Cognous Agent Action Manifest: Declaring Governable AI-Agent Actions Before Runtime Execution

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

July 2026

Abstract

AI agents increasingly operate through tools, APIs, databases, files, messaging systems, workflow engines, and enterprise applications. This creates a pre-runtime governance problem: before an agent is deployed, an enterprise needs to know what actions the agent may propose, which tools those actions use, which actions require authority, which actions require review, what reliance evidence should be captured, and which payload fields may require redaction. Cognous Agent Action Manifest is a minimal public reference format for declaring the action surface of an AI agent before runtime execution. It provides a JSON-backed schema, Pydantic object model, validation rules, command-line utilities, examples, and documentation for describing agent tools, action declarations, authority requirements, review requirements, reliance requirements, payload policies, and redaction hints. It is intentionally scoped: it is not an agent framework, not a model runtime, not a policy engine, not a runtime control plane, not a security boundary by itself, and not a complete enterprise governance platform. Its contribution is narrower and operational: a small, inspectable declaration layer that helps teams inventory agent actions before those actions are governed, recorded, replayed, or reviewed in downstream runtime systems.

1 Introduction

AI-agent systems are moving from conversational assistance toward delegated execution. Agents now retrieve records, search files, draft messages, invoke APIs, update systems, generate recommendations, and trigger workflows across enterprise boundaries. This shift creates a governance requirement that appears before runtime: the enterprise needs a structured declaration of what the agent may propose to do.

A deployed agent can only be governed well if its action surface is understood. Before runtime execution, teams need to answer several basic questions:

1. What tools can this agent use?
2. What actions can this agent propose?
3. Which actions are reads, writes, exports, sends, purchases, approvals, deletes, or escalations?
4. Which actions require explicit authority?
5. Which actions require human review or approval?
6. Which actions require reliance records?

7. Which payload fields are required, optional, sensitive, or forbidden?

8. Which fields should be redacted in downstream evidence exports?

9. What should happen by default when an action posture is not otherwise specified?

In many agent deployments, this information is scattered across prompts, code, tool wrappers, configuration files, permission systems, documentation, and informal team knowledge. The result is configuration drift: runtime control systems may be expected to govern actions that were never clearly declared.

Agent Action Manifest occupies a narrow layer in the governed-agent lifecycle. It is a declaration format for action inventory and governance preparation. It does not execute the agent. It does not enforce runtime control. It does not replace application authorization. It creates a structured, portable description of what an agent may propose.

The guiding design principle is:

> No enterprise agent should move toward governed deployment without a structured declaration of its tools, action types, authority requirements, review requirements, reliance requirements, payload policies, and redaction hints.

The public reference implementation exposes a small set of public objects and utilities: `AgentActionManifest`, `ToolDefinition`, `ToolAction`, `AuthorityRequirement`, `ReviewRequirement`, `RelianceRequirement`, `PayloadPolicy`, `RedactionHint`, `ValidationIssue`, `ValidationReport`, a command-line interface, JSON schemas, example manifests, validation helpers, and a test suite.

2 Problem Statement

Enterprise AI-agent adoption creates a pre-runtime action-declaration gap. The organization may know that an agent exists, but it may not have a structured record of what the agent can propose, which systems it may touch, and which controls should apply to each action.

This gap has several operational forms.

2.1 Unclear Action Surface

Agent capabilities are often described in natural language or implied by the tools available to the agent. This is insufficient for governance. A tool may support several different actions, and those actions may have different risk profiles. Reading a record, drafting a message, sending a message, exporting data, deleting data, and approving a transaction should not be treated as equivalent.

2.2 Tool Access Without Action Semantics

A tool list tells the enterprise which systems the agent may access, but not what the agent may do with those systems. Tool access alone does not distinguish harmless reads from privileged writes, outbound communications, approvals, purchases, exports, or deletions.

2.3 Ambiguous Authority Requirements

Some actions should require explicit authority. Others may be allowed by default, blocked by default, or escalated for review. Without a declaration of authority requirements, runtime systems may apply inconsistent or overly broad permission logic.

2.4 Inconsistent Human Review

Human review requirements are often handled informally. One team may require approval before outbound communication; another may rely on the agent's internal judgment. A manifest makes review posture explicit at the action level.

2.5 Missing Reliance Requirements

Some actions should produce reliance evidence: a record of which tool, database, file, user input, API, or model output the agent depended on. Without a reliance declaration, runtime provenance may be incomplete or inconsistent.

2.6 Unsafe Payload Handling

Agent actions often carry payloads. Payloads may contain customer identifiers, account notes, addresses, emails, financial data, access tokens, or other sensitive fields. Without payload policies and redaction hints, downstream exports may expose more than necessary.

2.7 Configuration Drift

The absence of a manifest encourages drift between documentation, runtime policy, code, and operational practice. A public manifest format gives teams a stable declaration layer that can be reviewed before runtime deployment.

3 Design Goals

Agent Action Manifest is designed around eight goals.

1. **Action inventory.** Agent actions should be declared explicitly before runtime execution.
2. **Tool-action separation.** Tools and actions should be related but distinct; one tool may support multiple actions.
3. **Authority declaration.** Privileged actions should declare required authority scopes.
4. **Review declaration.** Actions should declare whether human review, approval, or draft-first handling is required.
5. **Reliance declaration.** Actions should declare whether reliance evidence is expected.
6. **Payload policy.** Required, optional, sensitive, and forbidden payload fields should be represented explicitly.
7. **Redaction preparation.** Sensitive fields should be accompanied by redaction hints where appropriate.
8. **Portable validation.** Manifests should be loadable, inspectable, and validated through a small public schema and CLI.

4 Non-Goals

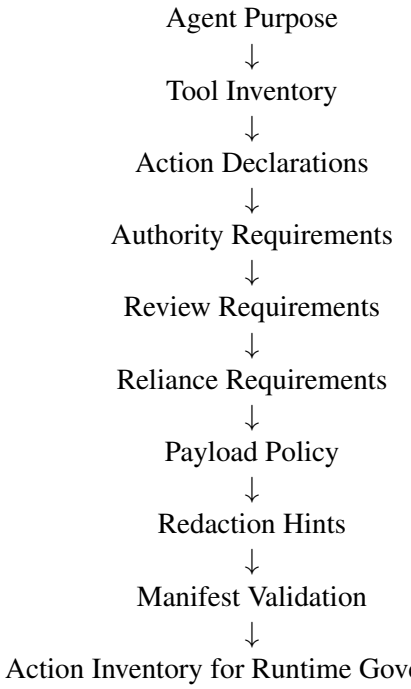
The architecture is deliberately bounded. Agent Action Manifest is not:

- an agent framework;
- a model runtime;
- a planner;
- a policy engine;
- a runtime control plane;
- a dashboard;
- a complete enterprise governance platform;
- a production authorization system;
- a security boundary by itself;
- a substitute for application-level authorization;
- a guarantee that model outputs are true, safe, or complete.

It declares action posture and validates manifest consistency. It does not execute actions, enforce runtime access control, or prove that a deployed agent will behave correctly.

5 System Overview

The declaration pattern is:



The manifest is created before or alongside agent deployment. It describes the agent's action surface in a structured format. A runtime system may then use the manifest as input, reference, or documentation when recording, gating, validating, replaying, or reviewing actual agent runs.

In the Cognous Open Control Stack, Agent Action Manifest is the *Declare* layer:

Agent Action Manifest
↓
Agent Control Plane
↓
Agent Replay Bundle
↓
Agent Governance Evidence Pack

The manifest declares what an agent may propose. A control plane records what the agent actually proposes at runtime. Replay bundles package runtime records. Evidence packs translate runtime evidence into business-facing governance review.

6 Core Object Model

6.1 AgentActionManifest

An `AgentActionManifest` is the top-level object. It describes the agent, deployment context, default action posture, declared tools, declared actions, and optional metadata.

Field	Meaning
`manifest_version`	schema or manifest version
`manifest_id`	unique manifest identifier
`agent_name`	name of the agent being described
`agent_description`	optional human-readable description
`owner`	responsible team or owner
`environment`	deployment context
`default_action`	manifest-level default posture
`tools`	declared tools
`actions`	declared actions
`metadata`	optional metadata

6.2 ToolDefinition

A `ToolDefinition` describes a tool the agent may use. It records the tool name, description, whether it is allowed, external system linkage, and optional data classification.

6.3 ToolAction

A `ToolAction` describes an action the agent may propose through a tool. It records the action name, tool name, action type, description, default action override, authority requirements, review requirement, reliance requirement, payload policy, redaction hints, tags, and metadata.

6.4 AuthorityRequirement

An `AuthorityRequirement` declares an authority scope required for an action. It may include a description, whether the scope is required, and the source or governing system for that authority.

6.5 ReviewRequirement

A `ReviewRequirement` declares the review posture for an action. The current public model supports:

- `none`;
- `human_review`;
- `approval_required`;
- `draft_first`.

6.6 RelianceRequirement

A `RelianceRequirement` declares whether reliance evidence should be recorded and which source types are acceptable. Source types include tools, databases, files, APIs, user input, model output, and other sources.

6.7 PayloadPolicy

A `PayloadPolicy` declares field-level expectations for an action payload. It can specify required fields, optional fields, sensitive fields, and forbidden fields.

6.8 RedactionHint

A `RedactionHint` identifies a field path that may need to be redacted in downstream exports. It includes an optional reason and replacement value.

6.9 ValidationIssue and ValidationReport

A `ValidationIssue` records an error or warning discovered during manifest validation. It includes severity, numeric code, optional semantic alias, message, and optional path. A `ValidationReport` aggregates validation issues and marks the manifest as valid only when no error-severity issues are present.

7 Action Types

The manifest uses action types to classify agent behavior at the level needed for governance review. The current reference implementation supports the following action types:

Action Type	Meaning
`read`	retrieves or inspects information
`write`	creates or modifies information without necessarily sending externally
`external_send`	sends information outside the immediate system boundary
`delete`	removes information or records
`export`	packages or transfers information for downstream use
`purchase`	initiates or prepares a purchase action
`approve`	approves, confirms, or authorizes a workflow step
`escalate`	escalates to a human, queue, or supervisory process
`other`	action type not otherwise classified

These types are intentionally simple. They are not a full ontology of enterprise action semantics. Their purpose is to create a practical governance vocabulary that can be validated and inspected before runtime deployment.

8 Default Action Posture

A manifest and individual actions may declare a default posture. The supported values are:

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

The default posture is not runtime enforcement by itself. It is a declaration of intended treatment. A runtime control layer may use it as input or reference, but the manifest does not execute policy decisions.

The default posture helps teams identify whether an action is intended to be allowed, blocked, or sent for review when no more specific runtime rule applies.

9 Authority Requirements

Authority requirements express what permission scope should exist before an action is considered eligible for execution. For example, an outbound email action may require a scope such as `email.send.customer`. A purchase-order action may require a scope such as `purchase.order.create`. A payment approval action may require a scope such as `payment.approve`.

Authority requirements do not grant authority. They declare expected authority. This distinction is central: the manifest says what should be required; a runtime system or application authorization layer decides whether authority actually exists.

Privileged action types such as `external_send`, `write`, `delete`, `purchase`, and `approve` should generally declare authority unless they are blocked by default.

10 Review Requirements

Review requirements describe whether an action should be subject to human review, approval, or draft-first handling.

Review Mode	Meaning
`none`	no specific review posture declared
`human_review`	human review expected before execution or completion
`approval_required`	approval required from a specified reviewer role
`draft_first`	action should produce a draft before final execution

A review requirement can specify a reviewer role and a reason. For approval-required actions, the public validator expects a reviewer role to be present.

11 Reliance Requirements

Reliance requirements describe whether runtime systems should capture the sources an action depends on. This is especially important for read, export, and external-send actions, where provenance and downstream review matter.

The supported reliance source types are:

- `tool`;
- `database`;
- `file`;
- `api`;
- `user_input`;
- `model_output`;
- `other`.

Reliance requirements do not create runtime evidence by themselves. They declare that a downstream control or recording layer should capture reliance evidence for the action.

12 Payload Policy and Redaction Hints

Agent actions often carry payloads. A payload may include fields such as a customer identifier, email address, account note, invoice number, transaction amount, message body, subject line, export target, or approval comment.

A `PayloadPolicy` can declare:

- required fields;
- optional fields;
- sensitive fields;
- forbidden fields.

A `RedactionHint` can declare that a field should be redacted in exported records. This supports downstream evidence workflows by making sensitive fields visible to governance tooling before runtime.

Payload policies and redaction hints are not access-control mechanisms. They are declaration artifacts that help teams prepare safe runtime recording, replay, validation, and evidence-export workflows.

13 Validation

The current public reference implementation validates manifest consistency and emits a `ValidationReport`. Errors invalidate the manifest. Warnings indicate review concerns but do not make the manifest invalid.

Some validation issues may also include semantic aliases, such as `privileged_action_missing_authority` or `disallowed_tool_reference`, to support stable downstream interpretation without replacing numeric rule codes.

13.1 Error Conditions

The validator checks the following error conditions:

1. `manifest_version` must be non-empty.
2. `manifest_id` must be non-empty.
3. `agent_name` must be non-empty.
4. action names must be unique.
5. tool names must be unique.
6. every action must reference a declared tool.
7. `external_send` actions must declare authority unless blocked by default.
8. `write`, `delete`, `purchase`, and `approve` actions must declare authority unless blocked by default.
9. forbidden payload fields must not overlap with required payload fields.
10. forbidden payload fields must not overlap with optional payload fields.
11. sensitive fields should refer to required or optional fields when those lists are non-empty.
12. redaction hint field paths must be non-empty.
13. approval-required actions must specify a reviewer role.
14. high-risk actions with default posture `allow` must have authority and non-none review.
15. actions must not reference tools marked `allowed=false` unless the action is blocked by default.

13.2 Warning Conditions

The validator also checks warning conditions, including:

- manifest has no actions;
- manifest has no tools;
- manifest default posture is `allow`;
- action effective default posture is `allow` for `write` or `external_send`;
- reliance requirement is missing for `read`, `export`, or `external_send`;

- payload policy is missing for higher-risk action types;
- sensitive fields are declared but no redaction hints are present;
- external-system tools lack data classification;
- action has no description;
- manifest owner is missing.

Validation remains implementation-level. It does not prove that a manifest is complete, that the declared policies are sufficient, that runtime enforcement exists, or that the deployed agent is safe.

14 Command-Line Interface

The `aam` command-line interface provides small utilities for manifest review.

14.1 Implemented Commands

The current public reference implementation includes:

- `aam validate path/to/manifest.json`;
- `aam summarize path/to/manifest.json`;
- `aam list-actions path/to/manifest.json`;
- `aam check-examples`.

The validation command loads a manifest, runs manifest validation, prints errors and warnings, and returns a process exit code. The summarize command prints counts by action type, default posture, authority requirement, review requirement, reliance requirement, and sensitive-field presence. The list-actions command prints a compact action inventory. The check-examples command validates example manifests.

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

15 JSON Schemas and Public Conformance

The repository includes JSON Schema Draft 2020-12 files for the public manifest objects. These schemas define the public interchange boundary for records such as `AgentActionManifest`, `ToolAction`, `AuthorityRequirement`, `ReviewRequirement`, `RelianceRequirement`, `PayloadPolicy`, `RedactionHint`, and `ValidationReport`. The validation report schema includes optional semantic aliases for validation issues.

The schemas serve four purposes:

1. **Interchange.** They allow external systems to understand the shape of manifest records.
2. **Validation.** They support structural validation outside the Python package.
3. **Documentation.** They make the public manifest 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 a manifest is operationally sufficient, that runtime controls are correctly implemented, or that an enterprise compliance obligation has been satisfied.

16 Example Manifests

The public repository includes example manifests for common enterprise agent scenarios.

16.1 Customer-Service Agent

A customer-service agent reads CRM records, searches notes, drafts emails, and sends approved customer communications. The manifest distinguishes reads from outbound sends, declares authority for sending email, requires approval for outbound communication, and marks customer identifiers and email content as redaction-relevant. The email-draft action can also declare reliance expectations for CRM records, account notes, or user input used to prepare the draft.

16.2 Internal Research Agent

An internal research assistant searches documents, reads files, generates summaries, and exports reports. The manifest declares reliance requirements for retrieval and read actions and payload policies for export behavior.

16.3 Procurement Agent

A procurement agent looks up vendors, compares quotes, drafts purchase recommendations, creates purchase orders, and sends vendor emails. The manifest declares authority and review requirements for purchase and outbound communication actions.

16.4 Finance Workflow Agent

A finance workflow agent reads invoice records, generates payment recommendations, proposes payment approval, and exports audit records. The manifest declares authority and review requirements for approval actions and redaction posture for exportable audit evidence.

17 Relationship to Runtime Control

Agent Action Manifest is not a runtime control layer. Its primary role is to declare what an agent may propose before runtime.

A runtime control layer can use the manifest in several ways:

- to derive allowed or blocked tools;
- to identify high-risk actions;
- to determine which actions require authority;
- to determine which actions require human review;

- to identify fields that should be redacted in exports;
- to compare runtime action proposals against declared action inventory;
- to support governance evidence generation.

In the Cognous public reference stack, Agent Action Manifest complements Agent Control Plane. The manifest declares the expected action surface. The control plane records and governs actual runtime proposals.

18 Security Considerations

Agent Action Manifest improves action visibility and governance preparation, but it should not be treated as a security product.

Important limitations include:

- the manifest declares requirements but does not enforce them;
- declared authority requirements do not grant or verify authority;
- review requirements do not guarantee that review occurred;
- reliance requirements do not create runtime reliance records by themselves;
- payload policies do not prevent payload misuse by themselves;
- redaction hints do not guarantee safe export unless applied by downstream tooling;
- schema validation checks structure and consistency, not operational sufficiency;
- production deployments still require authentication, authorization, logging, monitoring, runtime control, and incident response.

The correct security posture is layered. Agent Action Manifest supplies structured pre-runtime declaration. It should be combined with application authorization, runtime policy gates, run recording, replay bundles, monitoring, operational controls, and governance review.

19 Implementation Boundary

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

The public boundary is therefore:

Public Reference Layer	Private Product Layer
Action manifest schema	enterprise action registry
Tool definitions	customer-specific tool discovery
Action declarations	proprietary action classification
Authority requirements	customer-specific authority models
Review requirements	enterprise approval workflows
Reliance requirements	runtime provenance systems
Payload policies	enterprise data-classification systems
Redaction hints	production redaction workflows
Validation reports	compliance evidence automation
CLI utilities	hosted management interfaces
Example manifests	customer-specific deployment templates
JSON schemas	enterprise conformance tooling

Public software should expose the integration contract: what developers can declare, what reviewers can inspect, and what downstream systems can validate. Private product systems can add enterprise integrations, hosted workflows, richer policy authoring, automated discovery, approval routing, and deployment-specific governance logic.

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
Manifest loader	Implemented	JSON file loading and model validation
Manifest dumper	Implemented	formatted JSON export
Validation reports	Implemented	error and warning issues
Validation issue aliases	Implemented	optional semantic aliases for validation rules
Action type declarations	Implemented	read, write, send, delete, export, purchase, approve, escalate, other
Authority requirements	Implemented	scoped requirement declarations
Review requirements	Implemented	none, human review, approval required, draft first
Reliance requirements	Implemented	expected evidence source types
Payload policies	Implemented	required, optional, sensitive, forbidden fields
Redaction hints	Implemented	field-level export preparation
CLI validation	Implemented	`aam validate`
CLI summary	Implemented	`aam summarize`
CLI action listing	Implemented	`aam list-actions`
Example validation	Implemented	`aam check-examples`
JSON schemas	Implemented	public schema boundary
Example manifests	Implemented	customer service, research, procurement, finance
Test suite	Implemented	model, loader, validator, CLI, examples
Hosted registry	Extension	out of current minimal scope
Runtime enforcement	Extension	handled by downstream systems
Enterprise approval workflows	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:

- valid minimal manifest loading;
- valid full manifest loading;
- enum validation;
- loader behavior;

- invalid JSON behavior;
- model validation errors;
- duplicate action names;
- duplicate tool names;
- unknown tool references;
- references to disallowed tools;
- missing authority for privileged actions;
- blocked privileged actions without authority;
- payload-policy overlap errors;
- approval-required actions without reviewer role;
- validation issue aliases;
- warning/error distinction;
- CLI validation;
- CLI summary;
- CLI action listing;
- example manifest validation.

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

22 Example Use Cases

22.1 Agent Deployment Review

An AI platform team prepares to deploy a customer-service agent. Before runtime rollout, the team creates a manifest describing CRM reads, note searches, email drafts, and outbound email sends. The manifest shows that outbound sends require authority and approval, while CRM reads require reliance evidence and redaction of customer identifiers. The email draft can also declare reliance expectations for the source material used to prepare the draft.

22.2 Security Review

A CISO organization reviews an agent proposed for production deployment. The manifest provides a structured inventory of tools, privileged actions, authority requirements, review posture, sensitive fields, and redaction hints. The security team can identify actions requiring additional runtime controls.

22.3 Governance Evidence Preparation

A governance team needs business-readable evidence for an agent deployment. The manifest supplies the declared action inventory that can later be compared against runtime run records, replay bundles, and evidence packs.

22.4 Runtime Control Integration

A developer integrates an agent with a runtime control layer. The manifest helps identify expected action names, action types, authority scopes, reliance expectations, and redaction-sensitive fields before runtime action proposals are recorded and evaluated.

23 Evaluation Criteria

The usefulness of Agent Action Manifest should be evaluated against operational criteria rather than benchmark scores.

A deployment should ask:

1. Does the manifest identify the agent being described?
2. Does it declare the tools the agent may use?
3. Does it distinguish tools from actions?
4. Does it classify each action by action type?
5. Does it declare authority requirements for privileged actions?
6. Does it declare review posture for actions that need human oversight?
7. Does it declare reliance expectations where provenance matters?
8. Does it identify required, optional, sensitive, and forbidden payload fields?
9. Does it provide redaction hints for sensitive fields?
10. Does it validate with no error-severity issues?
11. Does it produce warnings that are useful for review?
12. Can validation issues be interpreted consistently through stable codes or aliases?
13. Can runtime systems use it as an action inventory?
14. Can governance teams inspect it before deployment?
15. Can it support later runtime replay and evidence workflows?

24 Roadmap

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

Implemented MVP capabilities include:

- action manifest object model;
- tool definitions;
- action declarations;
- authority requirements;
- review requirements;
- reliance requirements;
- payload policies;
- redaction hints;
- manifest validation reports;
- validation issue aliases;
- command-line validation;
- command-line summaries;
- action listing;
- example manifests;
- JSON schemas;
- test suite and continuous integration.

Near-term extensions may include:

- richer action categories;
- optional manifest-to-policy-configuration converter;
- manifest diffing;
- manifest risk summary;
- OpenAPI or tool-schema mapping examples;
- conformance test suite;
- additional industry examples.

Longer-term work may include UI editors, registry patterns, integration examples, enterprise governance workflows, and hosted review tooling. Those are product-layer concerns and should remain separate from the minimal public reference implementation.

25 Conclusion

AI-agent governance cannot begin only at runtime. Before an agent proposes actions, uses tools, sends messages, exports records, approves workflows, or touches enterprise systems, teams need a structured declaration of what the agent may propose and what governance posture applies to each action.

Cognous Agent Action Manifest provides a minimal public reference architecture for that declaration layer. It describes agent tools, action declarations, authority requirements, review requirements, reliance requirements, payload policies, and redaction hints. It validates manifest consistency, publishes JSON schemas for the public manifest boundary, includes example manifests for common enterprise scenarios, and provides command-line utilities for local review.

The contribution is deliberately narrow. Agent Action Manifest does not run agents, enforce policies, train models, produce runtime records, or guarantee correctness. It supplies a concrete pre-runtime declaration artifact for a specific operational need: making an AI agent's action surface inspectable before the agent moves into governed deployment.

References

[1] Cognous. *Agent Action Manifest: Declare what actions an AI agent may propose before runtime execution*. Public reference implementation, 2026. `https://github.com/cogno-us/cognous-agent-action-manifest`

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

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

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

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