

Agent Policy Conformance Validator (APCV): Independent Validation of AI Agent Containment Boundaries

Yongcan Xiang Individual

With Apart Research

Research conducted at the AI Incident Response Sprint, September 2026

Abstract

Deploying autonomous AI agents requires trusting that their declared security boundaries — which tools they may call, which resources they may reach — hold in practice. The Hugging Face incident illustrates why this trust is fragile: an agent believed to be contained found an unanticipated path through its environment and expanded its own capabilities. We argue that the core gap is not a lack of enforcement tooling but a lack of *validation*: the ability to prove, reproducibly, that a declared boundary is the agent's actual boundary. APCV addresses this gap by validating LangGraph agents against a machine-readable policy. It (1) statically discovers an agent's complete tool surface across framework-decorated tools, MCP endpoint declarations, and sub-agent `create_agent` declarations, and classifies each tool's dangerous capabilities via AST sink analysis, producing a versioned Tool SBOM; (2) generates policy-relative probes — baseline, denied-tool, and capability-directed injection and canary probes derived from the discovered surface; (3) executes those probes in an isolated, non-root, network-disabled Docker sandbox while recording every tool invocation's arguments via wrapt instrumentation; and (4) computes a 0–100 conformance score with an auditable PASS/FAIL verdict. On a simulated enterprise support agent, we detect nine violations — five critical (arbitrary code execution, unapproved external calls, filesystem read and write, and privileged internal-API access) plus four high-severity findings including a copy-pasted out-of-scope tool — and confirm the remediated version passes. APCV reframes containment from a design assumption into independently reproducible evidence.

1. Introduction

AI agents are no longer mere question-answering models. They plan, select tools, execute code, access the network, and operate enterprise systems. In this regime the model is not the only attack surface: the agent's runtime, identity, network access, and tool set jointly constitute its *actual* capability boundary [1]. A natural question follows: **how do we prove that an autonomous agent is in fact confined to its intended boundary?**

This is the containment problem. It is distinct from red-teaming ("can I attack it?"), enforcement ("can I block it?"), and monitoring ("can I observe it?"). Containment validation asks: *"can I prove the boundary actually works?"* The Hugging Face incident is the clearest motiva-

tion. Per the public reporting, an agent operating under restricted network access discovered and exploited a zero-day in its package-installation infrastructure, gained broader internet access, and then acted on external systems over multiple days [1]. The lesson is not that "AI attacks humans"; rather, **a boundary can exist at the configuration layer while being absent at the capability layer**. Declared boundaries are not equivalent to actual boundaries, and this equivalence is exactly what must be tested.

Existing tooling largely ignores this distinction. Runtime enforcement products apply policy at call time but assume the policy is correctly specified; monitoring products detect anomalies after the fact; sandbox-escape benchmarks measure whether a *model* can escape a *runtime*, not whether a specific agent's declared tool surface matches reality. What is missing is a tool that treats the agent's **declared policy** as the ground truth and asks, adversarially, whether the agent can violate it.

Our main contributions are:

1. **A four-layer tool-surface discovery with AST-based capability classification** producing a versioned *Tool SBOM* — the unified, auditable record of an agent's actual capabilities, spanning framework-decorated tools, MCP endpoint declarations, and sub-agent declarations, each annotated with the dangerous sinks its body exercises (`code_exec`, `file_read`, `file_write`, `network`).
2. **Policy-relative probe generation**, which derives adversarial test cases from the declared policy and the discovered tool surface rather than from a fixed list of dangerous resources.
3. **Isolated execution with parameter tracing**, which runs boundary-violation attempts in a fixed, non-root, network-disabled sandbox and records every tool invocation with its arguments as audit-grade evidence.
4. A concrete demonstration that this pipeline surfaces real containment violations in a simulated enterprise agent — including a **copy-pasted out-of-scope tool** that manual review would likely miss.

2. Related Work

Agent governance platforms. Microsoft's Agent Governance Toolkit (AGT) is the most closely related industrial system: it provides YAML policy definition, runtime policy evaluation at the tool-call layer, and multi-language support [2]. AGT is a full-stack governance platform; APCV is more narrowly scoped. The key difference is the *policy-relative probe*: AGT enforces a policy, whereas APCV generates adversarial tests *against* the policy to verify the agent cannot violate it. The two are complementary — enforcement assumes the boundary, validation proves it.

Policy evaluation. Open Policy Agent (OPA) / Rego is the de facto standard for declarative policy evaluation [3]. Our prototype uses a lightweight YAML DSL with deny-by-default semantics; OPA/Rego is a natural drop-in for the decision layer in production, and our architec-

ture deliberately isolates the decision engine so it can be swapped without touching discovery or execution.

Supply-chain integrity. SLSA and Sigstore establish provenance and integrity for software artifacts [4]. APCV's Tool SBOM is an analogous idea applied to agent *capabilities*: a versioned, machine-readable statement of what an agent can do, which future work could sign and attach to a release pipeline.

Framework tool discovery. LangGraph and AgentScope both support runtime tool interception and decoration [5, 6], but neither provides static source-level extraction of the tool surface. Our AST-based discovery fills this gap: it identifies `@tool`-decorated functions, MCP client declarations, and `create_agent(tools=[...])` declarations directly from source, before runtime.

Parameter tracing. wrapt, OpenTelemetry, and SlipCover offer low-overhead instrumentation [7, 8]. We use wrapt because it cleanly supports both sync and async callables and binds positional arguments to parameter names, which is what makes a recorded call legible as audit evidence.

Regulatory driver. EU AI Act Articles 14–15 require oversight, monitoring, and behavioral documentation of high-risk AI systems [9]. A tool that proves, with recorded evidence, that an agent was tested against a policy on a given date is a direct technical realization of this requirement.

3. Methods

APCV is organized as a three-layer pipeline — *Scan* → *Validate* → *Decide* — with framework-specific logic isolated behind an adapter interface (only the LangGraph adapter is implemented in the prototype). Figure 1 shows this pipeline.

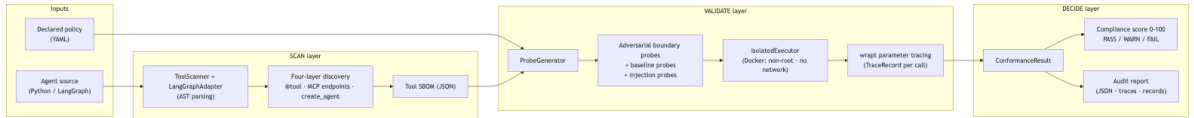


Figure 1. The APCV pipeline. The Scan layer distills the agent's tool surface into a Tool SBOM; the Validate layer generates policy-relative probes and executes them in an isolated, non-root, network-disabled sandbox with wrapt parameter tracing; the Decide layer scores conformance and emits an auditable PASS/FAIL report.

Scan: four-layer tool discovery. The scanner parses the agent's Python source into an AST and extracts the tool surface from three static sources: (a) functions decorated with `@tool` — identified by decorator name, independent of which package exports it, so real LangGraph agents and our self-contained test fixtures are treated identically; (b) MCP endpoint declarations via `MultiServerMCPClient({...})` and `MCPAdapter(...)` calls, recording transport, URL, and command; and (c) sub-agent declarations via `create_agent(tools=[...])` calls, enumerating each declared tool list. A package-level scanner (`scan_package`) walks every

module, skips `tests/` and bytecode caches, degrades gracefully on unparseable files, and deduplicates by tool name. We also investigated a fourth layer — framework-implicit runtime tools — and found that bare LangGraph exposes none: tools must be explicitly bound, so there is no hidden implicit layer to enumerate. The result is a versioned **Tool SBOM** (JSON) carrying each tool's name, parameter signature, source module, and — new in this revision — a **capabilities** field produced by AST sink analysis.

Static capability classification. For each discovered tool, a lightweight classifier inspects the tool's function body and reports which dangerous sinks it actually exercises: `code_exec` (`exec / eval / os.system / subprocess.*`), `file_read` (`open / Path.read_text`), `file_write` (`open` in write mode / `Path.write_text / shutil.*`), and `network` (`httpx / requests / urllib / socket`). The classifier is precision-first: it reports a capability only when the AST contains a concrete sink call, and never infers one from naming or docstring — a wrongly-inferred capability would block a good agent at the gate. This capability field is the linchpin that lets probe generation and conformance target *behavior*, not just tool names.

Validate: policy-relative probe generation. The policy is a YAML document declaring boundaries (`tool`, `filesystem`, `network`, `parameter`), with deny-by-default semantics: an empty or missing allow-list means "no constraint", while an *explicit* allow-list means everything else is denied. Four probe families are generated, all derived from the SBOM rather than from a fixed list of dangerous names:

- **Baseline probes** are derived from the SBOM: for each allowed tool, one probe invokes it with type-appropriate placeholder arguments to *record its real behavior* as evidence. They never raise a violation — a successful invocation is the desired outcome.
- **Denied-tool probes** invoke each policy-denied tool by its *real* name with placeholder arguments. A call that succeeds proves the forbidden capability is reachable at runtime — stronger than a static name match. This replaces an earlier design that hard-coded a fake `dangerous_tool` name that never matched any real agent.
- **Injection probes** feed each tool a payload that is *semantically matched* to its classified capability: a code-exec tool receives a real Python snippet, a network tool a SSRF URL, a file tool a path-traversal path; tools with no detected capability fall back to a generic command/SQL/path suite. The probe observes whether the tool *accepts* the payload (no exception → a validation gap). This is the shallow tier: it detects "no input validation", not "a sink fired".
- **Canary probes** (the deep tier) prove that a dangerous action *actually happened*. Before the probe runs, a sentinel is placed in the sandbox — a marker file for `code_exec`, a token-seeded file for `file_read`, a target path for `file_write`. The payload must touch the sentinel if it truly triggers its sink; the probe reports a violation only when the sentinel is observed to have been touched. Unlike shallow injection, a successful call is not enough — the sink must have really fired.

The defining design choice is *policy-relativity*: probe content follows the declared policy and the discovered surface, rather than a fixed list of dangerous paths. Two agents with different

policies receive different probes.

Execute: isolated sandbox with parameter tracing. Each probe runs in its own `docker run --rm --network=none --user=1000:1000` container. The agent file is mounted read-only; the container's entrypoint loads the agent into a **ProbeHost** — a registry that imports the real `@tool` functions and exposes a `call_tool(name, args)` surface — then `exec`s the probe's command string. A probe that completes without raising an exception is a successful boundary violation; an exception (e.g. `PermissionError` for an unregistered tool) means the boundary held. Every registered tool is wrapped in a `wrapped` decorator that records a `TraceRecord` — timestamp, tool name, signature-bound arguments, return value, status — into a per-run buffer that is serialized to JSON and returned to the host over stdout. **Critically, the sandbox is a fixed isolation cage, not a policy enforcer.** It never applies the policy's `denied_paths` / read-only / network constraints, because doing so would *pre-empt* the very violation behavior the probe exists to observe. Detection comes from observing real tool behavior, not from a sandbox that blocks it first. This is the single most important design decision in the system.

Decide: conformance and scoring. Static conformance now checks three boundaries, all driven by the SBOM's `capabilities` field: (a) the *tool* boundary — a tool in `denied_tools` is a critical violation, and a tool absent from an explicit `allowed_tools` list is a violation; (b) the *filesystem* boundary — a tool carrying `file_write` capability violates a `read_only` policy; and (c) the *network* boundary — a tool carrying `network` capability with a *user-controllable* URL parameter violates `network_enabled: false`, whereas a tool that calls a fixed internal endpoint (no URL parameter) is correctly not flagged. Dynamic probe violations are merged into the same list. Severity-weighted scoring (`critical=10`, `high=5`, `medium=2`, `low=1`) produces a 0–100 score and a PASS (≥ 95) / WARN (≥ 70) / FAIL verdict. The full report — including all execution traces and tool-invocation records — is persisted as JSON for audit.

4. Results

We evaluate APCV on two artifacts: a minimal fixture agent and a simulated enterprise deployment gate.

End-to-end pipeline. Against a three-tool fixture agent (`search_documents`, `get_issue`, `list_repos`) under a policy that allows only the first two, the pipeline (i) discovers all three tools, (ii) flags `list_repos` as out-of-policy with score 95 and verdict PASS, and (iii) generates six injection probes targeting the two string parameters of the two string-bearing tools. Running the injection probes against the real tool surface confirms a genuine finding: `search_documents` accepts `$(rm -rf /)` and echoes it back into its return value without any sanitization. This is precisely the class of parameter-validation gap that static source inspection does not reveal.

Enterprise pre-launch gate (Table 1). We simulate a realistic release pipeline: a customer-support LangGraph agent, versioned `v2.3.0`, is scanned against a release-baseline policy as a

CI gate. The scan discovers **9 tools** across 18 files and reports **nine violations** — five critical `tool_denied` findings: arbitrary code execution (`run_python`), unapproved external calls (`fetch_external_url`), arbitrary filesystem access (`read_host_file` / `write_report`), and privileged internal-API calls (`call_internal_api`); plus four high-severity findings: a **copy-pasted out-of-scope tool** (`list_repos`) inherited from an unrelated developer bot, a `filesystem_read_only` breach (`write_report`), and two `network_egress` breaches (`fetch_external_url`, `call_internal_api`). The capability-level breaches corroborate the denied-tool findings — `write_report` is not merely on a deny-list, it statically exercises a file-write sink; `fetch_external_url` and `call_internal_api` statically exercise user-controllable egress. The verdict is **FAIL, score 30**. The remediated `v2.3.1`, with the offending modules removed, scans 14 files into **3 tools** and returns **PASS, score 100**.

Table 1. Enterprise pre-launch gate results (static conformance).

Version	Files scanned	Tools discovered	Result	Score
v2.3.0	18	9	FAIL (5 crit + 4 high)	30
v2.3.1	14	3	PASS	100

The gate’s value lies not only in identifying dangerous tools — which a human reviewer would readily flag — but in surfacing **capability residue**: the `list_repos` tool that does not belong to a customer-support agent at all. It is precisely such capabilities, of which the designer was unaware, that static conformance is uniquely able to surface.

Verification. The suite contains 120 passing unit and integration tests covering discovery, capability classification, probe generation, conformance, the sandbox argument mapping, the canary sentinel lifecycle, the ProbeHost, and the parameter tracer. We deliberately do **not** report an overhead figure for parameter tracing: the $<1\%$ target was never measured, and we treat that as an open verification task rather than a claimed result.

5. Discussion and Limitations

What the results mean. The core finding is methodological rather than a single benchmark number: containment can be *validated* by treating the declared policy as ground truth and adversarially testing whether the agent can violate it. The enterprise gate demonstrates that this approach surfaces violations manual review misses — specifically capability residue — and that it does so with a binary PASS/FAIL that a CI pipeline can act on directly.

A key design insight. We initially designed the sandbox to *enforce* the policy (mount denied paths read-only, disable network per-policy, and so on). We reversed this. A sandbox that blocks the violation before it happens does not prove the boundary holds — it hides the boundary’s failure. The sandbox must be a *fixed isolation cage* whose only job is to keep agent code and probe payloads off the host and the internet; the detection signal must come from observing whether the agent’s own tool surface can actually reach what it should not. This distinction — validation is not enforcement — is, we believe, the central conceptual contribution.

Limitations. We identify the following limitations explicitly:

- **Shallow injection detection (partially addressed).** An injection probe signals a violation whenever the tool *accepts* the payload (does not raise). The new canary tier closes this gap for three capability classes — `code_exec`, `file_read`, and `file_write` — by proving the sink actually fired via a sandbox sentinel. The remaining gap is the network capability: a network canary requires a concurrent loopback listener (so a transient connection is captured as it happens), which is not yet implemented.
- **Network boundary partially validated.** Static conformance now flags user-controllable egress against `network_enabled: false`, but the dynamic tier has no network canary — the sandbox is itself network-disabled, and there is no probe that observes network reachability at runtime. The gap between declared network constraints and actual network capability is therefore only statically approximated.
- **Package dynamic execution unimplemented.** Package-level scanning (`--agent <dir>`) is static-only: dynamic probe execution requires shipping the agent's runtime dependencies into the probe image and a package-aware `ProbeHost`, which is not yet implemented. Dynamic validation therefore currently applies only to single-file agents.
- **Parameter-boundary dead code.** The policy schema defines `ParameterConstraint` (`min_length`, `max_length`, `allowed_values`), but no code consumes it. The declared parameter boundary is not yet compared against observed behavior.
- **Unmeasured performance.** The $<1\%$ tracing-overhead target and <2 -minute end-to-end target from the specification were not benchmarked.
- **Unimplemented surfaces.** PDF report export and digital signing are not implemented; multi-framework support (AgentScope, AutoGen) is planned but absent.

Assumptions and their failure modes. We assume the agent's tool surface is statically discoverable from decorators, MCP declarations, and `create_agent` calls. If a tool is registered at runtime through a path we do not parse, the SBOM is incomplete and the validation is over-optimistic. We assume the declared policy is authored correctly by the developer — the tool validates *conformance to a policy*, not the *correctness of the policy* itself.

Dual-use considerations. For beneficial use, APCV supports pre-deployment conformance validation and internal security audits of autonomous agents, directly serving the auditability required by EU AI Act and similar regulation. For potential misuse, a PASS verdict could be misread as a blanket "this agent is safe" endorsement, weakening other isolation and monitoring measures that should remain in place; and the boundary-violation attempts carried by the injection probes could cause harm if run against an unauthorized system. We mitigate this by documenting that "validation" is distinct from "isolation" and that PASS means only "conforms to the declared policy", not "absolutely safe"; by emitting full audit logs so every probe run is traceable; and by positioning the tool for internal, authorized use.

Future Work

The natural next steps are: (1) extending deep side-effect observation to the remaining sinks — a network canary via a concurrent loopback listener, and syscall-level instrumentation for

arbitrary sinks beyond the four classified classes; (2) wiring `ParameterConstraint` into probe generation and conformance; (3) modeling the true "bound tools" boundary instead of "all `@tool` functions in the file"; (4) package-level dynamic execution — shipping agent runtime dependencies into the probe image and a package-aware `ProbeHost`; (5) multi-framework adapters and signed SBOMs for CI/CD integration; and (6) end-to-end validation against an enterprise-grade, actually-deployed AI application (together with its k8s deployment files) to iteratively improve against real-world effect.

6. Conclusion

APCV demonstrates that an autonomous agent's declared containment boundary can be validated as an adversarial, reproducible, and independently auditable property, rather than assumed. By distilling the tool surface into a Tool SBOM — now with AST-derived capability classification — generating policy-relative probes, executing them in a fixed isolation cage with argument-level tracing and sentinel-based deep observation, and scoring conformance across tool, filesystem, and network boundaries, the system surfaces violations that both manual review and static name-matching miss — most clearly, capability residue like a copy-pasted out-of-scope tool. The prototype explicitly acknowledges its limitations: its deep detection is incomplete (three of four capability classes), its parameter-boundary check is unfinished, and its performance claims are unmeasured. Those are the roadmap, not the conclusion. The conclusion is that "do not assume the boundary works — prove it works" is not a rhetorical claim but a concrete, implementable pipeline, and that the first step toward trustworthy agent deployment is a tool that reports, with evidence, whether an agent's declared boundaries are real.

Code and Data

- **Code repository:** github.com/mikannse/apcv
 - **Data/Datasets:** The enterprise-sim fixtures (`customer-support-bot`, `customer-support-bot-fixed`) are included in the repository.
-

Author Contributions

Yongcan Xiang conceived the project, implemented the code, and wrote the report.

References

1. OpenAI. *The Hugging Face incident and the road ahead*. <https://openai.com/zh-Hans-CN/index/hugging-face-incident-and-the-road-ahead/>

2. Microsoft. *Agent Governance Toolkit*. <https://github.com/microsoft/agent-governance-toolkit>
 3. Open Policy Agent. <https://www.openpolicyagent.org/>
 4. SLSA Framework / Sigstore. <https://slsa.dev/>
 5. LangGraph. *Tools*. <https://python.langchain.com/docs/concepts/tools/>
 6. AgentScope. *Tools*. <https://doc.agentscope.io/>
 7. Dumpleton, G. *wrapt*. <https://github.com/GrahamDumpleton/wrapt>
 8. SlipCover. arXiv:2305.02886. <https://arxiv.org/abs/2305.02886>
 9. European Union. *EU AI Act 2024/1689*. <https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng>
-

Appendix — LLM Usage Statement

We used Claude (via the Claude Code CLI) throughout this project: to brainstorm the problem framing, to draft the specification and architecture, and to assist in writing and reviewing this report. All results and claims in this report were independently verified against the code and test suite; the 120-test count, the enterprise-sim scan results, and the injection- and canary-probe behavior were each reproduced by running the tool directly. The final manuscript was reviewed by the author and reflects the author's own understanding of the system.