

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

André de Lima
Cognous
Seattle, Washington
`andrede Lima@cogno.us`

October 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 require review, what reliance evidence should be captured, and which payload fields may require redaction. It also creates a handoff problem: a runtime controller that knows only an action's name cannot detect a substituted adapter, target, payload, or amount. Cognous Action Manifest is a minimal public reference format for declaring the action surface of an AI agent before runtime execution. It provides JSON schemas, a 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. Manifest 1.1 adds a backward-compatible runtime-consumption profile: stable action and adapter identities, finite target policies, explicit effect limits, reference-only Authority Context requirements, canonical SHA-256 commitments, and deterministic declaration matching. The component belongs to the Declaration & Policy Control group of the thirteen-component Cognous Open Control Stack, whose selected configuration is a bounded synthetic reference. It is intentionally scoped: it is not an agent framework, not a policy engine, not a runtime control plane, not an authorization system, and not a security boundary by itself. A valid manifest or a matching proposal is never an authorization token. Its contribution is narrower and operational: a small, inspectable declaration layer that runtime control can bind to exactly.

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, issue refunds, 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, and which institutional requirement governs that 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. Which targets may a consequential action reach, and within what amount and effect limits?
10. 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.

Action Manifest occupies a narrow layer in the governed-agent lifecycle. It is a declaration format for action inventory, governance preparation, and exact runtime binding. It does not execute the agent. It does not enforce runtime control. It does not resolve or issue authority. It creates a structured, portable description of what an agent may propose, precise enough that a downstream controller can tell whether a concrete proposal fits it.

The guiding design principle, unchanged since the first public version, 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 repository’s own summary of its role is shorter: *declare the action before evaluating permission*.

The public reference implementation exposes a small set of public objects and utilities:

Declaration	AgentActionManifest, ToolDefinition, ToolAction, AuthorityRequirement, ReviewRequirement, RelianceRequirement, PayloadPolicy, RedactionHint
Validation	ValidationIssue, ValidationReport, validate_manifest
1.1 binding	TargetPolicy, EffectLimits, AuthorityContextRequirement, RuntimeActionProposal
Commitments and matching	manifest_digest, payload_digest, proposal_digest, resolve_action, proposal_mismatches

together with a command-line interface, JSON schemas, example manifests, and a test suite.

Python class and package names retain the earlier “Agent Action Manifest” naming; the distribution is still called `agent-action-manifest`, and the module `agent_action_manifest`. The repository rename did not change package names, schema identifiers, or the `aam` command.

This paper updates the July 2026 version. Its principal changes concern the Manifest 1.1 runtime-consumption profile (Sections 13 and 14), the component’s position in the current stack (Section 5), its consumption by the Control Plane (Section 19), and a revised account of evidence and status (Sections 22 and 23).

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. A tool name is not a sufficient policy boundary.

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 Loose Runtime Handoff

When a declaration is consumed at runtime, name-level matching is weak. A proposal that reuses a declared action name can still substitute the executing adapter, the target account, payload fields, the amount, or the requested permissions. Unless the declaration carries stable identities and finite limits, and unless the proposal is committed to exactly, such substitutions are not detectable by the consumer.

2.8 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, and a stable digest that runtime records can refer back to.

3 Design Goals

Action Manifest is designed around nine 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 and, for runtime consumption, the institutional requirement to be resolved.
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. **Exact runtime binding.** Consequential actions should carry stable identities, finite targets, and explicit limits, and a proposal should be comparable to its declaration deterministically.
9. **Portable validation.** Manifests should be loadable, inspectable, and validated through a small public schema and CLI.

4 Non-Goals

The architecture is deliberately bounded. Action Manifest is not:

- an agent framework;
- a model runtime;
- a planner;
- a policy engine;
- a runtime control plane;
- an authority resolver, or an issuer of grants or approvals;
- 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, validates manifest consistency, and reports structural mismatches between a proposal and a declaration. It does not execute actions, enforce runtime access control, establish institutional legitimacy, or prove that a deployed agent will behave correctly.

5 System Overview

The declaration pattern is:

Agent Purpose → Tool Inventory → Action Declarations → Authority Requirements
→ Review Requirements → Reliance Requirements → Payload Policy → Redaction Hints
→ Runtime Binding (1.1) → Manifest Validation → Action Inventory for Runtime Governance

The manifest is created before or alongside agent deployment. A runtime system may then use it as input, reference, or documentation when recording, gating, validating, reconstructing, or reviewing actual agent runs. Under the 1.1 profile, a runtime proposal binds to the manifest by identifier, version, and digest.

The July 2026 version of this paper placed the manifest as the first of four sequential layers (Declare, Control, Replay, Evidence). The Cognous Open Control Stack is now organized differently: thirteen separately owned and versioned components in six responsibility groups, with Governed Exchange (GAX/IMX) as cross-cutting integration (Table 1). The groups are a responsibility map, not an execution sequence or an installation requirement.

Group	Components
01 Knowledge & Provenance	Evidence Registry (The Index); Evidence Attestation (BitRep)
02 Institutional Governance	Alvorada (Institutional Governance)
03 Declaration & Policy Control	Action Manifest ; Control Plane
04 Constrained Execution	Execution Runtime (Moltbot Safe)
05 Replay & Evidence	Replay Bundle; Governance Evidence Pack; ODES
06 Reasoning & Human Oversight	PRP; Research Intelligence; TFA (optional instruction layers)
Cross-cutting	Governed Exchange (GAX/IMX)

Table 1: Responsibility groups of the Cognous Open Control Stack.

Within group 03, the manifest declares the proposed action envelope; the Control Plane evaluates a Manifest-bound proposal against Authority Context supplied by an independently configured resolver and revalidates decision-critical inputs before effect. Alvorada (group 02) supplies the Authority Context schema that 1.1 declarations reference. The Governance Evidence Pack (group 05) imports the manifest together with reconstructed runtime records through a versioned transformation, so that the declared action surface and the observed run can be compared. The stack supports one workflow end to end: a synthetic refund delivered through local durable transport, evaluated by the Control Plane, and executed against a synthetic SQLite destination on one host.

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
<code>manifest_version</code>	manifest version (1.0 or 1.1)
<code>manifest_id</code>	unique manifest identifier
<code>agent_name</code>	name of the agent being described
<code>agent_description</code>	optional human-readable description
<code>owner</code>	responsible team or owner
<code>environment</code>	deployment context (default development)
<code>default_action</code>	manifest-level default posture (default escalate)
<code>tools</code>	declared tools
<code>actions</code>	declared actions
<code>metadata</code>	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, optional data classification, and, for 1.1 consumption, a stable adapter identity (`adapter_id`) naming the runtime adapter that executes its actions.

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. For 1.1 consumption it adds a stable `action_id`, a `target_policy`, `effect_limits`, and an `authority_context` reference (Section 13).

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 public model supports `none`, `human_review`, `approval_required`, and `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 a replacement value (default `[REDACTED]`).

6.9 Runtime Binding Objects

`TargetPolicy` lists a finite set of allowed runtime targets; its legacy `allow_any_target` flag must be false for 1.1 runtime consumption. `EffectLimits` declares a maximum amount, its unit, and a maximum effect count per proposal. `AuthorityContextRequirement` carries reference-only coordinates into the Alvorada Authority Context schema. `RuntimeActionProposal` is the exact proposal material a runtime submits for binding (Section 14).

6.10 ValidationIssue and ValidationReport

A `ValidationIssue` records an error or warning discovered during manifest validation. It includes severity, a stable code, an optional semantic alias, a message, and an 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 reference implementation supports the following action types:

Action type	Meaning
<code>read</code>	retrieves or inspects information
<code>write</code>	creates or modifies information without necessarily sending externally
<code>external_send</code>	sends information outside the immediate system boundary
<code>delete</code>	removes information or records
<code>export</code>	packages or transfers information for downstream use
<code>purchase</code>	initiates or prepares a purchase action
<code>approve</code>	approves, confirms, or authorizes a workflow step
<code>escalate</code>	escalates to a human, queue, or supervisory process
<code>other</code>	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. In the 1.1 profile, `write`, `external_send`, `delete`, `export`, `purchase`, and `approve` are treated as effectful for the purposes of target and Authority Context rules.

8 Default Action Posture

A manifest and individual actions may declare a default posture: `allow`, `block`, or `escalate`. The manifest-level default is `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. In particular, `allow` is never a substitute for the independently resolved authority that the Control Plane requires before an effect.

9 Authority Requirements

Authority requirements express what permission scope should exist before an action is considered eligible for execution. For example, a purchase-order action may require the scope `purchase.order.create`, and an outbound email action may require `email.send.customer`; the synthetic refund fixture requires `refund.issue.routine`.

Authority requirements do not grant authority. They declare expected authority. This distinction is central: the manifest says what should be required; a runtime controller, using authoritative institutional sources, decides whether authority actually exists. Under the 1.1 profile, each statically required scope must appear in a proposal's `requested_permissions`; this too is declaration matching, not proof that the permission was granted.

Privileged action types such as `external_send`, `write`, `delete`, `purchase`, and `approve` should 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
<code>none</code>	no specific review posture declared
<code>human_review</code>	human review expected before execution or completion
<code>approval_required</code>	approval required from a specified reviewer role
<code>draft_first</code>	action should produce a draft before final execution

A review requirement can specify a reviewer role and a reason. For approval-required actions, the validator expects a reviewer role to be present. In the integrated reference, the Control Plane does not accept a manifest reviewer label at face value: manifest review roles must resolve explicitly to Authority Context approval role identifiers through an institution-scoped, versioned mapping, and unmapped or ambiguous roles hold the effect.

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`, and `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 refund reason. A `PayloadPolicy` can declare required, optional, sensitive, and forbidden fields. A `RedactionHint` can declare that a field should be redacted in exported records.

Payload policies and redaction hints are not access-control mechanisms. They are declaration artifacts that help teams prepare safe runtime recording, reconstruction, validation, and evidence-export workflows; redaction must be applied by the exporting system. Under the 1.1 profile, however, the payload policy also defines the envelope used in declaration matching: a proposal missing a required field, carrying a forbidden field, or carrying an undeclared field does not match.

13 The Manifest 1.1 Runtime-Consumption Profile

Manifest 1.1 is backward-compatible at the package level: 1.0 manifests still load and validate. A consumer that requires deterministic runtime binding must require version 1.1 explicitly and reject unsupported versions rather than coercing them. A 1.1 action declaration adds:

action_id a stable action identifier, unique within the manifest;

tool_adapter_id a stable identity for the runtime adapter that will execute the action;

target_policy a finite list of allowed targets, required for effectful actions that are not blocked by default;

effect_limits maximum amount, unit, and effect count for one proposal;

authority_context a reference-only requirement in Alvorada Authority Context 0.1.0, required for effectful actions that are not blocked by default.

The Authority Context reference names a profile, requirement identifier, institution identifier, authority domain, consequence tier (T0–T4), and evidence obligation identifiers. The mapping to Authority Context concepts, and the obligation each places on the consumer, is summarized below.

Manifest field	Consumer obligation
<code>profile_ref</code>	require the exact supported implementation profile
<code>requirement_id</code>	resolve the current requirement
<code>institution_id, authority_domain</code>	bind the institutional domain and authority scope
<code>consequence_tier</code>	select the corresponding process burden
<code>evidence_obligation_ids</code>	resolve the required evidence
<code>target_policy.allowed_targets</code>	keep the proposal within both declaration and resolved grant
<code>effect_limits</code>	a downstream grant may only narrow these limits

The manifest deliberately does not copy grants, approval records, status records, or mandate records. Those are dynamic authority facts owned by institutional sources and resolved downstream. The pinned Authority Context baseline is a synthetic integration baseline, not evidence of institutional adoption.

14 Canonical Commitments and Declaration Matching

14.1 Commitments

The reference implementation computes three commitments, each SHA-256 over UTF-8 JSON with sorted keys, compact separators, `ensure_ascii=false`, explicit nulls, and non-finite numbers disallowed, stored as `sha256:<hex>`:

- `manifest_digest`, over the complete manifest representation;
- `payload_digest`, over a proposal payload;
- `proposal_digest`, over the complete runtime proposal.

14.2 Runtime Action Proposal

A `RuntimeActionProposal` carries the actor and principal, the manifest identifier, version, and digest, the action and adapter identifiers, the target, the payload and its commitment, the requested permissions, amount, unit, and effect count, Authority Context profile and requirement references, risk metadata, temporal bounds, correlation and run identifiers, expected side effects, and evidence references. Downstream authorization should bind to the complete proposal commitment, not to an action name. Changing any authorization-bound field produces a materially different proposal that needs a new downstream decision.

14.3 Matching

`proposal_mismatches` performs deterministic structural matching and returns named mismatches, including unsupported or changed manifest version, changed manifest identifier or digest, stale payload commitment, unknown action, unknown tool or substituted adapter, missing or unbounded target policy, target out of scope, missing required, forbidden, or undeclared payload fields, effect count or amount above the declared limit, unit mismatch, missing requested-permission scope, and Authority Context profile or requirement mismatch.

An empty mismatch list establishes only that a proposal fits the declaration. A valid manifest, a valid digest, or a matching proposal is never an authorization token.

Expired or revoked grants, approval failures, delegation-chain failures, and policy or status mismatches are outside this module; they belong to the Control Plane's resolver.

14.4 Tenant-Aware Proposal 1.2 (Accepted, Not Selected)

A later contract generation, `runtime-action-proposal/1.2`, adds a required `tenant_id`: an opaque string of 1–128 characters supplied by trusted deployment or authority context, never inferred from caller metadata, targets, payloads, or adapter responses, and included in the proposal commitment so that a tenant-only substitution changes the commitment. Historical 1.1 proposals are not rewritten or relabeled. This generation is merged in the component repository and appears in a separately recorded bounded-V1 reference candidate, but it is not part of the hub's selected release, and that candidate is not authorized for release.

15 Validation

The reference implementation validates manifest consistency and emits a `ValidationReport`. Errors invalidate the manifest. Warnings indicate review concerns but do not make the manifest invalid. Many issues also carry semantic aliases, such as `privileged_action_missing_authority` or `disallowed_tool_reference`, to support stable downstream interpretation without replacing the codes.

15.1 Error Conditions

The validator checks the following error conditions:

1. `manifest_version` must be non-empty (E001) and supported, currently 1.0 or 1.1 (E016);
2. `manifest_id` must be non-empty (E002);
3. `agent_name` must be non-empty (E003);
4. action names must be unique (E004);
5. tool names must be unique (E005);
6. every action must reference a declared tool (E006);
7. `external_send` actions must declare authority unless blocked by default (E007);
8. `write`, `delete`, `purchase`, and `approve` actions must declare authority unless blocked by default (E008);
9. forbidden payload fields must not overlap with required fields (E009) or optional fields (E010);
10. sensitive fields must refer to required or optional fields when those lists are non-empty (E011);
11. redaction hint field paths must be non-empty (E012);
12. approval-required actions must specify a reviewer role (E013);
13. high-risk actions with default posture `allow` must have authority and non-`none` review (E014);
14. actions must not reference tools marked `allowed=false` unless blocked by default (E015).

For 1.1 manifests the validator additionally requires unique `action_id` values (E017), an `action_id` on every action (E018), an `adapter_id` on every referenced tool (E019), and, for effectful actions not blocked by default, a finite non-empty target policy (E020) and an Authority Context reference (E021).

15.2 Warning Conditions

The validator also checks warning conditions: the manifest has no actions or no tools; the manifest default posture is `allow`; an action's effective posture is `allow` for a `write` or `external send`; a reliance requirement is missing for `read`, `export`, or `external send`; a payload policy is missing for higher-risk action types; sensitive fields are declared without redaction hints; an external-system tool lacks data classification; an action has no description; or the 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.

16 Command-Line Interface

The `aam` command-line interface provides small utilities for manifest review. The implemented commands are:

- `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 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 the bundled example manifests. These commands are intended for local validation and lightweight workflows; they are not a hosted service, daemon, dashboard, or enterprise control surface. Digest computation and declaration matching are available through the Python API rather than the CLI.

From a repository checkout, with Python 3.11 or later in an activated virtual environment:

```
python -m pip install -e ".[dev]"
aam validate examples/refund_integration_v1_1.manifest.json
aam summarize examples/customer_service_agent.manifest.json
aam check-examples
```

17 JSON Schemas and Public Conformance

The repository includes JSON Schema files for the public objects: the manifest, tool action, authority, review, and reliance requirements, payload policy, redaction hint, and validation report; the 1.1 target policy, effect limits, and Authority Context requirement; and the runtime action proposal in its 1.1 and tenant-aware 1.2 forms. The validation report schema includes optional semantic aliases.

The schemas serve four purposes:

1. **Interchange.** They allow external systems to understand the shape of manifest and proposal records.
2. **Validation.** They support structural validation outside the Python package.
3. **Documentation.** They make the public boundary explicit.
4. **Conformance.** They provide a basis for compatibility tests across implementations; the runtime-consumption contract lists the negative cases a conforming consumer should reject or hold.

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. Schema validity is not authorization.

18 Example Manifests

The repository includes four declaration examples for common enterprise scenarios and one integration fixture. All five validate with no errors and no warnings under `aam check-examples`. The four domain

examples demonstrate declaration structure; they are not evidence of deployed sector assurance.

18.1 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 drafting and sending email, uses draft-first review for drafts and supervisor approval for sends, and marks customer identifiers, recipients, and message bodies as redaction-relevant. The draft action declares reliance on tools, databases, or user input.

18.2 Internal Research Agent

A research assistant searches documents, reads files, generates summaries, and exports reports. The manifest declares reliance requirements for retrieval and read actions, human review for summaries, and approval for export.

18.3 Procurement Agent

A procurement agent looks up vendors, compares quotes, drafts purchase recommendations, creates purchase orders, and sends vendor emails. Purchase-order creation requires two authority scopes and approval; outbound vendor email requires authority and approval.

18.4 Finance Workflow Agent

A finance workflow agent reads invoice records, drafts payment recommendations, approves payments under dual-control authority, and exports audit records with redaction of sensitive amounts and identifiers.

18.5 Synthetic Refund Integration Fixture (1.1)

The `refund_integration_v1_1` fixture is the manifest used by the stack's supported workflow. It declares one synthetic refund adapter with a stable adapter identity and two refund actions: a routine action (consequence tier T1, human review, one permitted synthetic account, at most 100 USD and one effect) and a higher-consequence action (tier T2, approval by a named reviewer role, a different single permitted account, at most 1,000 USD and one effect, two evidence obligations). Both reference the same Authority Context profile and distinct requirement identifiers. The fixture is integration evidence for the bounded synthetic reference, not a deployed agent and not an institutional grant.

19 Relationship to Runtime Control

Action Manifest is not a runtime control layer. Its role is to declare what an agent may propose before runtime and to make that declaration exactly bindable. In the integrated reference, the Control Plane's bounded authorization/effect 0.1 interface is pinned to the selected Manifest 1.1 revision and consumes the declaration as follows.

- **Binding.** The Control Plane's authorization binding records the manifest identifier, version, and digest, the adapter, target, and payload commitment, the Authority Context and requirement identifiers, the grant revision, and policy versions, and it records the full proposal commitment.

- **Fit before authority.** Authorization requires the proposal to fit the manifest declaration, the Authority Context requirement's permission envelope, and the issued grant. A grant broader than the requirement does not widen it.
- **Consistency checks.** The manifest consequence tier must equal the Authority Context requirement tier; manifest review roles must map explicitly to authority role identifiers.
- **Limits.** `max_amount` and `max_effects` are first checked per proposal; for the local pilot the execution envelope uses the narrowest effect limit across manifest, requirement, and grant. Per-proposal limits are not a cumulative or distributed budget.
- **Effect-time revalidation.** Immediately before dispatch the Control Plane resolves inputs again and compares the complete binding; a changed manifest, payload, target, adapter, grant, approval, policy, or required evidence prevents the effect.

Two boundaries from the stack's release status apply to anything built on this declaration. First, the default (C0) path performs pre-effect revalidation, which is not destination-commit atomicity: a check-to-commit race remains. An optional same-host C1 profile orders cooperating authority mutations and the synthetic effect through one SQLite transaction boundary; it is not enabled by default, and no C2/C3 (remote or distributed) claim is made. Second, effect-identifier deduplication is not business-intent deduplication: two distinct, individually valid proposals with equivalent refund intent can both fit the manifest. A refund-intent profile exists in the selected runtime source as a separate optional profile; the default path does not enable it.

20 Security Considerations

Action Manifest improves action visibility, governance preparation, and handoff precision, 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 and Authority Context references do not grant or verify authority;
- declaration matching detects structural substitution; it does not establish current grants, mandates, approvals, or policy status;
- 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;
- per-proposal effect limits are not cumulative budgets;
- schema validation checks structure and consistency, not operational sufficiency;
- a valid signature, chain inclusion, message receipt, reasoning instruction, or evidence-package digest elsewhere in the stack does not authorize execution either;
- deployments still require authentication, authorization, logging, monitoring, runtime control, credential custody, and incident response, none of which is established by this component.

The correct security posture is layered. Action Manifest supplies structured pre-runtime declaration and exact binding material. It should be combined with independently resolved authority, application authorization, effect-time checks, run recording, reconstruction, monitoring, operational controls, and governance review.

21 Implementation Boundary

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

Public reference layer	Private product layer
Action manifest schema	enterprise action registry
Tool definitions and adapter identities	customer-specific tool discovery
Action declarations	proprietary action classification
Authority requirements and Authority Context references	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
Commitments and declaration matching	deployment-specific binding services
Validation reports	compliance evidence automation
CLI utilities	hosted management interfaces
Example manifests and fixtures	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 and bind to. Private product systems may add enterprise integrations, hosted workflows, richer policy authoring, automated discovery, approval routing, and deployment-specific governance logic.

22 Reference Implementation Status

The public reference implementation is intentionally small. It is meant to be inspectable, testable, and safe to publish. The package version (`agent-action-manifest 0.1.0`) and the manifest/runtime-profile version are separate version axes.

Capability	Status	Notes
Core object model	Implemented	Pydantic v2 models
Manifest loader and dumper	Implemented	JSON loading, model validation, formatted export
Validation reports and aliases	Implemented	error and warning issues, semantic aliases
Action types, authority, review, reliance	Implemented	as in Sections 7–11
Payload policies and redaction hints	Implemented	declaration only

Capability	Status	Notes
Manifest 1.1 runtime profile	Implemented; selected	action/adaptor IDs, targets, limits, Authority Context references
Commitments and declaration matching	Implemented; selected	manifest, payload, proposal digests; named mismatches
Runtime action proposal 1.2	Implemented; not selected	tenant-aware generation in component main only
Optional OpenAPPA declaration binding	Implemented; not selected	opt-in helper, grants nothing; OpenAPPA work closed for now
CLI	Implemented	validate, summarize, list-actions, check-examples
JSON schemas	Implemented	manifest, binding objects, proposals
Examples	Implemented	four domain examples, one synthetic refund fixture
Hosted registry	Extension	out of current scope
Runtime enforcement	Extension	Control Plane and runtime components
Enterprise approval workflows	Extension	out of current scope

Selected means selected by the hub’s component lock. The lock (profile `merged-producers-v1`) selects this component at the revision exposing “Manifest 1.1”. Newer accepted revisions in the component repository are not selected automatically. The stack as a whole is a merged, pinned bounded synthetic reference: one host, a synthetic SQLite destination, one supported refund workflow. It is not a production deployment, a compliance certification, or proof of institutional adoption. Release closeout froze feature intake on 13 October 2026; public-preview approval is pending, and final publication and operational trust remain on hold.

23 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 suite covers model and loader behavior, invalid JSON, enum and model validation errors, each validation rule and its alias, the warning/error distinction, the CLI commands, example validation, and, for the 1.1 profile, exact binding plus adversarial cases: unknown action, substituted adaptor, changed payload, target substitution, manifest version and digest mismatch, amount above declaration, authority requirement substitution, stale payload commitment, missing requested permission, and proposal digest change when a bound field changes. Further tests cover the tenant-aware 1.2 proposal and the optional OpenAPPA binding helper.

A GitHub Actions workflow installs the package, runs the test suite on Python 3.11 and 3.12, and runs `aam check-examples`. This gives a compatibility and regression signal without implying production certification.

Stack-level evidence is separate and must not be read as a count for this component. The current release evidence for the selected configuration (CI run 37694032916) resolved all 35 acceptance scenarios with the required suites executed twice. A fresh reproduction on 13 October 2026 ran 1,009 Python test cases in each of two repetitions and 20 separate Node tests, with 34 invariants passing and one characterized limitation. The optional C1 profile was qualified in a separate run of 73 tests. Earlier collateral citing 915 tests describes

a historical persistence generation and is not the current count. These figures establish bounded synthetic behavior at exact pins, not deployment benefit or compliance.

24 Example Use Cases

24.1 Agent Deployment Review

An AI platform team prepares to deploy a customer-service agent. Before 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 reads require reliance evidence and redaction of customer identifiers.

24.2 Security Review

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

24.3 Governance Evidence Preparation

A governance team needs business-readable evidence for an agent run. The Governance Evidence Pack imports the manifest together with the reconstructed run through a versioned, traceable transformation, so the declared action inventory can be compared with what was proposed, decided, attempted, and observed. The pack preserves that relationship; it does not convert a summary into independent assurance.

24.4 Runtime Control Integration

A developer integrates an agent with the Control Plane. The 1.1 manifest supplies stable action and adapter identities, finite targets, effect limits, and Authority Context references; the runtime constructs a proposal bound to the manifest digest; the Control Plane resolves authority independently and revalidates before effect.

25 Evaluation Criteria

The usefulness of 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, with stable adapter identities where runtime binding is needed?
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. Do consequential actions declare finite targets, effect limits, and Authority Context references?
11. Does it validate with no error-severity issues, and are its warnings useful for review?
12. Can validation issues be interpreted consistently through stable codes or aliases?
13. Can the receiving runtime bind proposals to the exact manifest digest and reject the documented negative cases?
14. Who supplies trusted authority, and which assumptions remain outside the manifest?
15. Can governance teams inspect it before deployment, and trace later evidence back to it?

26 Roadmap

The roadmap should remain implementation-focused and avoid overclaiming. Implemented capabilities include the object model, tool and action declarations, authority, review, and reliance requirements, payload policies, redaction hints, validation reports and aliases, the CLI, examples, JSON schemas, CI, the Manifest 1.1 runtime-consumption profile, canonical commitments and declaration matching, the tenant-aware proposal 1.2 contract, and an optional OpenAPPA declaration binding helper. The tenant-aware 1.2 contract is accepted, and the OpenAPPA helper is merged, in the component repository; neither is selected by the hub.

The repository roadmap lists near-term extensions that remain unimplemented: richer action categories, an optional manifest-to-policy-configuration converter, manifest diffing, a manifest risk summary, OpenAPI or tool-schema mapping examples, a formal conformance test suite, and additional industry examples. Longer-term items include UI editors, registry patterns, integration examples, and enterprise governance workflows. Those are product-layer concerns and should remain separate from the minimal public reference.

At the stack level, the current release closeout has frozen feature intake. Further OpenAPPA development is closed for now, and a declaration profile that requires expiry horizons for particular actions would need an agreed Manifest change rather than an unversioned field. Any newer revision must be separately qualified and selected before stack claims extend to it.

27 Conclusion

AI-agent governance cannot begin only at runtime. Before an agent proposes actions, uses tools, sends messages, exports records, issues refunds, or touches enterprise systems, teams need a structured declaration of what the agent may propose and what governance posture applies to each action. Once runtime controls exist, they also need that declaration to be exact.

Cognous Action Manifest provides a minimal public reference for that declaration layer. It describes agent tools, action declarations, authority requirements, review requirements, reliance requirements, payload policies, and redaction hints. Its 1.1 profile adds stable identities, finite targets, effect limits, Authority Context references, canonical commitments, and deterministic declaration matching, which the Control Plane consumes in the stack's bounded synthetic reference.

The contribution is deliberately narrow. Action Manifest does not run agents, enforce policies, resolve or grant authority, produce runtime records, or guarantee correctness. It supplies a concrete declaration artifact for a specific operational need: making an AI agent's action surface inspectable before deployment

and bindable at runtime, while leaving permission to be established separately, from authority the agent does not control.

References

- Cognous. *Cognous Action Manifest*: declare what actions an AI agent may propose before runtime execution. Public reference implementation, 2026. <https://github.com/cogno-us/cognous-action-manifest> (formerly `cognous-agent-action-manifest`).
- Cognous. *Cognous Open Control Stack*: hub, component lock, release status, and release closeout (13 October 2026). <https://github.com/cogno-us/cognous-open-control-stack>
- Cognous. *Cognous Control Plane*: bounded authorization, effect-time revalidation, and persistent runtime records. <https://github.com/cogno-us/cognous-control-plane>
- Cognous. *Cognous Governance Evidence Pack*.
<https://github.com/cogno-us/cognous-governance-evidence-pack>
- Cognous. *Alvorada (Institutional Governance)*, Authority Context 0.1.0. <https://github.com/cogno-us/cognous-institutional-governance>
- National Institute of Standards and Technology. *Artificial Intelligence Risk Management Framework*. NIST AI 100-1, 2023.
- OWASP GenAI Security Project. *State of Agentic AI Security and Governance*, version 2.01, June 2026.
- OECD. *Agentic AI in organisations: Early insights from practitioner interviews*. OECD Artificial Intelligence Papers, No. 65, 2026.