correctness. It supplies a concrete runtime control layer for a specific operational need: making agent runs inspectable, governable, redaction-ready, validation-ready, and replayable before delegated agent behavior becomes enterprise-critical.

References

[1] Cognous. *Agent Control Plane: Runtime control and replay for AI agents*. Public reference implementation, 2026. `https://github.com/cogno-us/cognous-agent-control-plane`

[2] National Institute of Standards and Technology. *Artificial Intelligence Risk Management Framework*. NIST AI 100-1, 2023.

[3] OWASP Foundation. *OWASP Top 10 for Large Language Model Applications*. Project documentation.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.