From Incident to Failing Build: Incident-Derived Regression Testing for AI-Agent Guardrails

Ajin Baby

cloudandsre.com

2026-09-03 · v0.1 · cloudandsre.com/research/incident-derived-regression-testing/

Abstract

The 2024–2026 period produced an abundance of AI-agent failure taxonomies and a steady supply of agent-security benchmarks. It also produced a steady supply of *actual production failures*: a customer-service agent inventing a refund policy its operator was held to, a coding agent dropping a production database during an explicit code freeze, an agent loop billing tens of thousands of dollars before anyone noticed. These two bodies of work barely touch. The benchmarks are large, sophisticated, and **authored** — scenarios written to exercise a threat model. The incidents are real, attributable, and inert: they live in press coverage and incident databases, where they inform nobody's build. We connect them. We contribute (1) a **corpus** of ten publicly documented agent production failures spanning six failure classes, each normalised to a replayable scenario with a decision-time plane separated from adjudication, and each carrying its cited public sources; (2) a **reference guard chain** of five deliberately minimal controls, offered as an honest baseline rather than a product; (3) a **CI gate** that replays the corpus against a guard configuration and fails the build on any escape; and (4) a **measured ablation study** (E1–E4, deterministic and exactly reproducible). The study finds: an unguarded configuration reproduces all ten incidents and the reference chain contains all ten; four of the five guards are each the *sole* container of exactly one failure class, a 1:1 necessity partition with no redundancy to trade away; of the 31 non-empty guard subsets exactly two are sufficient and exactly one is minimal, at size four, with the fifth guard load-bearing in no minimal chain — an honest null we report rather than bury; and, most consequentially for practice, **configuration drift is indistinguishable from guard deletion**: widening one trust list to admit tool output readmits precisely the two incidents that deleting that guard readmits, while a budget guard built from two ceilings in disjunction masks drift in either one for as long as the other still binds, so a CI assertion on one parameter reports green across a real loss of containment. The corpus is small — ten incidents against benchmark suites two orders of magnitude larger — and we argue the axis it competes on is provenance and consequence, not scale.

Keywords: AI agents, guardrails, regression testing, incident analysis, agent safety, continuous integration, prompt injection, site reliability engineering.

---

1. Introduction

There is no shortage of writing about how AI agents fail. Failure taxonomies arrived in volume across 2025 and 2026, as did benchmarks: authored scenario suites that measure whether a guardrail system detects an attack, contains a compromise, or refuses a harmful tool call. This work is genuinely good and this paper depends on it.

What none of it does is fail your build.

The gap is not analytical, it is *mechanical*. A taxonomy tells you that destructive actions without approval gates are a failure class. A benchmark tells you that your guardrail system scores 74.8% on 580 authored scenarios. Neither tells you that the specific change you merged this afternoon has re-opened the exact hole that deleted a production database at a named company on a named date. Software engineering solved the general form of this problem long ago, and the solution has a name: a regression test. You take the failure that actually happened, you encode it, and you run it forever, so that the second occurrence is caught by a machine at merge time rather than by a customer at 3am.

Agent guardrails have no equivalent. The incidents that would constitute the regression suite are scattered across press coverage, vendor postmortems, and the AI Incident Database in prose form, and prose does not execute.

This paper builds the missing artifact and then measures it, because an artifact that has not been measured is a proposal. Our contributions:

1. **A corpus** (§4) of ten publicly documented agent production failures, 2024–2026, across six failure classes, normalised to a schema that separates what the agent could see at decision time from the post-hoc adjudication used to score it, with every incident carrying its public sources.
2. **A reference guard chain** (§5) of five minimal controls, and a **CI gate** that replays the corpus against any guard configuration.
3. **A measured study** (§6): containment, guard necessity, minimal sufficient chains, and the price of configuration drift.
4. **A negative result we did not go looking for** (§6.2, §6.3): one of our own five reference guards is load-bearing in no minimal chain, and we report it as such.

2. Background

Why incidents and benchmarks are different evidence. An authored scenario encodes what its author believed the threat model to be. That is its strength — coverage can be designed, scaled, and balanced — and also its limitation: it cannot surprise you with a failure mode nobody thought to write down. A production incident carries the opposite properties. There are few of them, they are unbalanced, their reporting is incomplete and sometimes self-serving, and they arrive with no design at all. What they have is the one property authored scenarios cannot manufacture: **they happened**. A control that does not contain a documented incident has a demonstrated gap, not a hypothetical one, and that difference is what makes a regression suite persuasive to the people who approve the work to fix it.

Why a gate, not a score. A benchmark reports a number, and numbers invite negotiation — 74.8% is discussed, contextualised, and eventually accepted. A gate reports pass or fail against a specific named incident, and a failing gate that says *"this configuration would not have contained the Replit database deletion"* is a different conversation from *"containment dropped 3 points."* We built the gate rather than the leaderboard deliberately.

3. Related Work

Four lines of work are adjacent, and each covers ground this paper does not.

Authored agent-security benchmarks. AgentDojo [5] evaluates prompt-injection attacks and defences across 97 realistic tasks and 629 security test cases. GuardianAgentBench [1] spans 580 scenarios in six domains with five adversarial attack modes, reporting that the strongest configuration reaches 74.8% overall accuracy. ContainmentBench [2] evaluates post-exposure containment over a pre-specified 17,640-rollout study; its own abstract states that the evaluation uses synthetic workflows. LlamaFirewall [3] ships a production guardrail system with a 600-scenario

benchmark across seven injection techniques. All four exceed this corpus in scale by two to three orders of magnitude, and all four author their scenarios.

Runtime enforcement. AgentArmor [6] treats agent runtime traces as analysable programs with a type system enforcing data-flow policy. This is a considerably more sophisticated control than anything in our reference chain, which is the point: our guards exist to make the corpus's verdicts interpretable, not to compete on enforcement.

Incident catalogues. The nearest prior work is Token Budgets [4], which catalogues 63 production budget-overrun incidents drawn from 21 orchestration frameworks, each backed by a quoted GitHub issue, organised into an eight-cluster taxonomy, and paired with an affine-typed Rust mitigation validated on live APIs. It is the same instinct — real incidents, not authored ones — executed at six times our scale within its scope. That scope is the distinction: Token Budgets is confined to cost and budget overruns, and its mitigation is a type system rather than a replay gate. It does not, and does not aim to, cover destructive actions, injection, ungrounded output, or state hallucination.

The crosswalk.

Work	Scenario provenance	Scale	Failure classes	Ships a build gate
AgentDojo [5]	authored	97 tasks / 629 cases	injection	no
GuardianAgentBench [1]	authored	580 scenarios	6 domains, 5 attack modes	no
ContainmentBench [2]	synthetic workflows	17,640 rollouts	post-exposure containment	no
LlamaFirewall [3]	authored	600 scenarios	7 injection techniques	guards, not a corpus gate
Token Budgets [4]	real, cited issues	63 incidents	cost only	no (typed mitigation)
This work	real, cited public incidents	10 incidents	6 classes	yes

The gap. On scale and on enforcement sophistication the authored benchmarks lead comfortably, and nothing here should be read as displacing them. The unoccupied cell is the one this paper fills: an artifact whose units are *documented production failures across categories*, whose output is a build verdict rather than a score, and whose regression semantics are defined against configuration rather than against a model. We make no claim to being first to catalogue real incidents — [4] did that within its scope — and no claim that ten incidents constitute adequate coverage of anything.

4. The Corpus

4.1 Unit of the corpus. Each incident is a JSON record carrying the narrative fields a reader needs (system, date, failure class, severity, blast radius, time to detect and contain, containment as reported) and a **scenario**: an ordered list of tool calls, each with a tool name, a resource, a provenance label, and flags for whether the call is destructive, grounded, verified, or approved. The scenario is the executable part. The narrative is what makes the verdict legible to a human reviewing a failed build.

4.2 Two planes. The scenario separates the **decision-time plane** — what the agent could see when it acted — from the **adjudication plane** — the annotation of which step constituted harm, and which control class ought to have contained it. Guards see only the former. This is enforced structurally by the record format rather than by author discipline, because hindsight contamination in a replay corpus is not a risk that discipline reliably manages.

4.3 Contents. Ten incidents, 2024–2026, all involving systems operating autonomously at the time of failure.

Date	System	Failure class	Severity	Sources
2024-02-14	Air Canada customer-service chatbot	ungrounded_output	medium	3
2025-04-17	Anysphere / Cursor support bot	ungrounded_output	medium	2
2025-05-26	GitHub MCP server, untrusted issue body	prompt_injection	high	2
2025-07-17	Amazon Q Developer extension	supply_chain	low	2
2025-07-21	Replit AI Agent	destructive_action	critical	3
2025-07-25	Google Gemini CLI	state_hallucination	high	2
2025-11-15	Four-agent LangChain pipeline	runaway_loop	high	1
2026-04-10	Single autonomous agent, 63-hour run	runaway_loop	medium	1
2026-04-25	Cursor coding agent, PocketOS	destructive_action	critical	2
2026-07-15	GitHub Agentic Workflows	prompt_injection	high	1

Six failure classes: ungrounded output (2), prompt injection (2), destructive action (2), runaway loop (2), supply chain (1), state hallucination (1). Nineteen public sources across ten incidents, comprising press coverage, vendor statements, and AI Incident Database entries [7].

4.4 What normalisation costs. Turning a press report into a two-step scenario is lossy and interpretive. The record for the Replit deletion, for example, compresses a multi-day session into `query_database` followed by an unapproved `drop_database`. That compression is a *modelling choice made by us*, and a different modeller would produce a different scenario with different containment properties. We treat this as the corpus's principal threat to validity (§7) rather than as an implementation detail, and it is the reason every record carries its sources: a reader who disagrees with our reduction can check it against the reporting and open an issue against the record.

5. Guards and the Gate

5.1 The reference chain. Five controls, each the minimum honest implementation of a class rather than a competitive one. `ProvenanceFilter` refuses calls originating from untrusted provenance — content that arrived through a tool result may not choose the next action. `GroundingGate` refuses a statement relayed to a user that is not traceable to a retrieved source. `VerificationGate` refuses a destructive call that rests on an unverified belief. `ApprovalGate` refuses an irreversible operation lacking an explicit per-call approval. `BudgetCap` refuses a step that would breach a cost ceiling or a step ceiling, checked *before* admission rather than observed after.

A sixth control, `ScopeLimiter`, exists in the library but is deliberately excluded from the reference chain: a useful allowlist is workload-specific, and shipping a permissive default would let scope-escalation scenarios pass for the wrong reason.

5.2 The gate. `run_gate` replays every scenario through a chain, first block wins, and reports per-incident containment plus any escapes with the incident's identity, class, and the control class that was expected to contain it. A configuration with any escape fails. The gate also reports **unexercised guards** — controls in the chain that never fired — because a guard the corpus never exercises is a guard the corpus gives you no evidence for.

5.3 Determinism. Every result in §6 is exact and reproducible without seeds or trial counts. The corpus is fixed, the scenarios are fixed, the guards are pure predicates over a call and a run state. This is not a shortcut around statistical rigour; it is a requirement of the artifact's purpose. A gate that returned a distribution would not be usable as a gate.

6. Measured Study

All results from `examples/paper_study.py` in the companion repository, pinned by 17 tests in `tests/test_study.py`.

6.1 E1 — containment. The same ten incidents, the same replay, and the only variable is what was permitted to say no.

Table 1 — containment by chain.

chain	contained	escaped	rate
unguarded	0	10	0%
reference	10	0	100%

The unguarded baseline reproduces every incident, which is the corpus's basic sanity condition: a scenario that harms nothing even without controls is not encoding its incident. No guard in the reference chain is unexercised.

6.2 E2 — which guards are load-bearing. "Do we have guardrails" is unanswerable. "Which historical failure modes does removing this control readmit" is a number.

Table 2 — leave-one-out ablation.

guard removed	contained	readmitted	failure class readmitted
(none)	10	0	—
provenance_filter	8	2	prompt_injection
grounding_gate	8	2	ungrounded_output
verification_gate	10	0	—
approval_gate	8	2	destructive_action
budget_cap	8	2	runaway_loop

Four guards, four failure classes, two incidents each, no overlap: a **1:1 necessity partition**. There is no redundancy to trade away among these four, and no ordering effect — each is the sole container of its class. The practical reading is that a team dropping any one of these four controls is not weakening defence in depth, it is deleting the only thing standing between it and a documented failure mode.

`verification_gate` is the exception, and the per-incident matrix says why.

Table 3 — per-incident containment by individual guard.

incident	class	prov	grnd	verf	appr	bdgt
air-canada-chatbot	ungrounded_output		✓			
cursor-support-bot	ungrounded_output		✓			
github-mcp-prompt-injection	prompt_injection	✓				
amazon-q-wiper-injection	supply_chain	✓			✓	
replit-database-deletion	destructive_action				✓	
gemini-cli-file-deletion	state_hallucination			✓	✓	
multi-agent-ping-pong	runaway_loop					✓
agent-63-hour-runaway	runaway_loop					✓
pocketos-deletion	destructive_action				✓	
github-agentic-workflows	prompt_injection	✓				

Two incidents are doubly covered. The Amazon Q supply-chain injection is contained by provenance filtering *or* by approval. The Gemini CLI state hallucination is contained by verification *or* by approval — and since `verification_gate` contains no incident that `approval_gate` does not also contain, removing it changes nothing. That is real defence in depth and we would not remove it from a production chain. It is also, precisely, an absence of evidence: **this corpus cannot justify `verification_gate` independently**, and an incident that only verification can contain is a wanted contribution to the corpus.

6.3 E3 — minimal sufficient chains. Exhaustive over all 31 non-empty subsets of the five reference guards. A subset is *sufficient* if it contains all ten incidents and *minimal* if no proper subset of it is also sufficient.

Of 31 subsets, **exactly two are sufficient** — the full chain and the four-guard set — and **exactly one is minimal**: `{provenance_filter, grounding_gate, approval_gate, budget_cap}`. `verification_gate` appears in no minimal chain.

The sharper reading is the one about the guard space rather than about our particular guards: the necessity structure is almost entirely rigid. There is exactly one way to be minimally sufficient against this corpus, which means a team cannot substitute controls or economise. Whether that rigidity is a property of agent failure modes or an artifact of a ten-incident corpus is exactly the question a larger corpus would settle.

6.4 E4 — the price of configuration drift. Guard *absence* is the easy case, and CI catches it already. The failure that ships is a control still in the chain, still reporting green, and no longer load-bearing, because someone widened a parameter to unblock a workflow. Only two of the five reference guards carry parameters that can drift; the other three are binary predicates with nothing to widen. We sweep both in full factorial: the provenance filter's trust set (tight, or loosened to admit `tool_output` — the exact edit an engineer makes when the filter blocks a legitimate workflow), the cost ceiling, and the step ceiling.

Table 4 — full factorial over drifted parameters.

provenance	cost cap	step cap	contained	readmitted
tight	$10	50	10	0
tight	$10	500	10	0
tight	$100	50	9	1
tight	$100	500	9	1
tight	$1000	50	9	1
tight	$1000	500	8	2
loose	$10	50	8	2
loose	$10	500	8	2
loose	$100	50	7	3
loose	$100	500	7	3
loose	$1000	50	7	3
loose	$1000	500	6	4

Three results.

Drift is deletion. Adding one provenance label to a trust set readmits both prompt-injection incidents — *the identical loss* that deleting `provenance_filter` produces in Table 2. The control is still present, still in the chain, still fires on other traffic, and is worth exactly nothing against the failure class it exists for. Any CI check that asserts guards are *configured* rather than *effective* reports green across this change.

A disjunctive guard masks its own drift. `BudgetCap` blocks when the step ceiling is reached *or* the cost ceiling would be breached. Because the ceilings are in disjunction, either one binding is enough, so drift in one is invisible while the other still binds: raising the step ceiling tenfold from 50 to 500 changes nothing at all while the cost cap holds at $10, and only becomes load-bearing once cost has *also* drifted to $1000, at which point the multi-agent runaway returns. A team that asserts `max_steps` in CI and lets cost drift — or the reverse — has a test that cannot fail for the reason it was written.

Drift composes additively across guards, not multiplicatively. Loosening both readmits the union of what each readmits alone (2 + 2 = 4), with no interaction term. The two controls guard genuinely disjoint failure classes, which is the E2 partition showing up again.

7. Discussion and Threats to Validity

Ten incidents is the headline limitation. The authored benchmarks in §3 carry hundreds to tens of thousands of scenarios. This corpus carries ten, distributed unevenly across six classes, with two classes represented by a single incident each. No claim in §6 should be read as a statement about agent failures in general. E2's clean partition and E3's unique minimal chain are properties *of this corpus against these guards*, and the most likely effect of growing the corpus is that both become messier — overlapping coverage, multiple minimal chains, and `verification_gate` acquiring an incident only it contains. That is the intended trajectory, not a refutation.

Public reporting is a biased sample. Incidents enter this corpus because they were newsworthy, litigated, or publicly confessed. Failures that were quietly contained, or that occurred at organisations with no disclosure obligation, are absent

by construction. The corpus therefore over-represents the spectacular and under-represents the routine, and its severity distribution (two critical, four high) should be read as a property of what gets reported rather than of what happens.

Scenario reduction is interpretive. As noted in §4.4, each scenario is our reading of a public account. We mitigate this with cited sources per record and a schema that makes the reduction inspectable, but a scenario that mis-models an incident will produce confident, wrong verdicts. This is the threat we would most want an external reviewer to attack.

The guards are ours. E2 and E3 measure necessity among *our five reference controls*. A team running AgentArmor-style program analysis [6] or LlamaFirewall [3] has a different and generally stronger control surface, and the necessity structure over that surface would differ. The gate is designed to be pointed at a real configuration; the reference chain is a baseline for interpreting it, not a recommendation.

Determinism cuts both ways. §5.3 argues determinism is required of a gate. It also means this study says nothing about model-dependent variance — whether a given agent, on a given day, would even attempt the harmful step. The corpus measures whether the *control* would have stopped it, which is a question about your configuration and deliberately not about your model. Non-determinism in agent behaviour is real and is exactly what the authored benchmarks with N-rollout designs [2] measure well.

No live-agent evaluation. The replay is offline and control-focused by design; we do not drive a live model through these scenarios and make no claim about end-to-end agent behaviour.

8. Conclusion and Future Work

Agent failure taxonomies are papers. Benchmarks are scores. Neither is a build that fails when the control that would have prevented a named, dated, public incident stops working. This paper contributes the missing mechanical link — a corpus of real incidents, normalised and executable, behind a gate — and then measures what it can actually tell you. The measurements are more interesting than the artifact: guard necessity over this corpus is a rigid 1:1 partition with exactly one minimal sufficient chain, one of our own reference controls cannot be justified by the corpus at all, and configuration drift is worth precisely as much containment as guard deletion while remaining invisible to the CI checks teams actually write.

The last of those is the finding we would ask practitioners to act on: assert that your guards are *effective against named incidents*, not that they are *present in your configuration*, because the two diverge exactly where it costs you.

Future work is mostly corpus growth, and it is the kind of growth that benefits from other people: incidents that discriminate between controls our present ten cannot separate, classes we have one example of, and — most valuable — an incident that `verification_gate` alone contains. Beyond that: severity-weighted gating, so that a build can fail differently for a readmitted critical incident than for a low one; and a study of whether the necessity partition survives contact with a production control surface rather than a reference one.

References

(Author lists, titles and identifiers resolved against arXiv metadata and live URLs, 2026-09-03.)

[1] V. I. Naik, C. Xu, D. Dong, H. Hassan, A. Pradhan, O. Mendelevitch, T. Shafat, and H. Irshad. GuardianAgentBench: Where Agents Fail and How to Guard Them. arXiv:2607.20982, 2026. [2] W. Lan, S. Li, M. Wu, X. Lai, J. Yang, and H. Shen. ContainmentBench: Trace-Based Evaluation of Post-Exposure Containment in Tool-Using LLM Agents. arXiv:2607.23999, 2026. [3] S. Chennabasappa, C. Nikolaidis, D. Song, D. Molnar, S. Ding, S. Wan, S. Whitman, L. Deason, N. Doucette, A. Montilla, A. Gampa, B. de Paola, D. Gabi, J. Crnkovich, J.-C. Testud, K. He, R. Chaturvedi, W.

Zhou, and J. Saxe. LlamaFirewall: An open source guardrail system for building secure AI agents. arXiv:2505.03574, 2025. [4] S. Khan. Token Budgets: An Empirical Catalog of 63 LLM-Agent Budget-Overrun Incidents, with an Affine-Typed Rust Mitigation as a Case Study. arXiv:2606.04056, 2026. [5] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. arXiv:2406.13352, 2024. [6] P. Wang, Y. Liu, Y. Lu, Y. Cai, H. Chen, Q. Yang, J. Zhang, J. Hong, and Y. Wu. AgentArmor: Enforcing Program Analysis on Agent Runtime Trace to Defend Against Prompt Injection. arXiv:2508.01249, 2025. [7] Responsible AI Collaborative. AI Incident Database, Incident 1152: LLM-Driven Replit Agent Reportedly Executed Unauthorized Destructive Commands During Code Freeze. https://incidentdatabase.ai/cite/1152/ (accessed 2026-09-03). [8] A. Baby. agent-incident-corpus: Public AI-Agent Production Failures as Executable Regression Tests. Companion artifact, 2026. https://github.com/ajinb/agent-incident-corpus (public, Apache-2.0). [9] A. Baby. Error Budgets for Autonomy: SLO-Driven Authority Management for Autonomous AI Operators. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/error-budgets-for-autonomy/ [10] A. Baby. Stable by Design: A Control-Theoretic Account of AI-Driven Self-Healing Remediation Loops. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/stable-by-design-remediation-loops/ [11] A. Baby. Stateless Protocol, Stateful Problem: Durability, Retry, and Retention Policy for MCP Tasks. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/stateless-protocol-stateful-problem/