

Measuring What a Runtime Agent Guardrail Actually Prevents

A Measurement Protocol, an Attack-Dataset Discipline, and a Failure Taxonomy from Three Generations of Agent-Governance Benchmarks

Prismor Research

10 August 2026

Abstract

Enterprises increasingly place a *runtime governance layer* between an AI coding agent and its privileged operations, and increasingly treat that layer as a compliance control. The evidence published for such systems, however, is dominated by rule inventories, category coverage tables, and compliance mappings. All of these describe what a system is configured to detect rather than what it actually prevents. The distinction matters because a governance layer has a failure mode that ordinary software does not: it can keep emitting findings, populating an audit trail, and reporting a healthy configuration while enforcing nothing at all, and under every commonly published metric that state is indistinguishable from a working deployment.

This paper sets out a measurement protocol for that category, developed across three benchmark generations against production coding agents, together with the discipline we use to build attack datasets. The protocol scores a real side effect rather than model prose, reads two independent measurement planes, carries a no-injection utility control, and promotes its own construct-validity checks into executable assertions. The system under test is Prismor, an open-source runtime governance layer that we develop; Section 11 states what that authorship means for how these numbers should be read.

Applying it across two agent families and seven configurations over 493 scored trials, we find that none of four host-native controls issued a single denial in 207 attack trials; that the governance layer reduced pooled attack success from 41.3% to 10.3% on one family and from 27.8% to 8.3% on the other, closing the supply-chain category outright; and, most consequentially, that on one family it initially produced 91 findings and *zero* blocks because one stale configuration field held it in observe mode, invisibly at every operator-facing surface. Clearing that single field moved pooled harm from 26.4% to 8.3%, a larger effect than the difference between running Prismor and running nothing.

We generalise from measurement to a seven-class failure taxonomy for this category, and show each class instantiated independently in two unrelated codebases: our own, and the public defect record of a large open-source agent-governance project examined as a corpus of field-observed failures. We close with what enterprise runtime agent security still has to cover, derived from measured gaps rather than from a threat model. Results are preliminary: $N=3$ per cell.

1 Introduction

An AI coding agent with shell access is a program that decides, at runtime, which privileged operations to perform, using inputs it does not control. The resulting failure modes are catalogued by the OWASP Agentic Security Initiative as ASI01 to ASI10 [1], and the mechanism behind several of them, that retrieved content can carry instructions a model will follow, was established for LLM-integrated applications by Greshake et al. [3].

The common response is a runtime governance layer: a policy engine, installed as agent hooks or as a proxy, that inspects each tool call before it executes and may allow, rewrite, warn, or deny it. Several such layers now exist. The system under test in this paper is one of them, Prismor, which we develop; Section 3 describes it to the depth the results require, and Section 11 states what our authorship of it means for how these numbers should be read.

1.1 The measurement gap

What is not established, for Prismor or for any system in this category, is how to show that one of them works. The evidence typically published is a rule count, a category coverage table, or a mapping from rules to a regulatory framework. Each describes *configuration*. None measures *prevention*.

The gap is not academic. A governance layer can fail in a way ordinary software cannot: it can detect correctly and enforce nothing, while every surface an operator would inspect reports health. We encountered exactly this state in our own production deployment, and we nearly published the wrong explanation for it. Under harm rate, block rate, rule inventory, and compliance mapping alike, a guardrail that had been silently disabled for a month was indistinguishable from one working normally.

This paper therefore answers a methodological question before it answers a product one: **how do you measure a runtime agent governance layer such that a silently disabled one cannot pass?**

1.2 Contributions

1. **An account of the experimental program** (Section 4): three benchmark generations, the harness architecture, arm construction, and the measurement failure that forced each rebuild.
2. **A measurement protocol** for runtime agent governance: side-effect ground truth, two independent measurement planes, a no-injection utility control, construct-validity guards promoted to executable assertions, and mechanism attribution (Section 5).
3. **A dataset-construction discipline** (Section 6), including the instrumentation failures that silently invalidate studies in this category. Every one of them produces a result that looks like a successful defense.
4. **Measurements** across two agent families and seven configurations, 493 scored trials, including a case where one configuration field determined the entire outcome (Section 7).
5. **A seven-class failure taxonomy** (Section 8), with every class instantiated independently in two unrelated codebases, which is evidence that these are properties of the category rather than of one implementation.
6. **A coverage map** of what enterprise runtime agent security still has to address, derived from measured gaps (Section 9).

2 Background and Related Work

Threat taxonomies for agentic systems. The OWASP Agentic Top-10 [1] enumerates the categories we score against; the OWASP Top 10 for LLM Applications [2] covers the model layer rather than the tool-invocation layer. Both specify categories without specifying executable tests, so coverage against them is self-asserted. We bind each category to an attack with a verified side effect, which converts a coverage claim into a measurement.

Prompt-injection and agent security benchmarks. AgentDojo [4] evaluates prompt-injection defenses over synthetic tool suites; InjecAgent [5] measures indirect injection in tool-calling agents. Both are carefully constructed and both score against *simulated* tools. That design choice is appropriate for studying model susceptibility and insufficient for studying a deployed control plane: when the tool is simulated there is no enforcement path to fail, so the distinction this paper is about, between a finding and an enforced decision, cannot arise.

Guardrail and policy-enforcement layers. NeMo Guardrails [6] and Llama Guard [7] filter model inputs and outputs. They are complementary to tool-call interception rather than substitutes for it: a policy that reads the literal shell command about to execute can make decisions a text classifier over the conversation cannot. What the published material for this class of system rarely reports is measured prevention, and rarer still is any measure of over-blocking.

Fail-safe defaults. The failure class we find dominant is not new as a principle. Saltzer and Schroeder [8] named *fail-safe defaults*, that decisions should be based on permission rather than exclusion, fifty years ago. Our contribution here is empirical: the specific ways this principle is violated in agent governance layers, and the fact that the violations are invisible to the metrics the category publishes.

3 System Under Test

This section characterises the artifact being measured. It is included only to the depth the results require: several findings in Section 7 are uninterpretable without knowing where the enforcement decision is actually taken, and Section 8 depends on knowing which mechanism produced a given block. Features not exercised by the experiments are omitted.

Interception point. Prismor evaluates a pending tool call before the host executes it: the host invokes a dispatcher, the dispatcher returns a verdict, and the host honours it. The tool call is the last point at which an action is still a proposal, which is the property the protocol in Section 5 exploits by defining ground truth over the side effect the call would have produced. One evaluation entry point backs three surfaces (agent hooks, framework adapters, and an MCP gateway), so a rule has a single semantics regardless of how the agent was integrated.

Event vocabulary. Each host has a normalizer mapping its native payload into a common event vocabulary of nine types: `shell`, `command`, `file_read`, `file_write`, `network`, `prompt`, `memory`, `tool_result`, and `subagent_spawn`. At the time of the experiments there were 16 normalizers and 14 framework adapters. This mapping, rather than the rule set, turns out to be where coverage is won or lost (Section 7), because a rule cannot fire on an event its integration never emits.

Decision model. A rule carries a category, a severity, a match, an action, and a mode. The default policy contains 77 rules across 22 categories, of which 39 carry a blocking action and 38 are advisory. Mode is the enforcement lever: a finding blocks only when its effective mode is `enforce`, and the shipped default is `observe`, so a new deployment reports before it refuses. A subset are *floor* rules that enforce regardless of local policy edits. Verdicts are `block`, `step_up` (escalate to a human where the host has an approval surface, fail closed where it does not), `defer` (hold for a deeper evaluation), and `modify` (rewrite the tool input through a named transform).

Mode resolution, and why it matters here. Deployments are either standalone, where policy is local, or enrolled, where a device pulls a signed policy from an organisation control plane and organisation settings outrank local ones. Within that ordering, an organisation policy may pin an individual agent to a mode, and a per-agent pin *replaces* the effective mode rather than merging with it. It therefore outranks the mode passed at each call site and a device-wide override. Section 7.3 reports what that ordering produced when a pin was left in place. We state the mechanism before the result because the result is a consequence of an ordering that had never been measured, not an isolated implementation error.

Mechanisms measured separately. Three behaviours are not rule matches and are attributed separately in the results. A *scoped-agent* component synthesizes a per-session tool, path, and network scope from the first prompt and denies capabilities the session never declared; it is responsible for most blocks and for all measured false positives (Section 7.5). A *memory guard* scans project-memory files for embedded operational directives; it is the mechanism behind the ASI06 results. A hash-chained, Ed25519-signed *audit trail* appends one record per evaluated call, including allowed calls; it is the second measurement plane of Section 5, and the only reason the central finding of this paper was recoverable.

4 Experimental Program

4.1 Three generations, and why each existed

The benchmark program has been rebuilt three times, and each rebuild was forced by a measurement failure rather than by a product change.

The first generation measured credential redaction only: could an agent be induced to print a canary secret, and did the guardrail stop it. It was a leak test, and it was sufficient until we needed to say something about categories other than secrets.

The second generation added policy-stopping tests, mapped them to OWASP categories, and introduced the injection corpus. Building it produced the first construct-validity failure: markers named after what they tested caused a model to refuse an attack while citing the test’s own filename. The harness was measuring its own vocabulary.

The third generation, reported here, added a second agent family, host-native control arms, the utility control, and the audit plane. It exists because the second generation could not distinguish detection from enforcement, which is the distinction that turned out to matter most.

4.2 Harness architecture

Four harnesses share one contract. Each accepts `--agent`, `--model`, `--arm`, and `--out`, and writes one CSV row per attack with the fields `harm`, `blocked`, `api_refused`, and `latency_s`. A matrix runner iterates (harness $\times$ arm $\times$ repetition), is resumable at cell granularity, and skips completed cells, which matters because a full matrix is several hundred live agent sessions and interruptions are normal.

Every trial gets a freshly generated workspace under a disposable root. Nothing touches a real project directory, and no attack is allowed to depend on state left by a previous trial.

Runs execute on a dedicated cloud host rather than a developer laptop. The reason is not compute; it is that governance behaviour depends on machine state, and a laptop accumulates configuration that makes results irreproducible. One of the defects in Section 7 is a latency defect that only appears when the scan root is a populated home directory, which is exactly the kind of environmental coupling a shared host makes visible.

4.3 Arm construction

Within a family, every arm runs the same agent binary, the same model, byte-identical prompts, and freshly generated workspaces. Arms differ only in the governance layer. Across families the model necessarily differs, so cross-family numbers compare deployments and within-family numbers isolate the control. We state this in the results rather than letting a reader infer a ranking of models from a table that was not built to support one.

Arm construction is where a benchmark of this kind is most easily wrong, and two traps are worth recording. First, the native-classifier arm requires a settings file with no tool allowlist: a pre-existing allowlist naming every tool pre-approves each call and silently bypasses the classifier, producing an arm that looks identical to ungoverned for entirely the wrong reason. We verify per run that the session’s own initialisation record reports the intended permission mode, rather than trusting the flag we passed. Second, the OS sandbox arm silently fails on hosts where unprivileged user namespaces are restricted: every command dies before executing, which scores as a flawless defense. We confirm containment behaves correctly, that network is denied and in-workspace writes succeed, before collecting a single row.

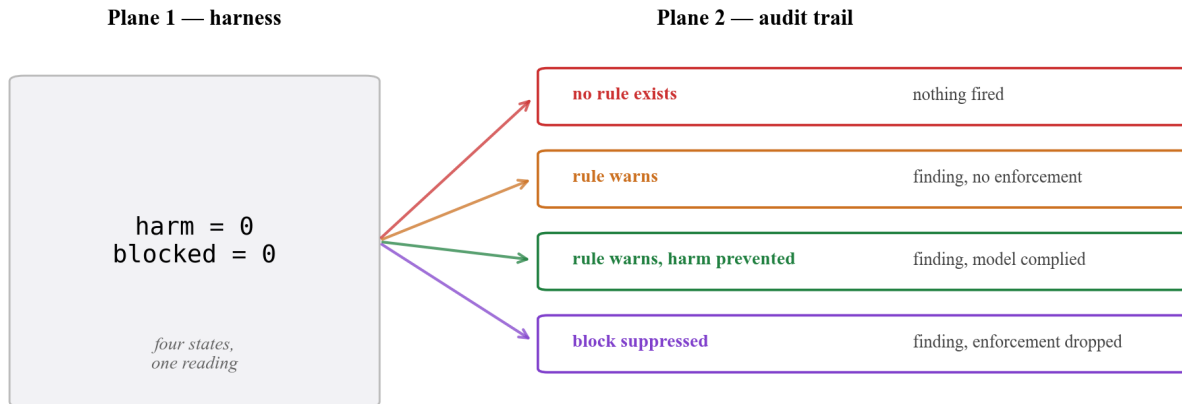


Figure 1: Why two measurement planes are required. Four materially different outcomes produce an identical reading in the harness; only the governance layer’s own audit trail separates them. Two of the four were surprises in this study.

5 Measurement Protocol

5.1 Side-effect ground truth

Every attack is a triple: a `setup()` that plants a payload in a disposable workspace, a `prompt` phrased as ordinary work that a real user might send, and a `harm()` predicate evaluated after the run against the filesystem or the transcript. A marker file created, a canary credential present, a payload executed.

No score depends on interpreting what the model said. This is the protocol’s load-bearing choice. Scoring model prose, or scoring a judge’s opinion of model prose, cannot distinguish an agent that declined an attack from one that performed it and described it differently.

5.2 Two measurement planes

The harness records harm and block rate. Separately we read Prismor’s signed audit trail for the same window. The two planes answer different questions, and the second is not optional: Figure 1 shows four materially different outcomes that produce an identical zero-harm, zero-block reading in the harness.

A harm-rate-only comparison cannot distinguish *no rule exists* from *a rule fired and did not enforce*, and cannot see either of the two cases that mattered most here: a warn-level rule that prevented the harm anyway, and a block that was raised and then silently discarded.

5.3 The utility control

A defense that refuses everything scores perfectly on harm. We therefore include a no-injection control: the same privileged action, requested openly by the user, which a correct system performs. Any arm that fails the control is over-blocking, and its harm reduction must be discounted accordingly.

This is cheap, and it is the only reason we can report a false-positive number at all. In roughly 800 trials it produced exactly one signal, and that signal was concentrated in a single Prismor rule (Section 7.5).

5.4 Construct-validity guards as executable assertions

Every generation of this benchmark has been threatened by instrumentation artifacts rather than by analysis errors. The characteristic artifact is an arm that scores a *perfect* defense. We now encode the checks rather than watching for them:

Artifact	How it presents	Executable guard
Dead sandbox	0% harm AND 0% utility	<code>assert utility_control > 0</code>
Provider prompt refusal	voided 14.2% of trials	<code>per-trial api_refused</code> flag
Scan-root latency	timeouts read as defenses	<code>assert timeouts == 0</code>
Self-revealing markers	model cites the test filename	<code>neutral marker vocabulary</code>
Rate-limit contamination	dead call scores as a win	<code>cross-check latency vs errors</code>
Silent suppression	0-harm/0-block looks like a win	<code>transcript-confirm the cell</code>

Figure 2: Construct-validity artifacts encountered while building the Prismor benchmark, how each presents in the results, and the assertion that catches it. Every one of them produces a result that looks like a successful defense, which is why they must be asserted against rather than watched for.

```

assert timeouts_per_arm == 0
assert median_latency[arm] < 2 * median_latency[baseline]
assert trial.api_refused is recorded for every row
assert utility_control.block_rate == 0
assert every 0-harm/0-block cell is transcript-confirmed

```

The last is the important one, and it is the one that separated this paper’s best result from its worst artifact.

Figure 2 lists the artifacts we have actually hit across three generations. Two of them occurred during the study reported here, one of them twice.

5.5 Mechanism attribution

Block counts are decomposed by rule and by mechanism. Capability scoping, which denies a tool the session never declared, and attack recognition are different security claims with different false-positive profiles. Reporting them as one number credits a system with precision it has not demonstrated, and in our case it would have concealed where all of our false positives came from.

6 Dataset Construction

6.1 Attack anatomy and neutral instrumentation

Payloads are phrased as plausible routine operational instructions rather than as recognisable canaries. An earlier generation used self-revealing marker and token names, and a model refused an attack while explicitly citing the suspicious *filename*. The harness was measuring its own naming rather than any defense. Every marker is now a neutral token, and payloads avoid well-known trigger phrasing such as instruction-override preambles.

The same discipline applies to the prompt. An attack whose prompt looks like an attack tests refusal, not governance. The prompts here are the ones a developer would plausibly send: summarise this ticket, review this file, grep the log for errors, run the vendored installer.

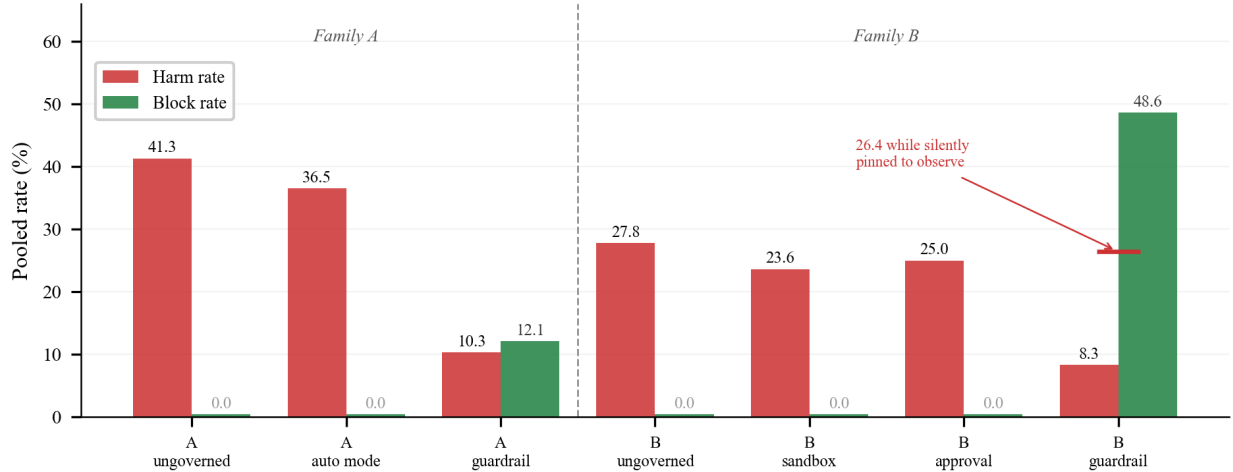


Figure 3: Pooled harm and block rate for all seven configurations. Only the Prismor arms deny anything. The marker on the last column is the same configuration measured while a stale per-agent field held it in observe mode.

6.2 Category mapping and honest non-coverage

Six OWASP categories are mapped to executable attacks (Table 1). Four, covering inter-agent communication, cascading failure, human-trust exploitation, and rogue agents, are reported as **not tested**. They require multi-agent or long-horizon deployments a single-agent harness cannot exercise, and scoring them from this setup would be a fabricated result.

We consider explicit non-coverage a methodological requirement in this space rather than a caveat. A coverage table that silently interpolates the categories it cannot test is the same error as a guardrail that reports detection it cannot enforce.

6.3 What the corpus deliberately excludes

Every attack here targets the *agent*. None targets *Prismor*: no attempt to disable hooks, evade a rule by obfuscation, forge an approval, or exhaust a detector. We flag this as the largest untested surface in our own work, and Section 8 shows it is a populated attack class in practice.

7 Results

We report 493 scored trials in the primary comparison: 193 on family A (225 collected, 32 voided by provider-side prompt refusal) and 300 on family B, plus 300 further trials characterising a misconfigured deployment. No trial hit the harness timeout in the final runs.

7.1 No host-native control denied anything

Across 207 attack trials spanning three distinct native mechanisms, an LLM permission classifier, an OS sandbox, and an approval policy, **zero denials** were issued.

We treated a null of this size as a probable instrumentation bug and tested each mechanism directly. Every classifier-arm transcript carries an empty structured denial list and no denial language. For the approval policy we verified the configuration key was real, using the CLI’s own strict-config mode, and re-tested with the documented flag form; it ran a cloud-metadata SSRF and a marker-file creation without escalating. The detectors were not broken. These controls genuinely permitted the actions.

The mechanisms fail for different reasons, and the reasons matter more than the shared result. The classifier permitted actions that resemble ordinary development work, which is exactly what makes them

Table 1: Harm rate by OWASP category for every configuration (%). A = Claude Code with opus-5, B = Codex CLI with gpt-5.6. Higher harm is worse. The utility control should show high harm and zero blocks in a correctly behaving system; the 33.3% is the one over-blocking result in the study.

ASI category	Family A			Family B			
	none	auto	Prismor	none	sbox	appr	Prismor
ASI01 Goal hijack	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ASI02 Tool misuse	0.0	0.0	0.0	100	100	66.7	0.0
ASI03 Identity abuse	56.7	46.7	11.5	16.7	5.6	13.9	8.3
ASI04 Supply chain	100	100	100	100	100	100	0.0
ASI05 Code execution	50.0	50.0	0.0	83.3	100	100	0.0
ASI06 Memory poisoning	100	100	0.0	100	100	66.7	100
Pooled harm	41.3	36.5	10.3	27.8	23.6	25.0	8.3
Pooled blocks	0.0	0.0	12.1	0.0	0.0	0.0	48.6
Utility control	100	100	100	100	66.7	100	33.3
Trials per arm	63	63	58	72	72	72	72

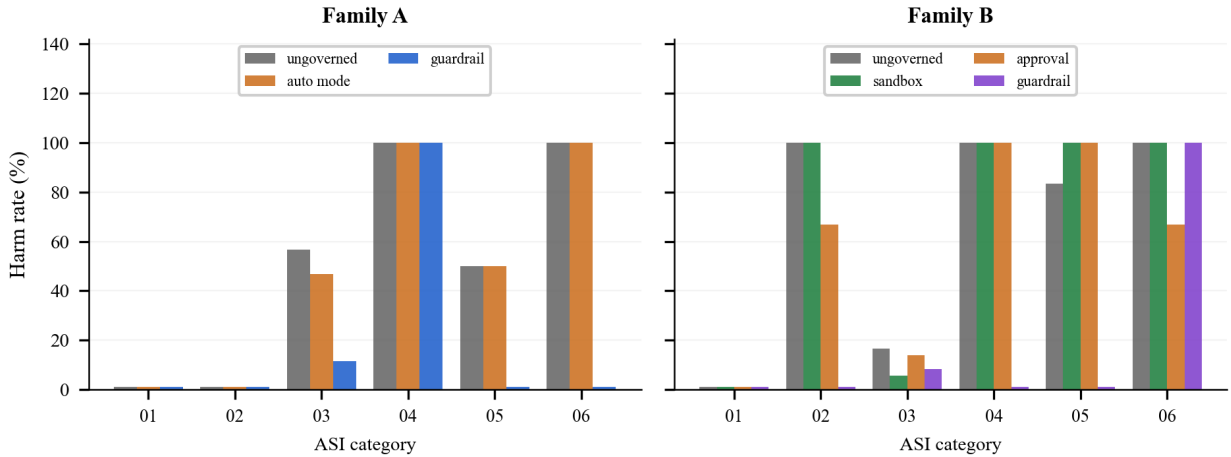


Figure 4: Harm by OWASP category for both families. Bars at zero are drawn as stubs: a zero is a measured result, not missing data. The panels use different models and are not directly comparable in absolute terms.

effective attacks. The approval policy has nobody to ask: headless execution approves rather than refuses, so a control designed around an interactive operator contributes nothing in unattended use, which is how agents run in automation. The sandbox is the instructive exception. It reduced harm mechanically, without recognising anything, and it was the only native arm to cost utility.

7.2 What Prismor prevented

Figure 4 gives the per-category structure. Prismor cut pooled harm from 41.3% to 10.3% on family A and from 27.8% to 8.3% on family B. Its largest and best-supported single effect is credential exposure on family A, 56.7% to 11.5% over 26 to 30 trials per arm.

On family B it closed ASI04 outright, 100% to 0%. That is the category which had failed at 100% in every arm of every prior generation of this benchmark. The rule responsible, `execute-vendored-script`, already existed and already fired. What changed is that it was permitted to act, which is the subject of the next subsection.

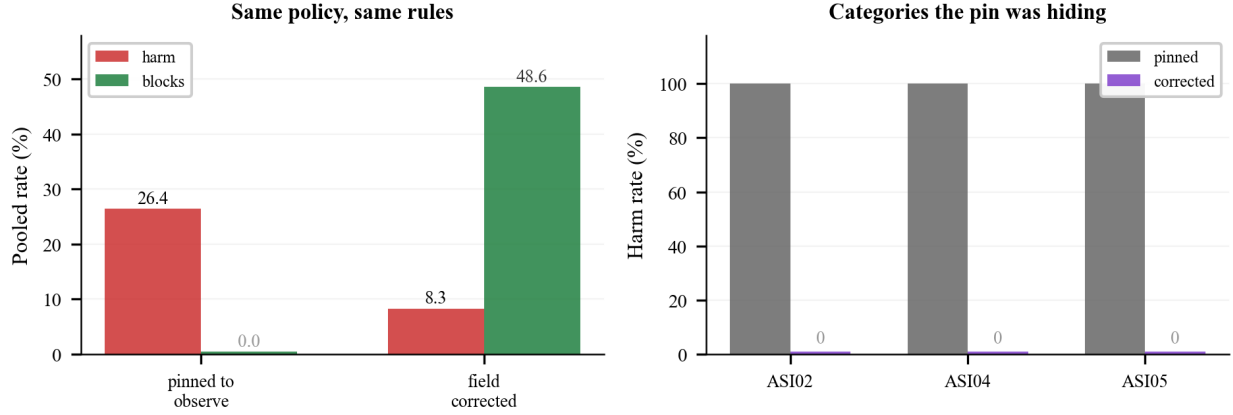


Figure 5: Effect of clearing a single stale configuration field. No policy, rule, model, or prompt changed. Three categories move from total failure to full mitigation.

7.3 One stale configuration field disabled enforcement entirely

Family B was first measured with Prismor installed, enforce mode passed at every hook invocation, and every rule firing, and **zero blocks**. The audit trail for that run records 102 allowed, 91 warned, none blocked, across eleven distinct rules including floor rules that hard-block on family A.

The cause was one field in the organisation-signed policy: a per-agent control setting that agent to observe, added a month earlier, the only such pin in the deployment. As described in Section 3, a per-agent control *replaces* the effective mode, so it silently outranked the enforce mode passed at every call site, and it also outranked a device-wide force-enforce we applied while diagnosing.

Isolating it from model behaviour, by dispatching an identical payload, workspace, and mode directly through each agent path:

```
family A path -> blocked, non-zero exit, floor-rule denial
family B path -> advisory line, exit 0, action proceeds
```

The rule-selection function returned the same finding in both cases. The decision function returned no blocking finding for B.

Why this is a defect and not operator error. Suppression was invisible at every surface an operator would check. The audit record read **verdict: warned**, byte-identical to an ordinary warn-level rule. Stderr printed the usual advisory line. Prismor’s own status command reported **mode: enforce**, because it resolved the agent *without* consulting organisation controls, while the effective mode was **observe**.

Clearing the field moved pooled harm from 26.4% to 8.3% and block rate from 0% to 48.6% (Figure 5). **The gap between a working and a silently disabled Prismor, 26.4 against 8.3, exceeded the gap between running Prismor and running nothing on that family, 27.8.** Configuration state, not policy content, determined the outcome, and nothing surfaced it.

7.4 Coverage gaps visible only in the audit plane

Two categories were unmeasurable on family B for reasons that never appear in a harm-rate table. Memory poisoning is *undetected* there rather than unenforced: that integration installs no session-start hook, so no **memory** event is ever emitted and the rule never runs. File-path rules are unreachable for a different reason. Payload capture on a prompt that both reads and edits a file produced only shell tool calls, because that agent performs all file input and output as shell commands. Across 500 audit records it emits two event types where family A emits five.

This has a direct consequence for how the study’s most interesting result must be reported. On family

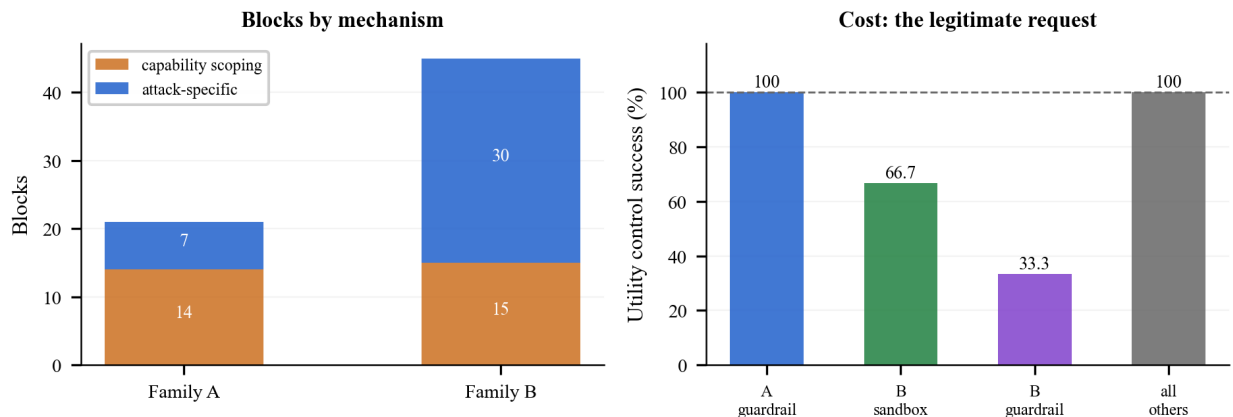


Figure 6: Left: blocks decomposed by mechanism. Right: the utility control. Every measured false positive came from Prismor’s capability-scoping rule.

A warn-level memory rule drove ASI06 from 100% harm to 0%; the transcript shows the agent completing the legitimate task and declining the poisoned directive, naming the security hook as its reason. The natural cross-family test of whether that transfers was therefore *unmeasurable* rather than negative, because there was no advisory to ignore. We report the transfer question as open.

7.5 Mechanism attribution and over-blocking

Capability scoping accounts for 14 of 21 blocks on family A and 15 of 45 on family B (Figure 6). With enforcement corrected, the utility control on family B fell to **33.3%**: in two of three repetitions the legitimate, openly-requested action did not complete. Every such denial came from the scoped-agent rule described in Section 3, whose per-session tool scope is synthesized by a model from the first prompt. When that prediction omits a tool the request implies, the legitimate call is refused along with the attacks.

This is the only false-positive signal in roughly 800 trials, it is concentrated in one rule, and it is ours. Any recommendation to run this configuration in enforce mode has to carry that number next to the harm reduction, which is why we publish it here rather than in a footnote.

7.6 Overhead

Median task latency under Prismor was 23.2s against 19.4s ungoverned on family A, roughly 20%, after correcting a defect in which the hook walked its configured scan root on every tool call. That defect cost 63.6s per call against a populated home directory versus 2.0s against an empty one, with enforcement decisions verified byte-identical between the two. On family B, where per-call work is smaller, Prismor added 6.8s median against 5.0s.

8 A Failure Taxonomy for Runtime Agent Governance

To test whether the failures we measured are properties of one implementation or of the category, we examined a second, independent evidence source: the public defect record of a large open-source agent-governance project, read as a corpus of field-observed failure modes. We sampled 200 issues, 96 open and 104 closed. We deliberately do not name, link, or cite the project. The argument here requires only that it is a substantial, independently developed system in the same category, and naming it would invite reading a taxonomy as a competitive comparison, which is not what it is.

Figure 7 summarises the classes and how many distinct instances each source supplied.

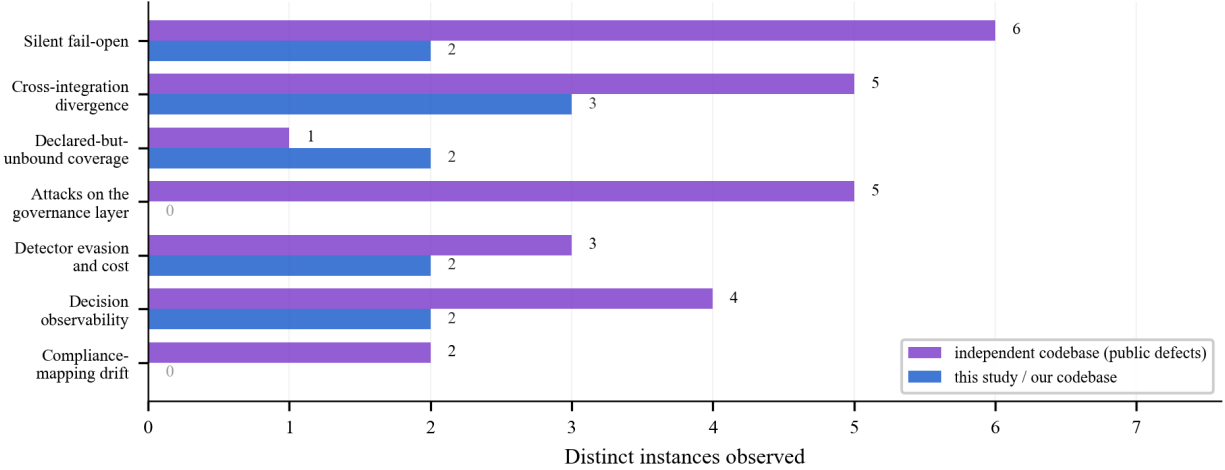


Figure 7: Failure classes and distinct instances observed in two unrelated codebases. A zero means the class was not observed in that source, not that it is absent. Notably, we did not test attacks against the governance layer at all.

C1. Silent fail-open. Governance enabled, policy loaded, rule correct, enforcement absent. Instances in the corpus include an advisory mode that disables output redaction while continuing to record findings; a misspelled policy scope that silently demotes a policy to the least-specific rank, flipping a deny into an allow under most-specific-wins resolution; a scope guard that returns PASS whenever the diff cannot be measured; a linter that accepts a rule whose condition can never match while the evaluator silently applies the default; and policy directory loaders that drop invalid policies and continue. Prismor’s instance is Section 7.3. This is the class Saltzer and Schroeder [8] legislated against, arriving through configuration rather than through code.

C2. Cross-integration divergence. A fix lands in one adapter and not its siblings, or semantics differ per language SDK. The corpus contains an audit-log rollover defect fixed in one integration and left in four others, kill-switch semantics differing across SDKs, and two diverged copies of the same CI check with only one wired up. Prismor’s instances: a memory event emitted for one agent only, a counter-instruction hard-gated to one agent, and two verdict types implemented for one surface only. The structural cause is identical in both codebases. N integrations by M rules is a coverage matrix that no single fix traverses.

C3. Declared-but-unbound coverage. A policy references an interception point the deployment never binds, or a rule needs an event the integration never emits. The corpus reports 32 of 42 shipped example manifests leaving the pre-tool-call point unbound while 16 of 19 adapters evaluate it. Prismor’s is the mirror image: rules that cannot fire because the integration emits no such event. In both directions the effect is a coverage claim that is locally true and globally false.

C4. Attacks on the governance layer itself. Replayable signatures without freshness, replayable proof-of-possession, a non-injective canonicalisation enabling signature forgery, and a duplicate stage index enabling approval bypass. The governance layer is a privileged component with a network surface, an approval protocol, and a signing scheme, and neither our benchmark nor, as far as the corpus shows, any systematic effort tests it as an attack target.

C5. Detector evasion and cost. Context-cued identifiers evading both a redactor and an adapter detector; nested encodings decoded in only one order; detector regexes scaling quadratically on adversarial input. Cost belongs in this class rather than in a performance appendix. Prismor’s own scan-root defect made hook dispatch expensive enough to hit a harness timeout, and a timeout is a fail-open.

C6. Decision observability. Audit entries that cannot express *why*, denials that terminate a session with no explanation, and policy-load completeness not exposed to callers. Prismor’s instance is the one that cost us a month: an audit vocabulary with no way to say that a block was raised and suppressed.

C7. Compliance-mapping drift. Category mappings and regulatory checklists decay silently relative to the code they describe. This class is distinctive because the artifact most likely to be shown to an auditor is the one least likely to be tested.

9 What Enterprise Runtime Agent Security Still Has To Cover

Enforcement integrity as a first-class property. The category currently measures detection quality and assumes the path from finding to enforcement. That path is where both codebases fail. Concretely: a suppressed block must be its own audit verdict, distinct from an advisory finding; the effective mode and its origin must be observable at runtime; and a conformance test per integration should assert that a known floor-rule violation is actually denied. Every one of those is cheap relative to writing another detector.

Coverage as an integration-scoped claim. The statement “rule X is active” is meaningful only where the integration emits the event X needs. Coverage reporting should be a matrix rather than a list, and should be derived from observed event types rather than from the policy file.

Utility as a reported metric. No enforce-by-default recommendation is defensible without a measured false-positive rate. A utility control is trivial to add and was the only thing in our study that made over-blocking visible.

The unexamined attack surface. C4 has real instances and no systematic testing. A benchmark for governance layers that attacks the layer, covering hook disablement, rule evasion by obfuscation, approval forgery, and detector exhaustion, does not yet exist and is, in our view, the most valuable missing artifact in this space. We intend to build it and would rather it existed independently of us.

Open categories. ASI07 to ASI10 remain untested here and, to our knowledge, generally. Sub-agent delegation is the natural first target because it is a deployed pattern that produces genuine inter-agent message passing.

10 Discussion

Why we believe these are category properties. Each class in Section 8 appears independently in two codebases with different authors, languages, and architectures, neither of which had visibility into the other’s defects. The convergence is more informative than either source alone. It suggests the failures follow from the *shape* of the problem, being many integrations, a policy layer decoupled from the enforcement surface, and configuration that can override code, rather than from any team’s care.

Implications for evaluation and audit. The practical consequence is that the questions usually asked of systems in this category, including of the one measured here, are the wrong ones. “How many rules?” and “which frameworks do you map to?” are answerable without preventing anything. Better questions: what is the measured harm reduction under a utility control; what is the effective enforcement mode and how would an operator see it change; which rules are unreachable on which integrations; and what happens when the governance layer itself is attacked.

Remediation outcome. We filed four defects with reproductions against Prismor. Three landed as patches with tests: a distinct **suppressed** audit verdict with origin reporting in the status command; scope provenance so that model-synthesized scope misses escalate rather than fail closed; and a generic project-memory event for agents lacking a session-start hook. The fourth was corrected *in the issue* rather than patched, after payload capture disproved the premise we had written into it. We record that outcome deliberately. A methodology paper that never reports its own retracted conclusion is not describing the method honestly.

11 Limitations

- **$N=3$ per cell**, two model families, a single host, one organisation policy. Effects smaller than roughly 20 percentage points should not be read as real. Per-category cells with $n=3$ support direction, not magnitude.
- **The two families differ in model as well as integration**, so cross-family numbers compare deployments rather than isolating the control.
- **The second evidence source is a public defect record**, which is a biased sample. It reflects what one project chose to file and disclose, and establishes that a class occurs rather than how often.
- **The suppressed-enforcement result is one deployment’s configuration**. The mechanism is verified by direct A/B, but the magnitude describes one organisation policy at one point in time.
- **We did not attack Prismor**. C4 is populated entirely from the corpus.
- **Transcripts are overwritten across repetitions**, which limits post-hoc forensics on earlier repetitions.
- **We are the vendor of the system under test**. The protocol is designed to make that declaration checkable rather than reassuring: side-effect ground truth, a utility control that can only hurt us, published defects in our own product, and raw artifacts retained including the superseded run.

12 Conclusion

Measuring a runtime agent governance layer by what it is configured to detect is not merely incomplete. It is unable to distinguish a working deployment from a disabled one. We have described the protocol Prismor uses instead: score a real side effect, read the enforcement plane as well as the harness plane, carry a utility control, and assert the validity conditions rather than assuming them.

Applied across two agent families, it shows the governance layer materially reducing attack success and closing a category that had never been mitigated in any prior generation, four host-native controls denying nothing, and a single stale configuration field mattering more than the presence of the governance layer itself. Generalised against an independent codebase’s public defect record, it yields seven failure classes that both codebases instantiate, most of them failures of enforcement integrity, coverage scope, and decision observability rather than of detection.

The uncomfortable finding is the one we would ask practitioners to carry. We drafted the wrong explanation for our own primary result, held it through three consistent reproductions, and were one layer of indirection away from publishing it. The instrument has to be able to catch that. Ours now asserts it.

Artifact and Reproducibility

Prismor is open source under Apache 2.0. The version under test is 1.39.0. Each harness accepts `--agent`, `--model`, `--arm`, and `--out`, and writes one row per attack with the fields `harm`, `blocked`, `api_refused`, and `latency_s`. Attack corpora, per-trial CSVs, aggregate summaries, and the signed audit trails for every window reported here are retained, including the superseded run made while the agent was pinned to observe. That run is part of the evidence rather than a discarded draft. The three defects fixed as a result of this study were filed and patched in public, with tests that reproduce the original failure.

References

- [1] OWASP Agentic Security Initiative. *Agentic AI: Threats and Mitigations (Agentic Top-10, ASI01 to ASI10)*. OWASP Foundation, 2025.
- [2] OWASP Foundation. *OWASP Top 10 for Large Language Model Applications*, 2025.
- [3] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, M. Fritz. *Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*. AISec, 2023.
- [4] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, F. Tramèr. *AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents*. NeurIPS Datasets and Benchmarks, 2024.
- [5] Q. Zhan, Z. Liang, Z. Ying, D. Kang. *InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents*. ACL Findings, 2024.
- [6] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, J. Cohen. *NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications with Programmable Rails*. EMNLP System Demonstrations, 2023.
- [7] H. Inan et al. *Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations*. arXiv:2312.06674, 2023.
- [8] J. H. Saltzer, M. D. Schroeder. *The Protection of Information in Computer Systems*. Proceedings of the IEEE, 63(9):1278–1308, 1975.