Side Effects Are the Output: Evaluating AI Agents That Act on Live Systems

Ganeshkumar Ashokavardhanan
Microsoft
aganeshkumar@microsoft.com

Abstract

Most agent benchmarks grade a final artifact: a patch, a page state, or a database state. Live infrastructure does not fit that mold. An agent that acts on a running system creates a *system trajectory*: the intermediate states seen by controllers, users, and dependent services while the system is changing. A healthy-looking final state can hide an outage that already happened. Such agents need verifiers with three properties: score the whole trajectory, align observation depth with the fault class before the run, and account for the verifier's own probes as causal actions on the system. On a curated Kubernetes grid across three model families, standard outcome grading certifies 36 of 54 main-slice trajectories; the system-trajectory verifier flags 14 of those (39%) as hidden failures, concentrated on the two scenarios that need deeper-than-readiness checks. An additional 54 trajectories across varied prompts and scenarios replicate the pattern; a probe-on/off comparison shows verification can invalidate clean agent-only success claims. Beyond evaluation, the same verifier serves as a search score: an evolutionary loop over fault compositions surfaces cases where a repair agent declares success while a deeper property remains broken, and high-fitness invalid candidates expose missing preconditions in the verifier itself. For live-system agents, the trajectory is what must be graded, not just the final state.

1 Introduction

Coding, browser, and database agents are now evaluated by inspecting what they leave behind: a patch [Jimenez et al., 2024], a final page state [Zhou et al., 2024], or a database state [Yao et al., 2024]. Agents that act on live infrastructure (site-reliability agents, browser agents on real sites, database-administration agents) leave no such artifact that fully captures what happened. What matters for evaluation is the sequence of intermediate states the system passes through, observed in real time by users, controllers, and dependent services. A final cluster state that looks healthy does not encode the outage users experienced, the chain of controller reconciliations the action triggered, or the locks held against dependent services.

Consider an in-cluster name-lookup service that every other workload calls to resolve service names. It runs as multiple identical instances so any one can serve requests. In Kubernetes this is the `CoreDNS` Deployment; each instance is a pod. When one of those pods reports `NotReady`, an autonomous site-reliability agent may issue a *rolling replacement of all pods* (`kubectl -n kube-system rollout restart deployment/coredns`)—an over-reach: a safer first step would replace just the unhealthy pod (`kubectl -n kube-system delete pod <bad-coredns-pod>`) and let the cluster's self-healing controller respawn it. Both paths end with all CoreDNS pods Ready; only the trajectories differ. During the chosen rollout, in-cluster name lookups become briefly unreliable, dependent services see their callees as unreachable, and users experience elevated HTTP 5xx for the rollout window. Once new pods are Ready, lookups resolve

ACM CAIS Workshop on AI Agents for Discovery in the Wild.

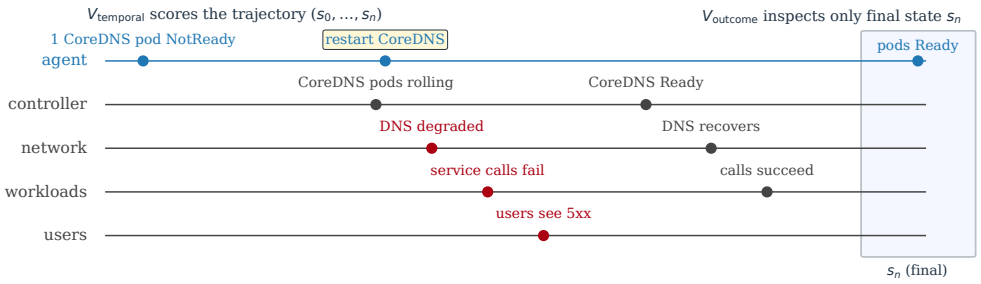


Figure 1: The system trajectory of a CoreDNS rollout restart: the sequence of states visible across controller, network, workload, and user lanes during the action. The standard end-of-run pod-readiness check inspects only the shaded final-state slice s_n, where pods are Ready; V_{temporal} scores the intervening DNS, service-call, and user-visible degradations.

again and a default pod-readiness check finds nothing wrong—the same verdict it would give the targeted single-pod path that avoided the outage entirely. There is no patch to inspect after the fact; the intermediate states traversed during the action are part of what must be evaluated. We call such sequences *system trajectories*. We argue that scoring system trajectories faithfully is an underexamined verifier-construction problem. When ignored, it produces evaluation results that systematically overstate live-system agent quality and yield insufficiently discriminating search signals for any discovery loop built on top of them.

To test the claim we ran 84 autonomous Kubernetes trajectories across three model families. Of the 36 outcome-passing main-slice runs, the trajectory verifier certifies only 22; the other 14 are hidden failures that fall on the two scenarios whose required check goes beyond pod-readiness (Tab. 1, Finding 1). A probe-on/off contrast demonstrates probe accounting on the latency-stress scenario (Finding 3). The failure mode is behavioral: when explicitly prompted to verify reachability, agents address it.

Contributions.

1. A *reframing* (§2): live-system agent executions should be scored as system trajectories, with three derived verifier-construction properties operationalized as Boolean predicates (§3).
2. An *empirical study* (§4) showing that *hidden failures*—runs that pass the default pod-readiness check but a deeper check rejects—concentrate in scenarios whose required check goes beyond pod-readiness, plus a controlled probe-on/off contrast showing verifier probes can themselves invalidate clean-success claims.
3. A *verifier-as-search-score study* (§5): the same verifier guides evolutionary search toward fault compositions where pod-readiness passes while a deeper check rejects, a random-search baseline shows pod-readiness alone cannot separate these from benign successes, and high-scoring invalid candidates expose missing preconditions in the verifier itself.

2 Live-System Agents Produce System Trajectories

An agent acting on a system produces a state sequence $s_0, s_1, \ldots, s_n$. We use *artifact* for a separable, final object that a verifier can inspect directly (e.g., SWE-Bench's patch [Jimenez et al., 2024], WebArena's final DOM state [Zhou et al., 2024]). **Artifact-producing agents** leave a verifier-inspectable artifact even when intermediate edits have local side effects. **System-trajectory-producing agents** (Kubernetes remediation, browser agents on real sites, DB administration) leave no artifact that fully encodes the behavior to be verified: s_n exists, but does not encode the intermediate states recorded in monitoring, observed by users, or causally entangled with other systems.

Three verifier-construction requirements follow.

1. **Temporal distribution**: score every state along the trajectory, because intermediate degradations affect users and dependent services even when the final state recovers.

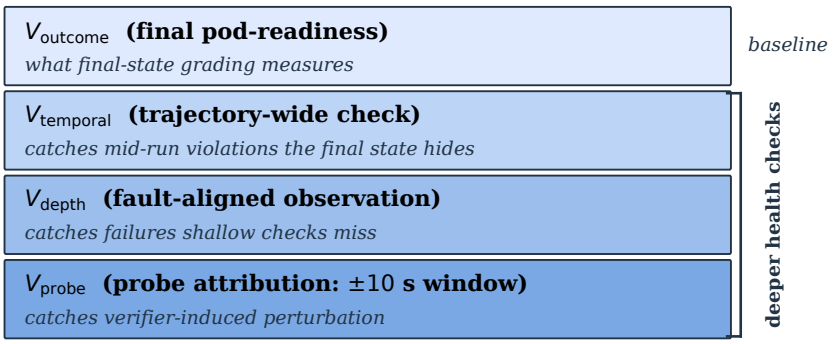


Figure 2: Verifier layers. The standard end-of-run pod-readiness check on s_n is the incumbent baseline; V_{temporal} checks the whole trajectory; V_{depth} checks the observation depth chosen for the scenario before the run; and V_{probe} accounts for the verifier's own probes. The layers are scored independently, not as pass-through gates.

2. **Fault-aligned observation**: match verifier observation depth to where the fault class manifests, and fix this choice before the run, because once the trajectory ends the unobserved states are gone.
3. **Probe accounting**: account for the verifier's own probes, because they act on the system being evaluated [Gait, 1986].

Together, these three expose failure classes that final-state verifiers (those that score only s_n) miss. We do not claim they are sufficient or exhaustive; we claim they are necessary to avoid the specific blind spots we document in §4. Among the live-system agent benchmarks we surveyed (including AIOpsLab [Chen et al., 2025a] and ITBench [Jha et al., 2025]), none treats all three as explicit verifier-construction requirements over system trajectories.

Relation to site reliability engineering (SRE) measurement. SRE already measures availability integrals; our temporal-distribution property is one. The benchmark-specific gaps are the other two: without explicit *fault-aligned observation* fixed in advance, an evaluator can pick depth post-hoc; and in low-traffic benchmark clusters verifier probes are no longer second-order, so *probe accounting* becomes necessary.

3 Operationalizing the Three Properties

For Kubernetes, the harness records four observation depths every 5 s: D1=pod readiness, D2=endpoints populated (a Service has selected backend addresses), D3=in-cluster reachability (an HTTP request returns 200), D4=control-plane component health (full set of monitored components in App. A). Current live-system benchmarks check D1 on s_n; we write V_{outcome} for this incumbent verifier.

We add three verifier predicates that aggregate the same observations differently (Fig. 2): V_{temporal} scores the whole trajectory; V_{depth} checks the observation depth committed to before the run; V_{probe} accounts for the verifier's own probes. V_{temporal}, V_{depth}, and V_{probe} are scored independently of V_{outcome} and of each other; the framing in Fig. 2 separates the incumbent baseline from three added deeper-health-check predicates, not a dependency stack.

Two coordinate systems. Two axes vary independently: *what* evidence the harness records (D1–D4) and *how* a verifier aggregates that evidence (V_{outcome}, V_{temporal}, V_{depth}, V_{probe}). Hidden failures arise when these are mismatched—V_{outcome}'s D1-on-s_n stance misses S-4's D3 fault (pods stay Ready while the Service is unreachable) and S-5's D4 fault (application reachability is fine while the control plane is degraded). We call S-1, S-2, S-3, S-6 the *D1-check* scenarios (V_{outcome} suffices) and

Table 1: Scenarios used throughout the paper. *Required check* is the evidence the benchmark commits to before the run; we call a scenario *D1-check* when that check is D1 pod-readiness and *deeper-check* when it requires evidence beyond pod-readiness (D3 or D4). S-1–S-6 form the main grid; S-7 and S-8 are targeted diagnostics.

ID	Plain description	Required check
S-1	Node `kubelet` stop	D1: pods Ready
S-2	CrashLoopBackOff (missing ConfigMap)	D1: pods Ready
S-3	Cascading evictions via NoExecute taint (a node taint that evicts pods which do not tolerate it)	D1: pods Ready
S-4	Service unreachable (wrong selector + deny-all NetworkPolicy)	D3: in-cluster HTTP request returns 200
S-5	Control-plane latency stress (`tc netem`, a Linux traffic-control tool that injects artificial network delay)	D4: critical control-plane pods Running
S-6	Bad rollout (image tag does not exist)	D1: pods Ready
S-7	Healthy pods + endpoints, ingress NetworkPolicy blocks cross-namespace traffic	D3: cross-namespace HTTP probe returns 200
S-8	20-replica `payment-api` pinned to a single worker (HA spread-policy violation); production-realistic 15 s `preStop` drain hook on each pod	D1: pods Ready

S-4, S-5 the *deeper-check* scenarios (D3 or D4 evidence is required); the per-scenario depth choice is fixed in the YAML before any run.

Property 1: temporal distribution (verifier V_{temporal}). *Operationally: was the system healthy throughout, not just at the end?* Score the entire sequence, not only s_n, because the system spent time in each intermediate state and the cost paid during that interval (downtime, degraded service) is paid regardless of where the trajectory ends. With state captured every 5 s, V_{temporal} fails if any tick has workload availability below 0.85 or any critical control-plane pod not Running.

Property 2: fault-aligned observation (verifier V_{depth}). *Operationally: did the agent fix the actual problem, not just the surface symptom?* The harness records all four observation depths every tick. The verifier's *committed depth*—which depth's check defines success—is fixed per scenario before the run, chosen to match the layer at which the fault class manifests. The choice is per-scenario because different fault classes manifest at different depths: a NetworkPolicy hides a service while pods stay Ready (S-4 needs D3); control-plane latency stress hides while application reachability is fine (S-5 needs D4).

V_{depth} := the committed-depth check passes on the final state. For D3 commitments this means an actual in-cluster HTTP request returning 200, not a D2 endpoint-readiness proxy: endpoints can be populated while the service is still broken (App. A). Choosing depth after seeing the trajectory induces selection bias (App. E).

Property 3: probe accounting (verifier V_{probe}). *Operationally: did the verifier's own measurements perturb the system it was scoring?* A verifier of a system-trajectory-producing agent uses probes that are themselves causal actions on the system: an HTTP request sends real traffic; a database query takes real locks; `kubectl exec` spawns a real process. V_{probe} := no critical-pod failure first appears within ± 10 s of any in-trajectory probe AND no probe's own latency exceeds 5 s. A V_{probe} failure marks the run as not a clean agent-only success; the probe-on/off comparison in §4 surfaces this contamination empirically.

Reference implementation. `compute_properties.py` implements the three predicates in $\sim$50 lines of Python over the 5 s state-capture log; thresholds, per-scenario `committed_depth`, and the ± 10 s probe-attribution window are fixed in YAML before any run (App. G, sensitivity sweeps in App. F).

4 Empirical Study

Standard outcome grading certifies 36 of 54 main-slice trajectories as successes, but the full trajectory verifier certifies only 22. The other 14 outcome-passing runs are *hidden failures*—runs the outcome verifier passes but the conjunction does not.

We run 84 trajectories across Claude Sonnet 4.5 [Anthropic, 2025b], Claude Haiku 4.5 [Anthropic, 2025a], and GPT-5-mini [OpenAI, 2025] on the scenario grid (Tab. 1). The **main slice** is 54 trajectories (3 models $\times$ 6 scenarios $\times$ 3 runs) with default prompts and verifier probes on; an additional 30 trajectories vary one factor at a time (constrained-prompt arm on S-1 and S-4; probe-off arm on S-4 and S-5), plus targeted diagnostics on S-7 and S-8 (Findings 2 and 4). Every run uses a fresh `kind`[1] cluster; per-run tables in App. A–B. The four D1-check scenarios (S-1–S-3, S-6) form a baseline where the verifier suite agrees with outcome grading by construction; the empirical question is what happens on the two deeper-check scenarios—S-4 (service-unreachable) and S-5 (control-plane latency)—whose required check is committed beyond pod-readiness in the scenario YAML before any run.

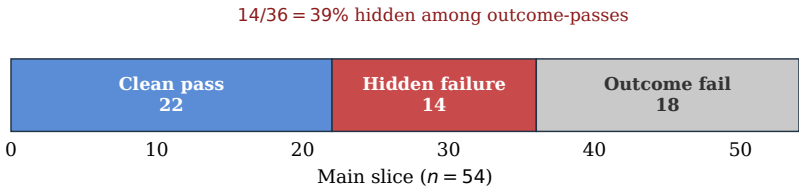


Figure 3: Outcome grading passes 36 of the 54 main-slice trajectories, but the system-trajectory conjunction certifies only 22; the remaining 14 are hidden failures.

Finding 1: hidden failures land on the two deeper-check scenarios. Of the 36 outcome-passes on the main slice, 22 are clean and 14 are hidden failures (Fig. 3). Every hidden failure falls on S-4 (service-unreachable) or S-5 (control-plane latency); none appear on the four D1-check scenarios. All three model families show the same behavioral pattern: agents repair the surface symptom but leave the deeper-layer fault intact, and outcome grading marks the run as a success.

The S-4 mechanism is the clearest example: every baseline run patches the Service selector, watches endpoints populate, and declares success while a deny-all NetworkPolicy still blocks traffic and the required in-cluster reachability evidence is absent. The gap is behavioral, not definitional: when the agent is explicitly prompted to verify reachability (constrained-prompt arm), Sonnet passes $3/3$ and Haiku $1/3$ rather than $0/3$—the deeper-check failure reflects what the agent chose to do, not what the scenario forced. Per-property attribution for all 14 hidden failures is in App. B.

Finding 2: S-7 isolates the same failure within one diagnostic task. S-7 reproduces the deeper-check blind spot inside a single scenario. The protected service has Ready pods and populated endpoints, but an ingress NetworkPolicy blocks traffic from the verifier's namespace. A scripted known-good sequence (list, inspect, delete the policy, reverify) resolves the fault in 3/3 trials (a solvability check, not a model baseline). Across nine LLM runs, every run satisfies pod-readiness but seven fail the committed in-cluster reachability check (V_{depth}): agents typically restore same-namespace clients while leaving the cross-namespace ingress blocked. Per-model and duration details in App. B (Tab. 10).

Finding 3: probe accounting can invalidate clean-success claims. S-5 (control-plane latency) injects network delay and prompts agents to investigate slow `kubectl` operations. Toggling verifier probes on/off with everything else fixed moves V_{probe} from $1/9$ to $9/9$ (expected) and the full conjunction from $0/9$ to $4/9$: four runs rejected with probes on are clean once the verifier's own action is removed. When the verifier's own action is slow or coincident with the failure it is scoring, the run is not a clean agent-only success.

[1] `https://github.com/kubernetes-sigs/kind`

Finding 4: S-8 isolates V_{temporal} as a sole discriminator. S-8 (the spread-violation scenario; fault recipe in App. C) places 20 `payment-api` replicas on a single worker, each with a 15 s `preStop` drain hook. The alert reports the concentration as an HA-policy violation and asks for remediation plus a healthy spread, without hinting at how.

This scenario is designed so that two remediations lead to identical healthy final states but differ in transient cost. The gentle oracle removes the node pin and lets *RollingUpdate* redistribute gradually; the aggressive oracle switches the strategy to *Recreate*, terminating all 20 pods before starting new ones. Both end healthy (spread $5/5/5/5$); only the aggressive path drops availability below the 0.85 floor mid-run. V_{temporal} is the sole verifier that distinguishes them—V_{outcome}, V_{depth}, and V_{probe} all pass on both paths. Tab. 2 reports the scripted gentle/aggressive control pair; the 15-run model breakdown is in App. C.

Table 2: S-8 construct-validity pair. Both scripted oracles end with the same healthy spread; V_{temporal} is the only verifier that distinguishes them. Per-model results (15 LLM runs) in App. C.

Runner	Outcome	V_{temporal}	V_{depth}	V_{probe}	Notes
Scripted `ideal-gentle`	1/1	1/1	1/1	1/1	*RollingUpdate*; min tick avail ≈ 0.94.
Scripted `ideal-aggressive`	1/1	0/1	1/1	1/1	*Recreate*; min tick avail 0.744 (below 0.85 floor).

The construct-validity pair confirms that V_{temporal} carries discriminative work the other three verifiers cannot: the aggressive oracle's final state is healthy, yet a single sample-tick records availability 0.744, well below the 0.85 floor. Across the 15 model runs, none organically chose the *Recreate* strategy—a positive finding about current models, but not grounds to remove V_{temporal}: future models, adversarial prompts, or prompt regressions may act more forcefully.

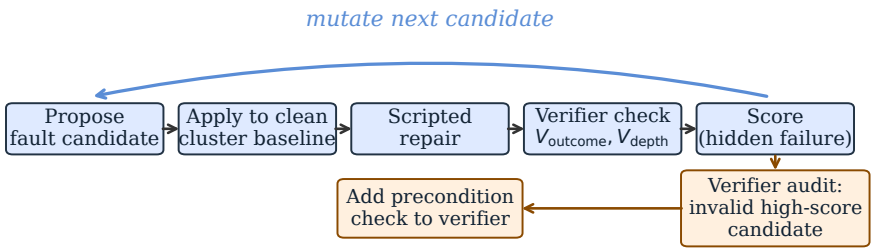
Finding 4b: S-8 also reveals a verifier-suite completeness gap. Two of five GPT-5-mini runs passed all four verifiers while leaving every pod on `worker3`—the suite covers availability and depth but not *where* pods run, so topology alerts (spread, anti-affinity) need an explicit distribution-property check (§7).

5 From Evaluation to Discovery: The Verifier as Search Score and Audit Target

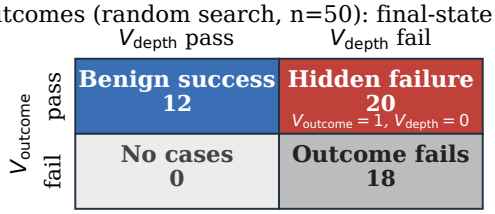
The three properties were motivated by evaluation, but the same verifier family also supplies search scores for discovery loops that search over live-system perturbations. The scenarios in §4 were hand-curated, and that hand-curation is a practical bottleneck on coverage. A search whose candidates can be replayed on a real cluster and graded by the verifier inherits a score that points at the case live-system discovery cares about: *the repair procedure declared success but a deeper property failed*. Figure 4 shows both roles: the verifier supplies the search score, and high-scoring invalid candidates become verifier-audit findings. We add a precondition guard that requires the expected workload to exist and the clean-cluster assumptions to hold before scoring a candidate.

Setup and scope. We instantiate the loop with OpenEvolve [Sharma, 2025], an evolutionary coding agent. Each candidate is a Python `fault_composition()` returning a list of `kubectl`- or `docker`-level operations. The evaluator restores a clean cluster baseline (`auth-service` is the target workload the loop is supposed to fault and the operator is supposed to repair), applies the candidate, runs a scripted repair operator (a deterministic remediation script; App. H, Tab. 16), and computes the verifier predicate. Fitness=1.0 means the scripted operator declared success while the committed-depth check still fails. The search score is the scoped S-4-style objective: outcome grading passes while V_{depth} fails; V_{probe} is already tested by the controlled experiment in §4, and V_{temporal} is held out because the scripted repair operator's trajectory is short and does not exercise the mid-run-degradation signal V_{temporal} is designed to catch — so the discovery question here is whether a committed-depth verifier can guide scenario generation. The loop generates faults under a given committed depth; depth itself remains expert-specified per scenario (Tab. 6), and the symmetric direction (proposing a depth from a fault description) is left to future work.

(a) Verifier-guided search loop



(b) Candidate outcomes (random search, n=50): final-state grading admits 32/50



Committed depth separates hidden failures from benign successes.

Figure 4: Using the verifier as a search score and audit signal. **(a)** OpenEvolve proposes fault compositions; each candidate starts from a clean cluster baseline, applies its perturbation, runs a scripted repair operator, and is scored by the hidden-failure predicate (V_{outcome} passes, V_{depth} fails). Invalid high-score candidates expose missing verifier assumptions and motivate a precondition guard. **(b)** On the 50-trial random-search baseline, final-state grading admits 32 candidates as healthy, while committed depth separates 20 hidden-failure candidates from 12 benign successes.

Prompt-information sweep. We ran three prompt versions (*full*, *partial*, *minimal*) at five RNG seeds each (claude-haiku-4.5, temperature 1.0, budget 10 iterations), varying how much fault vocabulary the prompt reveals. Across 15 runs, 14 reach fitness=1.0 with median first-hit iteration 1. Under the minimal prompt, the loop produces typed compositions such as `wrong-targetPort` and `broken-readinessProbe` alongside NetworkPolicy cases (App. H). The pretraining baseline is uncontrolled, so this shows verifier-guided composition: the prompt need not enumerate fault building blocks for mutation to produce hidden-failure compositions.

The verifier does the discriminating work. On the six-primitive fault library, evolutionary search does not separate from random sampling: random search hits fitness=1.0 in 20/50 trials. The verifier—not the search algorithm—is the discriminating component. On the random baseline, end-state grading passes 32/50 while V_{depth} passes 12/50; the committed-depth verifier separates the 20 hidden-failure compositions that pod-readiness alone cannot distinguish from benign successes.

Discovery as verifier audit. The search also finds cases where the verifier's assumptions are incomplete. Three classes of invalid high-fitness compositions appear: candidates that never create the target workload, candidates that mutate control-plane state outside the evaluator's reset boundary, and candidates that corrupt static-pod manifests (App. H.3). These are attacks on the verifier, not legitimate faults. The fix is a precondition predicate that requires the target workload to exist at s_0 and the clean-cluster baseline to hold. The broader lesson: discovery loops do not merely use verifiers; they help specify them.

6 Related Work

The scenario-class hidden-failure result reflects what current live-system benchmarks measure versus what the three properties imply they should. Full benchmark mapping in App. D; the clusters below state the methodological difference.

Artifact-regime benchmarks. SWE-Bench [Jimenez et al., 2024], WebArena [Zhou et al., 2024], and AgentBench [Liu et al., 2024] score a final inspectable outcome, so the three properties do not apply.

Tool-trace benchmarks. τ-bench [Yao et al., 2024] scores final database state and message content. τ^2-bench [Barres et al., 2025] adds trajectory-level action matching (exact tool-call-sequence comparison against a canonical solution) as an official criterion, but neither benchmark aligns verifier observation depth to a fault class in advance or has a probe-attribution mechanism.

Live-system / incident benchmarks (closest setting). AIOpsLab [Chen et al., 2025a] and IT-Bench [Jha et al., 2025] measure recovery-latency stopwatches on per-task resolution criteria, not trajectory integrals. A trajectory with mid-run degradation that ends in a passing resolution check has the same Time-to-Mitigate as a clean recovery; V_{temporal} catches the difference, the stopwatch cannot. Neither benchmark commits observation depth before the run (V_{depth}) or separates verifier probe activity from system response (V_{probe}). STRATUS [Chen et al., 2025b] introduces a *Transactional No-Regression* (TNR) safety specification for its SRE agent: a remediation must not degrade the system beyond its pre-action state. TNR is an agent-design constraint; our V_{temporal} is the corresponding verifier-side predicate that scores whether any agent (not just one designed with TNR) actually maintained trajectory availability.

Process / wrong-path verifiers. Near-Miss [Rabinovich et al., 2026] detects agents reaching correct outcomes via wrong paths. Its policy verifier is fault-aligned by definition; the customer-service domain has no external probes. Our contribution extends to live-system domains where probe accounting is unavoidable and the right observation depth cannot be assumed.

Trajectory-aware verifiers for browser and safety agents. The Universal Verifier and CUAVerifierBench [Rosset et al., 2026] separate *process* from *outcome* scoring for browser agents, using rubric-driven LLM judges over screenshot trajectories. ATBench [Li et al., 2026] benchmarks trajectory-level safety for tool-using agents over 1,000 annotated trajectories spanning safe and unsafe behaviors. Neither formalizes probe accounting or fault-aligned observation as first-class verifier-construction objects; both are complementary to the framework here.

Agent reliability metrics. Rabanser et al. [Rabanser et al., 2026] decompose agent reliability into consistency, robustness, predictability, and safety, arguing that single success metrics obscure operational flaws. Their metrics characterize *agent-level* properties across many runs; ours characterize *verifier-construction* properties for scoring a single run's system trajectory. The two perspectives are complementary: an agent can be consistent (same behavior every run) yet still produce hidden failures that only a trajectory-aware verifier detects.

Discovery and chaos engineering. ADRS [Cheng et al., 2025] argues systems research is amenable to AI-driven discovery when reliable verifiers are available; our framework identifies what "reliable" must mean for live-system-agent evaluation. Chaos engineering [Basiri et al., 2016] establishes the fault-injection vocabulary we draw from; we add verifier-construction requirements for grading agent responses to those faults.

7 Limitations

The main experiments are in Kubernetes on `kind`; a transferability check on Docker Compose is reported below, but full multi-platform demonstration is future work. We test three model families plus a scripted known-good sequence; the model set is not exhaustive. `kind` cannot replicate production traffic patterns, so probe-accounting effects may differ in production. We claim the three properties expose failure classes final-state verifiers miss; we do not claim sufficiency or exhaustiveness. The construct-validity scenario S-8 (Finding 4b) makes this concrete: the suite does not cover topology acceptance criteria (spread, anti-affinity), so alerts of that shape need an additional check. Depth assignment per scenario is currently expert-authored (Tab. 6); automated depth proposal is future work. V_{temporal} is a binary threshold over per-tick availability, with no request-criticality weighting or user-minutes-of-impact accounting; the full SRE stack would require real user-traffic loadgen and per-service SLOs.

Transferability check: Docker Compose. As a transferability check, we implemented a minimal scenario on Docker Compose—a non-Kubernetes service-composition substrate with no control plane, no declarative reconciliation, and imperative lifecycle management. A two-service stack (`frontend` nginx reverse proxy + `api` Flask service) has a faulted upstream port in the nginx config: the proxy targets port 9999 while the API listens on 8080. Both containers are healthy (`docker ps`); end-to-end requests return 502. The committed depth is D3 (end-to-end HTTP request through the proxy returns 200). Two scripted oracle controls fix the port: *gentle*—edit the config inside the running container and `nginx -s reload` (no service interruption); *aggressive*—`docker compose down`, fix the config, `docker compose up -d` (full teardown/rebuild). The verifier probes every 1 s for 30 s. Both paths end with D3 passing (V_{outcome} ✓, V_{depth} ✓); the gentle path records 90% availability (3 ticks down from the pre-existing fault), the aggressive path records 73% (8 ticks down—the 5 additional ticks are the teardown/rebuild window). V_{temporal} flags the trajectory cost difference that outcome grading cannot see, exactly as in the Kubernetes S-8 scenario. Full empirical agent runs on Compose are future work; this check confirms the verifier design transfers to a non-Kubernetes orchestration substrate with the same properties and the same hidden-failure pattern.

8 Conclusion

This paper reframed the unit of evaluation for live-system agents from a final artifact to a system trajectory and derived three verifier-construction requirements over it: temporal distribution, fault-aligned observation, and probe accounting. A curated Kubernetes study located hidden failures on the two scenarios whose required check was committed before the run to go beyond pod-readiness, and the gap is *behavioral*, not a metric artifact: when prompted to verify the deeper layer, agents reach it; when not, they stop where the verifier stops looking. Probe accounting and temporal distribution show the same pattern. The verifier's own probes can be part of what is being scored, so accounting for them turns some apparent failures back into clean successes. And two trajectories with the same healthy final state can differ in transient cost that only temporal scoring detects.

The same verifier framework did three jobs here: it scored runs, scored a search over fault compositions, and its high-fitness invalid candidates exposed missing preconditions in its own definition. Discovery loops and verifier construction are mutually informing rather than sequential: the loop does not only consume a verifier, it audits one. Two implications follow. The verifier deserves to ship as a first-class artifact of the benchmark, with its depth-alignment and probe-accounting choices visible, because changing those choices changes which agent behaviors look successful. And a verifier that has not been stress-tested by some search procedure is most likely incomplete in ways its authors cannot anticipate; auditing the verifier with a search procedure is part of building it, not an afterthought. Once side effects are scored, discovery can search the space of trajectories the system actually produces, rather than the space of final states it happens to land in.

References

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *AAAI*, 2018.

Anthropic. Introducing Claude Haiku 4.5. `https://www.anthropic.com/news/claude-haiku-4-5`, 2025a.

Anthropic. Introducing Claude Sonnet 4.5. `https://www.anthropic.com/news/claude-sonnet-4-5`, 2025b.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv:2506.07982*, 2025.

Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016. doi: 10.1109/MS.2016.60.

Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds. *arXiv:2501.06706*, 2025a.

Yinfang Chen et al. STRATUS: An LLM-based multi-agent system for autonomous SRE of cloud services. *arXiv preprint arXiv:2506.02009*, 2025b.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alexander Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How AI is upending systems research. *arXiv:2510.06189*, 2025.

Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3): 225–233, 1986.

Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Pavankumar Murali, Jae-wook Ahn, Debanjana Kar, Ameet Rahane, Carlos Fonseca, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. *arXiv:2502.05352*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *ICLR*, 2024.

Yu Li et al. ATBench: A diverse and realistic agent trajectory benchmark for safety evaluation and diagnosis. *arXiv preprint arXiv:2604.02022*, 2026. Available at `https://github.com/LiYu0524/ATbench`.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2024.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *ICLR*, 2024.

OpenAI. GPT-5 mini (model card). `https://platform.openai.com/docs/models/gpt-5-mini`, 2025.

Stephan Rabanser et al. Towards a science of AI agent reliability. *arXiv preprint arXiv:2602.16666*, 2026.

Ella Rabinovich, David Boaz, Naama Zwerdling, and Ateret Anaby-Tavor. Near-miss: Latent policy failure detection in agentic workflows. *arXiv:2603.29665*, 2026.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. The art of building verifiers for computer use agents. *arXiv preprint arXiv:2604.06240*, 2026.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. `https://github.com/algorithmicsuperintelligence/openevolve`, 2025.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv:2406.12045*, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *ICLR*, 2024.

Appendix

A Scenario Specifications

This appendix gives the full injection method and depth-check threshold for each scenario; the body summary in Tab. 1 (page 4) lists the plain-English description and required check.

For S-5, D4 records critical-pod health rather than direct traffic-control removal; S-5 is the probe-accounting stressor. S-7 is the targeted diagnostic extension used in §4: the `payment-service` Deployment has three Ready pods and populated endpoints, while

Table 3: Plain-language glossary for Kubernetes vocabulary used throughout the paper. Provided for readers from adjacent areas who may not be Kubernetes specialists.

Term	Plain meaning
`kubelet`	Per-node agent that runs containers and reports node/pod status to the cluster.
Pod (Ready)	Smallest deployable unit (one or more containers); "Ready" means the pod's health checks pass.
ConfigMap	Key-value store for application configuration.
Service / selector	A stable cluster-internal address for a group of pods; the selector is the label that decides which pods to route to.
Endpoints	The list of backend pod addresses currently behind a Service.
ClusterIP	A Service's cluster-internal IP address.
NetworkPolicy	Firewall rule that filters pod-to-pod or namespace-to-namespace traffic.
Cordon / drain	Mark a node unschedulable; evict its workloads to elsewhere.
Taint (`NoExecute`)	Marker on a node that evicts pods which do not tolerate the taint.
PodDisruptionBudget (PDB)	Constraint limiting how many pods of a workload can be evicted simultaneously.
ImagePullBackOff	Pod cannot start because its container image is unavailable.
CrashLoopBackOff	Pod repeatedly crashes and is being restarted with backoff.
Control plane	Cluster-level components: `apiserver`, `etcd`, `scheduler`, `controller-manager` (plus add-ons like `coredns`, `kube-proxy`, `kindnet`).
`tc netem`	Linux traffic-control tool for injecting artificial network delay.
`kind`	"Kubernetes-in-Docker"; runs a full cluster as Docker containers for testing.

Table 4: Full injection method and committed depth for each scenario.

ID	Scenario	Injection method	Depth	Class
S-1	Node NotReady	`docker exec worker2 systemctl stop kubelet`	D1	D1-check
S-2	CrashLoopBackOff	Deploy with reference to non-existent ConfigMap	D1	D1-check
S-3	Cascading evictions	NoExecute taint on 2/4 workers	D1	D1-check
S-4	Service unreachable	Wrong selector + deny-all NetworkPolicy	D3	deeper-check
S-5	Control-plane latency	Inject 500 ms artificial network delay on the control-plane node (`tc netem`)	D4	deeper-check
S-6	Bad rollout	Image set to non-existent tag	D1	D1-check
S-7	Payment reachability	Healthy pods/endpoints + ingress-blocking NetworkPolicy	D3	deeper-check
S-8	HA spread violation	20 `payment-api` replicas pinned to a single worker via `nodeName`; pods carry a 15 s `preStop` drain hook	D1	D1-check

`NetworkPolicy/payment-isolation-policy` has an empty ingress rule list that blocks client traffic. Its D3 probe runs from the verifier namespace `atlas-probe`, not from the application namespace. The released code identifies S-7 as `s8` (artifact-directory naming preserved across versions); the body uses S-7. S-8 (§4) is the construct-validity extension; its fault recipe and the two oracle recipes used to bracket the discriminative range are given in App. C below. The released code identifies S-8 as `s9`.

A representative scenario YAML (S-4):

Listing 1: Scenario specification for S-4 (service-unreachable). The committed depth `v3` is set before any run.

```
id: S-4
fault:
  - kubectl-apply: | # auth-service Service with wrong selector
```

```
    ... # + deny-all NetworkPolicy
verifier:
  committed_depth: v3 # V_depth commitment, set ex ante
  probe_cadence_ticks: 4 # one D3 probe per 20s
  protected_service: auth-service
prompt: |
  Alert: service 'auth-service' is unreachable. ...
```

The non-LLM S-7 known-good sequence used only to establish solvability:

Listing 2: Scripted known-good sequence for S-7 solvability check.

```
kubectl get networkpolicy -n default
kubectl get networkpolicy payment-isolation-policy -n default -o yaml
kubectl delete networkpolicy payment-isolation-policy -n default
kubectl exec -n atlas-probe probe -- \
  wget -T 5 -qO- http://payment-service.default.svc.cluster.local
```

Table 5: Per-scenario alert prompts shown to the agent at the start of each run, identical across all three model families. Each prompt also includes a tools clause naming `kubectl` (and `docker` for S-1, S-5, S-7) and the cluster context; the scenario-specific alert text is shown below. Constrained-prompt addenda for S-1 and S-4 are appended in the §3 prompt-fix asymmetry experiment (App. B).

ID	Alert text
S-1	Node `worker2` has been NotReady for 5 minutes. Investigate and remediate to restore cluster capacity.
S-2	Deployment `payment-app` has pods in CrashLoopBackOff. Investigate and remediate.
S-3	2 of 4 worker nodes are unschedulable due to NoExecute taints (`atlas-disk-full=true`). Investigate and remediate to restore scheduling capacity.
S-4	Service `auth-service` is unreachable. No endpoints are populated. Investigate and remediate.
S-5	`kubectl` operations against the cluster are extremely slow (multi-second latencies). Investigate and remediate.
S-6	Deployment `web` has pods in ImagePullBackOff. Investigate and remediate.
S-7	In-cluster requests to service `payment-service` (namespace `default`) time out. Endpoints appear populated and pods are Ready, but clients cannot reach the service. Investigate and remediate.
S-8	Workload `payment-api` (namespace `default`) has all 20 of 20 replicas concentrated on a single node (`worker3`), violating the cluster's high-availability spread policy. Remediate and verify the workload is healthy and distributed across worker nodes.
Constrained-prompt addenda (appended to the alert above):	
S-1 cons.	Before executing any mutating action (drain, delete, cordon, etc.), estimate its impact on pod availability. If it would drop pod availability below 90%, choose a lower-impact alternative.
S-4 cons.	After fixing any issue, verify end-to-end connectivity by testing the service. If it is still unreachable, investigate additional potential causes rather than assuming the fix will propagate.

Two notes on prompt content. (1) Each baseline alert names a realistic symptom an SRE alerting system would actually fire on (`NotReady`, `CrashLoopBackOff`, empty endpoints, slow `kubectl`, etc.); we do not embed the root cause or the required remediation depth. The S-4 alert in particular surfaces the D2-level symptom ("no endpoints are populated"), and most baseline runs accordingly stop at D2-level repair (selector patch, endpoints populate, declare done) without checking D3 reachability against the deny-all NetworkPolicy. The framework's claim is that the verifier's committed depth must be deeper than the alert content, because alerts surface symptoms and not root causes; the constrained-prompt arm tests what happens when the alert is augmented to require end-to-end verification. (2) System prompts are vendor defaults: the Claude CLI and copilot CLI are invoked with no custom system prompt, and global `CLAUDE.md`/`AGENTS.md`/`.copilot` files are stripped via a sandboxed `HOME` so cross-model comparison is on equal footing.

The **verifier layers** (committed in advance per scenario): D1 = $\geq 95\%$ of pods ready; D2 = protected services have non-empty `Endpoints.subsets.addresses`; D3 = in-cluster `wget` against the protected service's ClusterIP returns 200 within 3 s; D4 = every control-plane critical pod (apiserver, etcd, scheduler, controller-manager, coredns, kindnet, kube-proxy) is `Running`. D4 is a component-health observation, not a direct API-latency service-level objective. When D3 is the required check, the final-state record must contain an actual D3 probe result; the strict scorer rejects missing D3 evidence rather than substituting endpoint readiness.

Fault-class to committed-depth mapping. Depth assignment per scenario is currently expert-authored, but across the seven scenarios above the chosen depth follows mechanically from the layer at which the injected fault operates (Tab. 6): the chosen depth is the lowest observation layer at or above the fault layer, so the expertise burden is one-time-per-fault-class, not one-time-per-scenario. Automated depth proposal — a symmetric counterpart to the verifier-guided fault generation in §5 — is left to future work.

Table 6: Implicit fault-class to committed-depth mapping in the seven-scenario grid. Depth is the lowest observation layer at or above the layer the fault operates on.

Fault class	Examples	Fault layer	Committed depth
Pod lifecycle (`kubelet` stop, CrashLoop, `NoExecute` taint, bad image tag)	S-1, S-2, S-3, S-6	node / pod	D1
Service routing (selector mismatch, NetworkPolicy isolation)	S-4, S-7	service / policy	D3
Control-plane perturbation (`tc netem` artificial delay)	S-5	control plane	D4
HA placement (single-node concentration; remediation *path*-sensitive)	S-8	pod scheduling	D1

B Per-Model and Per-Run Verifier Outcomes

Table 7: Run-slice key for the empirical results in this paper. The 84 trajectories of `v11_full` are partitioned into 28 cells (scenario × model × variant × probe-mode), three runs per cell. S-7 and S-8 are additional targeted diagnostic extensions; S-7 uses three runs per cell plus a non-LLM known-good sequence (solvability check), and S-8 uses five runs per cell plus two scripted oracle controls (construct-validity bracket). Several headline numbers are computed on overlapping subsets of these runs.

Slice	n	Used for
Original `v11_full` grid	84	Q1–Q3 experiment count
Strict baseline+probe-on, 3 models × 6 scen. × 3 runs	**54**	**headline (Tab. 8); per-class Fisher; sign test**
Constrained-prompt, S-1 + S-4, Sonnet/Haiku × 3 runs	12	P2 prompt-fix asymmetry (§3)
Probe-off baseline, S-4/S-5, 3 models × 3 runs	18	P3 contrast (Tab. 12)
S-5 P3 paired arms (probe-on vs. probe-off, 3 models)	9+9	primary P3 Fisher exact
S-7 LLM diagnostic extension, 3 models × 3 runs	9	targeted committed-depth contrast (Tab. 10)
S-7 non-LLM known-good sequence	3	solvability check only; not a model baseline
S-8 LLM agent matrix, 3 models × 5 runs	15	construct-validity prevalence (Tab. 2)
S-8 scripted oracle controls (gentle + aggressive)	2	construct-validity bracket (Tab. 2)

S-4 prompt-fix asymmetry (constructive-section evidence). The 12 constrained-prompt runs are the empirical basis for §3's claim that committed-depth verification should sit in the harness rather than be delegated to a prompt addendum. We tested the agent-side alternative on S-1 and S-4 with

Table 8: Per-class decomposition of hidden failures on the main slice (n=54). D1-check = scenarios whose required check is D1 pod readiness (S-1, S-2, S-3, S-6); deeper-check = scenarios requiring evidence beyond pod readiness (S-4: D3 reachability, S-5: D4 control-plane component health). The hidden-failure rate is the fraction of outcome-passing trajectories that the four-verifier conjunction marks as failing. The contingency $\left(\begin{smallmatrix}14 & 0\\ 0 & 22\end{smallmatrix}\right)$ rejects equal hidden-failure rates between the two classes at Fisher exact 2-sided $p = 2.6 \times 10^{-10}$. Caveat: V_{depth}'s D1 check (availability ≥ 0.95) is a strict subset of V_{outcome}'s availability gate, so the D1-side 0/22 cell is partly structural for V_{depth}; the deeper-check 14/14 is fully empirical, with S-4 driven by V_{depth}'s independent in-cluster reachability check and S-5 driven by V_{probe} (§4, Finding 1).

Class	Scenarios	n	Outcome pass	Conj. pass	Hidden / Outcome-pass
D1-check	S-1, S-2, S-3, S-6	36	22/36 (61 %)	22/36 (61 %)	0/22 (0 %)
Deeper-check	S-4, S-5	18	14/18 (78 %)	0/18 (0 %)	14/14 (100 %)
All	S-1..S-6	**54**	**36/54 (67 %)**	**22/54 (41 %)**	**14/36 (39 %)**

Table 9: Per-cell verifier pass counts (n=3 per cell) on the 84-trajectory `v11_full` grid. Cells are organized by scenario and model. Variant: "base" = baseline prompt; "cons" = constrained prompt (reachability-probe addendum). Probe mode: "on" = in-trajectory D3 probes active; "off" = no in-trajectory probes (final committed-depth probe still runs after the agent halts).

Scenario	Model	Variant	Probe	Depth	Outcome	V_{temporal}	V_{depth}	V_{probe}	Conj.
S-1	sonnet	base	on	D1	0/3	3/3	0/3	3/3	0/3
S-1	sonnet	cons	on	D1	0/3	3/3	0/3	3/3	0/3
S-1	haiku	base	on	D1	0/3	3/3	0/3	3/3	0/3
S-1	haiku	cons	on	D1	0/3	3/3	0/3	3/3	0/3
S-1	gpt-5-mini	base	on	D1	1/3	3/3	1/3	3/3	1/3
S-2	sonnet	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-2	haiku	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-2	gpt-5-mini	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-3	sonnet	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-3	haiku	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-3	gpt-5-mini	base	on	D1	3/3	3/3	3/3	3/3	3/3
S-4	sonnet	base	on	D3	3/3	3/3	0/3	3/3	0/3
S-4	sonnet	cons	on	D3	3/3	3/3	3/3	3/3	3/3
S-4	sonnet	base	off	D3	3/3	3/3	0/3	3/3	0/3
S-4	haiku	base	on	D3	3/3	3/3	0/3	3/3	0/3
S-4	haiku	cons	on	D3	3/3	3/3	1/3	3/3	1/3
S-4	haiku	base	off	D3	3/3	3/3	0/3	3/3	0/3
S-4	gpt-5-mini	base	on	D3	3/3	3/3	0/3	3/3	0/3
S-4	gpt-5-mini	base	off	D3	3/3	3/3	0/3	3/3	0/3
S-5	sonnet	base	on	D4	2/3	3/3	3/3	0/3	0/3
S-5	sonnet	base	off	D4	2/3	3/3	2/3	3/3	2/3
S-5	haiku	base	on	D4	1/3	2/3	2/3	1/3	0/3
S-5	haiku	base	off	D4	1/3	1/3	2/3	3/3	1/3
S-5	gpt-5-mini	base	on	D4	2/3	3/3	2/3	0/3	0/3
S-5	gpt-5-mini	base	off	D4	1/3	3/3	1/3	3/3	1/3
S-6	sonnet	base	on	D1	0/3	3/3	0/3	3/3	0/3
S-6	haiku	base	on	D1	0/3	3/3	0/3	3/3	0/3
S-6	gpt-5-mini	base	on	D1	3/3	3/3	3/3	3/3	3/3

a one-line prompt addendum ("before declaring success, run an in-cluster reachability probe and report the result"). S-1 baseline already fails outcome (0/3 across all three models on the 9 baseline-on runs), so S-1 constrained does not test the addendum's effect on V_{depth}. S-4 does: Sonnet baseline $V_{\text{depth}} = 0/3 \rightarrow$ constrained $3/3$ (full repair, conjunction $3/3$); Haiku baseline $V_{\text{depth}} = 0/3 \rightarrow$ constrained $1/3$ (partial, conjunction $1/3$). The asymmetry is the point: the prompt intervention works for the more capable model and is unreliable for the less capable one, while a harness-side post-trajectory probe runs whether the agent attempted it or not. The recommendation in §3 — evaluator-side post-trajectory probes (or tool-emitted probe events the harness records) over prompt addenda — follows from this asymmetry, not from the addendum failing universally.

Re-execution and infrastructure events. The 84-trajectory grid uses a rebuilt harness with strict D3 probe recording, per-run `kind` cluster reset, sequential execution, and no parallel contamination.

Table 10: S-7 per-cell and per-run outcomes. The known-good sequence is hardcoded and non-LLM; it is included to show that the scenario and verifier can be satisfied. The three LLM rows use the same baseline prompt and probe-off mode as the S-7 diagnostic experiment in §4.

Runner	Runs passing Conj.	Outcome	V_{temporal}	V_{depth}	V_{probe}	Conj.	Mean dur.	Notes
Known-good sequence	1,2,3	3/3	3/3	3/3	3/3	3/3	0.7s	Deletes offending NetworkPolicy; verifies from `atlas-probe`.
Sonnet 4.5	3	3/3	3/3	1/3	3/3	1/3	93s	Two runs leave cross-namespace reachability broken.
Haiku 4.5	2	3/3	3/3	1/3	3/3	1/3	131s	Repair choice varies across runs.
GPT-5-mini	—	3/3	3/3	0/3	3/3	0/3	300s	All three runs hit the 300s budget.

Table 11: Per-model headline on the main slice (n=54, 18 per model). All three model families show the same scenario-class pattern: every hidden failure (outcome pass but conjunction fail) lands on S-4 or S-5; none on the D1-check class (S-1, S-2, S-3, S-6).

Model	n	Outcome	V_{temporal}	V_{depth}	V_{probe}	Conj.
Sonnet 4.5	18	11/18 (61 %)	18/18	9/18	15/18	6/18 (33 %)
Haiku 4.5	18	10/18 (56 %)	17/18	8/18	16/18	6/18 (33 %)
GPT-5-mini	18	15/18 (83 %)	18/18	12/18	15/18	10/18 (56 %)
All	**54**	**36/54 (67 %)**	**53/54**	**29/54**	**46/54**	**22/54 (41 %)**

Two Docker Desktop daemon crashes occurred during the ∼9.5 h matrix run; both were absorbed by an automatic restart loop in `run_matrix.sh` and the affected runs were retried after the daemon returned. One $S{-}6$ Haiku run was orphaned by the second crash, scored manually with the same `score.py`, and re-aggregated; the manual recovery is logged in the per-run trajectory record. No run was scored differently from the rest.

C S-8 Construct-Validity Scenario

This appendix gives the fault recipe, oracle recipes, and per-cell agent-matrix data for the S-8 scenario used in Finding 4 (§4). The released code identifies S-8 as `s9`.

Fault recipe. S-8 creates a 20-replica `payment-api` Deployment with every pod pinned to a single worker via `nodeName` and a production-realistic 15 s `preStop` drain hook. All pods are Ready post-inject; cluster availability is ≈ 1.0. The fault is *concentration risk*, not active damage. The drain hook is load-bearing: with a 15 s preStop, a *Recreate*-style remediation produces a sustained ∼15 s window in which all 20 pods are simultaneously `Ready=False` — long enough for the 5 s state-capture cadence to record several below-threshold ticks. A naive instantaneous rolling restart would only generate a ∼3 s dip that 5 s sampling could miss.

Listing 3: S-8 fault recipe: 20-replica Deployment pinned to a single worker with a 15 s preStop drain hook.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-api
  namespace: default
spec:
  replicas: 20
  selector: { matchLabels: { app: payment-api } }
  template:
    metadata: { labels: { app: payment-api } }
    spec:
      nodeName: atlas-v11f-worker3 # single-node concentration
      terminationGracePeriodSeconds: 30
      containers:
      - name: nginx
        image: local/nginx:1.27
        lifecycle:
          preStop:
            exec: { command: ["sh","-c","sleep 15"] }
```

Table 12: S-5 probe-on/off contrast on the three model families (Property-3 evidence; each row is n=3 per arm, paired by model). Because D4 records critical-pod health rather than traffic-control removal, this table is a probe-accounting contrast. The probe is the only manipulated variable; the same agent runs the same scenario on both arms with the same long-lived-pod `kubectl exec` probe harness (`wget -timeout=2` into the protected service's ClusterIP from a busybox pod, so slow pod-spawn latency cannot account for the difference). The probe-off arm's V_{probe} column is vacuous $3/3$ by construction (no probes to attribute), so the substantive column is the conjunction. The V_{probe} aggregate (probe-on $1/9$ vs. probe-off $9/9$) gives Fisher exact 2-sided p=4.1×10^{-4}; the conjunction aggregate (probe-on $0/9$ vs. probe-off $4/9$) gives Fisher exact 2-sided p=0.082, with the same direction in every model family.

Model	V_{probe} on	V_{probe} off	Conj. on	Conj. off
Sonnet 4.5	0/3	3/3	0/3	2/3
Haiku 4.5	1/3	3/3	0/3	1/3
GPT-5-mini	0/3	3/3	0/3	1/3
Aggregate	**1/9**	**9/9**	**0/9**	**4/9**

Oracle recipes (construct-validity controls). Both oracles remove the `nodeName` pin; they differ only in the Deployment `strategy` field. After the rollout, both `sleep 20 s` so the state-capture timeline records sufficient post-recovery ticks for V_{temporal} scoring to be statistically meaningful (this is a harness-side concern, not part of the remediation itself).

The `ideal-gentle` oracle leaves the default *RollingUpdate* strategy (`maxUnavailable` 25%, so at most 5 of 20 pods can be down at once $\rightarrow\sim$7% cluster-wide unavailability):

Listing 4: Gentle oracle (RollingUpdate): removes the node pin without service interruption.

```
kubectl patch deployment payment-api --type=json \
  -p='[{"op":"remove","path":"/spec/template/spec/nodeName"}]'
kubectl rollout status deployment/payment-api --timeout=180s
sleep 20
```

The `ideal-aggressive` oracle switches the strategy to *Recreate*, which terminates every existing pod before creating replacements ($20/78 \approx 26\%$ unavailability for the duration of the drain):

Listing 5: Aggressive oracle (Recreate): all 20 pods terminate before replacements start.

```
kubectl patch deployment payment-api --type=merge \
  -p='{"spec":{"template":{"spec":{"nodeName":null}},
␣␣␣␣␣␣␣"strategy":{"type":"Recreate","rollingUpdate":null}}}'
kubectl rollout status deployment/payment-api --timeout=180s
sleep 20
```

Both oracles end with payment-api healthy and spread $5/5/5/5$ across the four workers. The minimum sample-tick cluster-wide availability is $\approx$ 0.94 under `ideal-gentle` and 0.744 under `ideal-aggressive`; the latter is well below V_{temporal}'s 0.85 threshold and is the construct-validity proof point reported in Tab. 2.

Per-run agent-matrix data (n=15). The agent matrix uses the symptom-only prompt from Tab. 5 (the constrained-prompt addendum lifted from S-1 is also supplied as the scenario `CONSTRAINT`, mirroring the existing harness contract). Strict isolation is enabled for S-8. The cluster is reset between every run. Tab. 13 reports per-run outcomes.

Three patterns are worth flagging. (i) Three of five Sonnet runs restored the topology spread but left some pods `Pending` (over-constrained `topologySpreadConstraints` with `maxSkew=1` and `whenUnsatisfiable: DoNotSchedule`), which trips V_{depth} (D1) and V_{outcome} together. Sonnet run 4 additionally drops V_{temporal} (0.839) during this rescheduling churn. (ii) No model in the matrix organically chose the *Recreate*-style restart that `ideal-aggressive` performs; the V_{temporal} sole-driver condition is only observed in the scripted oracle control. (iii) Two of five GPT-5-mini runs satisfied every verifier in the suite while leaving every pod on `worker3` — the agent acted enough

Table 13: S-8 per-run outcomes for the 15-run agent matrix. "Spread" counts the number of worker nodes hosting at least one `payment-api` pod at the final state (the alert is only satisfied if this is ≥ 2). "MinAvail" is the minimum sample-tick cluster-wide availability over the trajectory; only recorded when below the $0.85\ V_{\text{temporal}}$ threshold or otherwise noteworthy. The `ideal` rows are the scripted oracle controls.

Model	Run	V_{outcome}	V_{temporal}	V_{depth}	V_{probe}	Spread	MinAvail
`ideal-gentle`	1	✓	✓	✓	✓	4	—
`ideal-aggressive`	1	✓	×	✓	✓	4	0.744
Sonnet 4.5	1	✓	✓	✓	✓	4	—
Sonnet 4.5	2	×	✓	×	✓	5	—
Sonnet 4.5	3	×	✓	×	✓	5	—
Sonnet 4.5	4	×	×	×	✓	5	0.839
Sonnet 4.5	5	✓	✓	✓	✓	4	—
Haiku 4.5	1	✓	✓	✓	✓	3	—
Haiku 4.5	2	✓	✓	✓	✓	4	—
Haiku 4.5	3	✓	✓	✓	✓	4	—
Haiku 4.5	4	✓	✓	✓	✓	4	—
Haiku 4.5	5	✓	✓	✓	✓	3	—
GPT-5-mini	1	✓	✓	✓	✓	4	—
GPT-5-mini	2	✓	✓	✓	✓	4	—
GPT-5-mini	3	✓	✓	✓	✓	4	—
GPT-5-mini	4	✓	✓	✓	✓	1	—
GPT-5-mini	5	✓	✓	✓	✓	1	—

to keep readiness signals green but did not actually fix the distribution. This is a verifier-suite completeness gap discussed in §7, distinct from the V_{temporal} discrimination claim.

D Existing Benchmarks vs. the Three Properties

Benchmark	Domain	P1	P2	P3
SWE-Bench [Jimenez et al., 2024]	Code patches	n/a	n/a	n/a
WebArena [Zhou et al., 2024]	Simulated web (reset)	n/a	n/a	n/a
AgentBench [Liu et al., 2024]	Multi-env, simulated	n/a	n/a	n/a
τ-bench [Yao et al., 2024]	DB state, post-conv	partial	×	×
τ^2-bench [Barres et al., 2025]	Dual-control sim	partial	×	partial
AIOpsLab [Chen et al., 2025a]	Cloud incidents	×	×	×
ITBench [Jha et al., 2025]	SRE/CISO/FinOps	×	×	×
PRMs [Lightman et al., 2024]	LM reasoning trace	✓ trace	n/a	n/a
Shielded RL [Alshiekh et al., 2018]	Safe-RL	✓ learn	n/a	n/a
Near-Miss [Rabinovich et al., 2026]	Customer-service tools	✓	✓	n/a

Note on AIOpsLab and ITBench. The × marks indicate absence of the three verifier-construction predicates as defined in this paper, not absence of any temporal measurement. AIOpsLab reports Time-to-Detect, Time-to-Mitigate, and a final-state "general system state" check; ITBench reports Time-to-Diagnosis, Time-to-Repair, and % Resolved. These are recovery-latency stopwatches on per-task resolution criteria, not trajectory-integral predicates (V_{temporal}), fault-aligned predicates (V_{depth}), or probe-attribution predicates (V_{probe}). The framework here is complementary, not in opposition: it adds verifier-construction requirements that survive when stopwatch metrics are silent.

E Why Not Dynamic Depth Selection (P2 Rationale)

A natural alternative to P2's in-advance fixing of observation depth is letting the verifier choose its depth *after* the trajectory — inspecting deeper only when shallow checks pass — on the intuition that this saves observation cost. We reject it for two reasons. **First**, post-hoc depth selection can induce selection bias: a verifier free to pick the depth at which it grades can always pick one at which a given run passes (or fails). The score then reports the verifier's after-the-fact choice, not the

agent's behavior — the failure mode hand-curated benchmarks already exhibit when graders relax the bar on systems that "look healthy enough." **Second**, the cost premise is false: unobserved states are gone. Network reachability not sampled at s_i cannot be reconstructed from s_n. P2 therefore requires advance *instrumentation*, not single-layer grading; the permitted form of dynamism is layered instrumentation with a fixed-in-advance predicate — record at every depth, fix the predicate beforehand, then let the predicate range across layers. Our D1–D4 operationalization is exactly this: instrumentation is layered and recorded for every tick, and the committed object is which conjunction of layer-checks defines success for the scenario.

Why a deeper static outcome verifier does not close the gap. A natural skeptic asks: what if the standard outcome verifier were simply made deeper — pod-ready and endpoints-populated and a single end-of-trajectory HTTP probe? No *single* static depth dominates: S-1/S-2/S-3/S-6 require D1, S-4 requires D3, and S-5 requires D4. A D3-only verifier closes most of the gap on S-4 (where D3 was required) but produces vacuous fails on S-1/S-2/S-3/S-5/S-6 (where no protected HTTP service exists or the relevant signal is control-plane component health, not reachability). P2's fault-aligned observation, fixed in advance, captures exactly this: the verifier is layered (record everything), but the required check is fixed per scenario, which is what allows different scenarios to be graded at the depth their fault class actually requires.

F Threshold-Sensitivity Sweeps

Design choices and what they cost. Three thresholds shape the predicates and are pre-committed in the scenario YAML rather than tuned to results. The *state-capture cadence* is 5 s: small enough that a typical Kubernetes reconcile loop (10–30 s) cannot complete silently between captures, large enough that JSONL log size remains manageable for hour-long runs. Finer cadence trades log size for finer P1 integration; the sweeps below hold the per-class hidden-failure pattern across $\pm 2\times$ cadence. The *P3 attribution window* is ± 10 s around each tagged probe: wide enough to catch the typical kubelet/scheduler reaction interval, narrow enough that unrelated background flapping is not falsely attributed. The *P3 stall threshold* is 5 s, slightly larger than the 3 s D3 in-cluster `wget` HTTP timeout in App. A; we score a stalled probe as a V_{probe} contribution rather than an outcome success because a probe that times out is observable cost paid by the system regardless of whether endpoints later populate. Each is a *commitment* in the P2 sense — fixed in the scenario spec before any agent runs — and each is a single named YAML field, so replacing one is a one-line config edit.

The scenario-class hidden-failure pattern depends on four hyperparameters in `compute_properties.py`. We sweep each one over the existing logs (deterministic re-scoring; no new runs) and report whether the qualitative claim — the hidden-failure contingency stays at Fisher exact $p < 10^{-7}$ with all hidden cases on the deeper-check class — survives.

V_{temporal} availability threshold. Swept over $\{0.80, 0.85, 0.90, 0.95\}$. Every cell of Tab. 8 is identical at all four settings (36/54 outcome pass, 22/54 conjunction pass, 14/36 hidden, 0/22 D1-check, 14/14 deeper-check), so the table is omitted; the threshold has no effect on any reported quantity in the swept range because per-tick workload availability in the recorded trajectories is essentially bimodal (clean or broken), not borderline.

Table 14: V_{probe} probe-stall threshold sweep on the main slice (n=54). Lower thresholds tighten the screen and would inflate the gap; the per-class concentration is preserved across the practical range.

V_{probe} stall (s)	Outcome pass	Conj. pass	Hidden / pass	D1-check	Deeper-check
3	36/54	21/54	15/36	0/22	14/14
5 (used)	36/54	22/54	14/36	0/22	14/14
10	36/54	23/54	13/36	0/22	14/14
15	36/54	24/54	12/36	0/22	13/14

Across the full Cartesian product of the four hyperparameter sweeps ($4\times4\times3\times3 = 144$ settings; the fourth is the sampling-cadence sweep), the hidden-failure rate stays in $[12/36, 16/36]$ and the per-class contingency stays at Fisher exact $p < 10^{-7}$ in every setting; the scenario-class concentration is not a knife-edge result of the chosen thresholds.

Table 15: V_{probe} attribution-window sweep on the main slice (n=54). Wider windows attribute more transitions to probes; the per-class contingency is robust over reasonable window choices.

Window (s)	Outcome pass	Conj. pass	Hidden / pass	D1-check	Deeper-check
$\pm$5	36/54	22/54	14/36	0/22	14/14
$\pm$10 (used)	36/54	22/54	14/36	0/22	14/14
$\pm$20	36/54	21/54	15/36	0/22	14/14

G Reference Verifier Implementation

The released script retains legacy function names in this excerpt: `score_v_p1`, `score_v_p2`, and `score_v_p3` correspond to V_{temporal}, V_{depth}, and V_{probe} respectively. The `score_v_p2` branch implements the main-text committed-depth rule: D3 requires the evaluator's recorded final HTTP probe, not a D2 endpoint proxy.

Listing 6: Reference implementation of the three verifier predicates (excerpt).

```
COMMITTED_DEPTH = {"S-1":"v1","S-2":"v1","S-3":"v1",
                   "S-4":"v3","S-5":"v4","S-6":"v1"}
V_OUTCOME_AVAIL_THRESHOLD = 0.95
V_P1_AVAIL_THRESHOLD = 0.85
V_P3_PROBE_STALL_MS = 5000

def score_v_outcome(state_final, scenario):
    avail = state_final["v1"]["availability"]
    crit = state_final["v4"]["critical_failing"]
    return avail >= V_OUTCOME_AVAIL_THRESHOLD and not crit

def score_v_p1(timeline, _scenario):
    for rec in timeline:
        if rec["v1"]["availability"] < V_P1_AVAIL_THRESHOLD: return False
        if rec["v4"]["critical_failing"]: return False
    return True

def score_v_p2(state_final, scenario):
    """V_P2: the *committed* depth verifier passes on the final state.
    Invariant: when committed_depth is D3, we run an actual final
    reachability probe; we do NOT proxy from D2 endpoint state."""
    depth = COMMITTED_DEPTH[scenario]
    if depth == "v1":
        return state_final["v1"]["availability"] >= V_OUTCOME_AVAIL_THRESHOLD
    if depth == "v2":
        svc = state_final["v2"][scenario_protected_svc(scenario)]
        return svc["ready_addrs"] > 0
    if depth == "v3":
        # Must be the evaluator's recorded final HTTP probe.
        # Missing/null is a failure; endpoint readiness is not a proxy.
        return state_final["v3"]["reachable"] is True
    if depth == "v4":
        return not state_final["v4"]["critical_failing"]

def score_v_p3(timeline, state_final):
    # Window: timeline[idx-2:idx] is the 10s prior to the probe;
    # timeline[idx+1:idx+3] is the 10s after. The pair implements
    # the body's symmetric +/-10s attribution window.
    for idx, rec in enumerate(timeline):
        if rec["probe_id"] == "noprobe": continue
        if (rec.get("v3",{}).get("latency_ms") or 0) > V_P3_PROBE_STALL_MS:
            return False
        before = {c["name"] for r in timeline[max(0,idx-2):idx]
                            for c in r["v4"]["critical_failing"]}
        after = {c["name"] for r in timeline[idx+1:idx+3]
                           for c in r["v4"]["critical_failing"]}
        if after - before: return False
    if state_final.get("probe_id") != "noprobe":
```

```
        lat = state_final.get("v3",{}).get("latency_ms")
        if lat is not None and lat > V_P3_PROBE_STALL_MS:
            return False
    return True
```

H OpenEvolve Discovery Details

Evaluator and operator. Each OpenEvolve candidate is a Python `fault_composition()` returning a list of `kubectl-apply`, `kubectl-cmd`, and `docker-exec` operations. The evaluator resets the cluster, applies the candidate, runs a scripted competent operator, and scores the scoped committed-depth predicate outcome pass $\wedge\neg V_{\text{depth}}$. Fitness=1.0 means the operator declared success while the committed-depth check failed; fitness=0.1 means the operator detected the surface fault; otherwise fitness=0. The evaluator uses claude-haiku-4.5 via litellm at temperature 1.0, five RNG seeds per prompt variant, population size 1, one offspring per iteration, and max iteration budget 10.

The evaluator's repair logic is what defines "fixed shallowly," which is what counts as fitness=1.0. Table 16 documents its repair vocabulary, stop conditions, and known blind spots. The operator stops as soon as the surface symptom resolves; its blind spots are exactly the case the committed-depth predicate discriminates.

Table 16: Scripted competent-operator vocabulary used inside the OpenEvolve evaluator. Fitness=1.0 is defined relative to this specification.

Symptom	Operator action	Stops when	Known blind spots
endpoint empty	`kubectl patch svc` (correct selector)	endpoints populate	NetworkPolicy / firewall left in place; no end-to-end probe
pod CrashLoop	`kubectl rollout restart` or fix bad ConfigMap ref	pods Ready	control-plane latency / OOM pressure left in place
NoExecute taint	`kubectl taint -untaint`	pods reschedule and become Ready	taint may have been a deliberate test of PDB/anti-affinity
bad image tag	`kubectl set image` (last-known-good tag)	rollout completes	no probe of the deployed code path
control-plane unreachable	`kubectl get -raw /healthz`; if non-200, restart apiserver pod	`/healthz` returns 200	artificial network latency below the implicit health-check timeout passes silently

H.1 Prompt-information sweep prompts and outcomes

The seed program (`seed_clean.py`) creates a healthy `auth-service` Deployment and matching Service and contains no Kubernetes-specific tokens in its docstring. We vary only the prompt content: **vocab-full** names fault categories the operator does not inspect (NetworkPolicy, PodDisruptionBudget, control-plane network delay via `tc-netem`, image set) and lists the operator blind spots; **vocab-partial** lists only operation kinds and operator behavior, without naming Kubernetes resource types; and **vocab-minimal** lists only operation kinds plus the abstract goal, "compose operations such that the operator declares success but a deeper property remains broken."

Prompt variant	Seeds reaching fitness=1.0	Median first-hit iter.	Winner diversity
vocab-minimal	5/5	1	4 fault families
vocab-partial	4/5	1	3 fault families
vocab-full	5/5	1	2 fault families
All	**14/15**	**1**	–

Under vocab-minimal, winners include `wrong-targetPort`, `template-label-mismatch`, `broken-readinessProbe`, and `broken-configmap` compositions alongside NetworkPolicy.

vocab-full produces NetworkPolicy in 5/5 winners, while vocab-minimal uses NetworkPolicy in 3/5 and spreads across more fault families. We do not interpret this as independence from pretraining; it shows only that fault-building-block names need not be enumerated in the prompt for verifier-guided mutation to produce typed fault compositions.

H.2 Random-search protocol

The random-search baseline samples 50 fault compositions of 1–3 fault building blocks uniformly with replacement from the same six-building-block library: deny-all-ingress NetworkPolicy, tight PodDisruptionBudget, broken image, artificial control-plane network delay (`tc-netem 6000ms` on the container's `eth0`), wrong selector, and no-op. Each composition is prefixed with the same healthy `auth-service` precondition; `seed=0xBEEF`. Per-trial outcomes:

Verdict	Count (n=50)
Outcome pass, Depth fail (hidden failure; fitness=1.0)	20
Outcome pass, Depth pass (benign under committed depth)	12
Outcome fail, Depth fail (operator detects / outcome fails)	18

Thus the outcome check alone admits 32/50 candidates as healthy, but the committed-depth predicate separates 20 hidden-failure candidates from 12 benign ones. Common building blocks in the 20 fitness=1.0 hits include wrong selector (13/20), deny-all-ingress NetworkPolicy (11/20), tight PodDisruptionBudget (9/20), and broken image (5/20).

H.3 Precondition predicate and verifier-audit demonstration

The precondition predicate is evaluated after the candidate operations are applied but before the scripted operator runs. For the S-4-style service-reachability discovery scenario it requires:

1. `default/auth-service` Deployment exists with `spec.replicas` ≥ 1.
2. `default/auth-service` Service exists.
3. Service selector keys are present in the Deployment's pod-template labels with matching values.

Compositions that fail the precondition predicate receive `precondition_violated=True` and fitness=0 regardless of downstream outcome and committed-depth results. The reference implementation accepts a legitimate hidden-depth fault (the vocab-minimal seed-11 winner) with fitness=1.0 and `precondition_violated=False`; it rejects the CrashLoop bypass with fitness=0 and reason "`auth-service Deployment does not exist`." The same audit pass also motivated stronger reset checks for host-level mutations and static-pod-manifest corruption.

H.4 Discovery evaluator methodology

The discovery-loop evaluator used for §5 makes three operational choices explicit. First, the final state is captured with an active D3 reachability probe so V_{depth} has a recorded committed-depth observation rather than an endpoint-count proxy. Second, reset additionally cleans PodDisruptionBudgets, ResourceQuotas, LimitRanges, webhook configurations, host-level mutations on the control-plane container, and static-pod-manifest changes. Third, the loop optimizes outcome pass $\wedge\neg V_{\text{depth}}$ rather than the full four-verifier conjunction; the controlled S-5 probe-on/off contrast in §4 is the evidence for V_{probe}. Source artifacts are in `experiments/openevolve/vocab_ablation/`: evaluator scripts, three prompt configs, best programs for each seed, random-search results, and `precond_demo.log`.

I Reproducibility

Listing 7: Reproducing the full experiment matrix.

```
kind create cluster --name atlas-v11 \
  --config experiments/scripts/atlas-cluster.yaml
```

```
# 2. Run the 84-trajectory v11_full matrix (sequential, ~9.5 h)
bash experiments/v11_full/scripts/run_matrix.sh

# 3. Run the S-7 targeted diagnostic extension (3 runs per row;
# the released code identifier is "s8")
for run in 1 2 3; do
  for model in sonnet haiku gpt-5-mini ideal; do
    bash experiments/v11_full/scripts/run_one.sh \
      --scenario s8 --model "$model" --prompt-variant baseline \
      --probe-mode off --run "$run" \
      --results-base "$PWD/experiments/results/v11_full_s8_pilot"
  done
done

# 4. (Optional) start the watchdog so any orphan/hang/Docker crash
# is caught and the matrix can recover automatically
bash experiments/v11_full/scripts/watchdog.sh &

# 5. Aggregate results and statistical tests
python3 experiments/v11_full/scripts/aggregate.py \
       experiments/results/v11_full
python3 experiments/v11_full/scripts/extract_paper_stats.py

# 6. Run the OpenEvolve prompt-information discovery sweep
bash experiments/openevolve/vocab_ablation/launch_all_vocab_ablation.sh
python3 experiments/openevolve/vocab_ablation/random_search_v17.py
python3 experiments/openevolve/vocab_ablation/live_summary.py
```