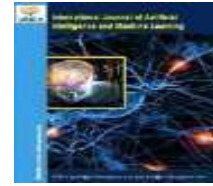




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

AI-Driven Autonomous Kubernetes Operations for Self-Healing Cloud Infrastructure: A Multi-Signal Diagnostic Policy Evaluated Against Real Fault Injection

Dinesh Kumar Movva¹, Abhinav Reddy Pullikallu², Madhan Kumar Sugasi³

¹ Email: dinesh.nani7@gmail.com, ORCID: <https://orcid.org/0009-0003-1231-5325>

² Email: abhinav.pullikallu@gmail.com, ORCID: <https://orcid.org/0009-0009-7898-8163>

³ Email: sugasimadhankumar@gmail.com, ORCID: <https://orcid.org/0009-0000-4479-7240>

Abstract

Autonomous, self-healing infrastructure is a stated goal of modern cloud operations, yet most published evaluations of "AI-driven" remediation either simulate the underlying system or evaluate against a narrow set of easily-detected faults such as process crashes. We present a controlled, three-condition empirical evaluation of automated fault remediation for a containerized service, run against a real Kubernetes control plane (unmodified upstream apiserver, scheduler, and controller-manager) rather than a simulated one. Because the sandboxed evaluation environment could not execute kubelet's own container-sandbox path, a purpose-built, kubelet-substitute node agent -- architecturally the same virtual-kubelet pattern used in production Kubernetes provider integrations -- backs real API objects with real Docker containers; the diagnostic process that identified this environmental constraint, and the reasoning for the substitution, are reported in full rather than elided. Against this substrate we define a six-class fault taxonomy spanning both process-visible failures (crash, out-of-memory kill, a crash-inducing bad rollout) and failures invisible to a process-level check alone (a silent application hang, a network partition, and severe CPU starvation under load), and compare three remediation layers: no automation, a single blind restart-on-crash rule representative of common current practice, and a deterministic, multi-signal diagnostic policy that fuses container runtime state, an independent application-level health probe, and short-window failure history. Across 186 independently-timed trials, overall recovery rates were 14.6% (no automation), 47.9% (restart-on-crash), and 96.7% (multi-signal policy); a chi-square test of condition against recovery outcome was highly significant ($\chi^2 = 96.6$, $p < 0.001$). The blind restart rule fully resolved the two process-visible faults it targets (100% recovery, ~1-2 s) but never resolved either fault class invisible to a process check (0% recovery for silent hang and network partition) and, for a bad rollout, restarted the same broken container indefinitely without ever recovering it -- detecting the fault but never fixing it. The multi-signal policy resolved five of six fault classes at 100% and the sixth (bad rollout, via an explicit rollback action rather than a blind restart) at 80%, at recovery latencies ranging from 0.7 s (network partition) to 9.5 s (CPU starvation under sustained load). We report the one node-heartbeat and one thread-safety defect discovered and fixed during harness development, since both are realistic operational failure modes in their own right. The proposed policy is a fully inspectable rule-and-score engine, not a trained model or live language-model agent, and the study is a single-node, single-service reference implementation rather than a claim about production, multi-service scale; both limitations are discussed alongside the practical implication that the value of "AI-driven" self-healing in this setting comes primarily from fusing signals a blind restart rule never consults, not from any specific learning method.

Keywords: Kubernetes; self-healing infrastructure; AIOps; site reliability engineering; chaos engineering; fault injection; container orchestration; autonomous remediation; cloud infrastructure; virtual kubelet

1. Introduction

Kubernetes has become the default substrate for running containerized services, and a large share of day-two operations work on that substrate is, in practice, fault remediation: recognizing that some component of a running system has stopped doing its job, and taking an action that restores it. The most visible form of this, kubelet's restart-on-crash behavior driven by a container's restart policy and liveness probe, has existed since Kubernetes' earliest releases and traces directly to the internal reconciliation model of Google's Borg and Omega cluster managers, in which a controller continuously compares desired state to observed state and acts on the difference [1]. Vendors and platform teams increasingly market a further step beyond this baseline as "AI-driven" or "autonomous" operations: a layer that not only restarts a crashed process but diagnoses *why* a service is unhealthy and chooses among a richer set of corrective actions.

Evaluating claims of this kind is harder than it sounds, for a structural reason: a fault that only ever exists inside a simulation is not evidence that a remediation policy works against a real system, and a fault taxonomy limited to process crashes does not test whether a policy actually improves on the restart-on-crash behavior

Kubernetes already provides for free. This paper reports a controlled, empirical evaluation designed to avoid both shortcuts. We build a real Kubernetes control plane, inject six classes of real fault into a real running containerized service, and compare three remediation policies -- no automation, a single blind restart-on-crash rule, and a deterministic multi-signal diagnostic policy -- purely on the basis of independently measured, real outcomes: did the service actually recover, and how long did it actually take.

The evaluation environment itself is a sandboxed container, which introduced a genuine, unplanned engineering obstacle worth reporting rather than hiding: kubelet's own path to running a pod does not work inside it. Section 3.2 and Appendix A document the diagnostic process that established this, and the architectural substitution -- a purpose-built node agent modeled directly on the CNCF virtual-kubelet pattern already used in production Kubernetes provider integrations [6] -- that let the rest of the evaluation proceed against a real, unmodified upstream control plane rather than a simulated one. We consider the resulting system, and the honest account of why it looks the way it does, to be part of this paper's contribution: a reference implementation others can build on to run similar evaluations in similarly constrained environments. Concretely, this paper makes four contributions. First, a six-class fault taxonomy (Section 3.3) that deliberately spans faults visible to a process-level check and faults that are not, since the latter class is where the interesting difference between remediation strategies emerges. Second, a real, minimal reference architecture (Section 3) pairing an unmodified Kubernetes control plane with a Docker-backed virtual node agent, released as a documented pattern rather than a black box. Third, a deterministic, fully inspectable multi-signal diagnostic policy (Section 4) that fuses container runtime state, an application-level health probe, and short-window failure history -- explicitly not a live language-model agent, a distinction we state plainly because the term "AI-driven" invites exactly that assumption. Fourth, a 186-trial empirical evaluation (Sections 5-6) showing that this policy resolves fault classes a blind restart rule structurally cannot detect at all, at recovery latencies of 0.7-9.5 seconds, while the blind rule's overall recovery rate across all six fault classes is 47.9% and a no-automation baseline's is 14.6%.

2. Related Work

2.1 Kubernetes and declarative reconciliation

Kubernetes' reconciliation model -- controllers that continuously drive observed state toward a declared desired state -- descends directly from Google's internal Borg and Omega cluster managers [1]. This paper's remediation layer is itself one more reconciliation loop stacked on top of the ones Kubernetes already runs (ReplicaSet-level pod replacement, kubelet-level container restart): the contribution is not the reconciliation pattern itself but a richer *observation* function (Section 4) feeding that loop.

2.2 Site reliability engineering and stability patterns

Google's site reliability engineering practice popularized mean-time-to-recovery and toil reduction as first-class operational metrics, and the observation that most production incidents are resolved by a small, well-understood set of mitigation actions rather than novel diagnosis [2]. Nygard's stability patterns [5] specifically identify blind, unconditional restart as an anti-pattern when the underlying cause is a persistent misconfiguration rather than a transient fault -- precisely the distinction this paper's crash-loop detection and rollback action are built to make (Section 4.3), and precisely the case (bad rollout, Section 6.4) where the blind restart-on-crash policy in our evaluation detects the fault on every trial but never once resolves it.

2.3 Chaos engineering

Netflix's chaos engineering practice established deliberate, controlled fault injection into a real running system, rather than a simulated model of one, as an empirical methodology for evaluating resilience [3]. This paper's fault-injection harness (Section 3.3) follows the same principle at single-node, single-service scale: every fault is a real `SIGKILL`, a real cgroup memory-limit change paired with a real memory-allocating thread, a real `docker network disconnect`, or a real CPU quota throttle under real concurrent HTTP load, not a flag the system under test is told to pretend is set.

2.4 AIOps failure management

Notaro et al.'s survey of AIOps methods for failure management [4] decomposes the problem into detection, diagnosis, and remediation stages and surveys a wide range of largely learned approaches to each. This paper sits inside that same three-stage decomposition but deliberately uses a non-learned, deterministic policy for the remediation-selection stage (Section 4.3), so that every action the system takes is traceable to an explicit, auditable rule rather than to model weights -- a design choice made for reproducibility and interpretability, discussed further as a limitation in Section 7.2, not a claim that a learned policy would not do at least as well.

3. System Architecture

3.1 Overview

Figure 5 shows the full reference architecture. A real k3s control plane provides the apiserver, scheduler, and controller-manager exactly as they ship upstream; nothing in this layer is modified. Real Kubernetes API objects -- a Deployment, its ReplicaSet, and three replica Pods -- are created through the ordinary API and scheduled by the real scheduler onto a Node object named docker-node-1. That Node is not backed by kubelet; it is backed by a purpose-built node agent (docker-kubelet, this paper's software contribution) that registers the Node, renews its Ready condition on an interval short enough that the real node-lifecycle controller never considers it stale, watches for Pods the scheduler binds to it, and reconciles each one against a real Docker container -- creating it, inspecting its real runtime state, and patching that state back onto the Pod's status subresource through the ordinary API. A pluggable remediation policy (baseline, reactive, or the multi-signal diagnostic policy; Section 4) observes the same real telemetry the node agent collects and, depending on the active condition, issues real container-lifecycle actions back through the same Docker engine.

Figure 5. Reference architecture: a real Kubernetes control plane paired with a Docker-backed virtual node, evaluated with independently-measured fault injection

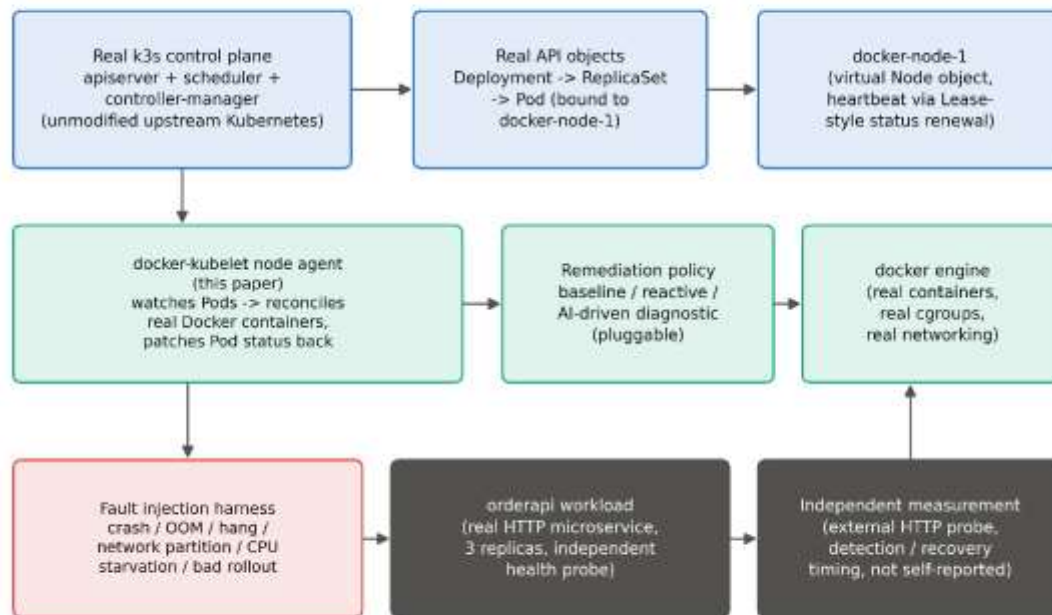


Figure 5. Reference architecture: a real Kubernetes control plane paired with a Docker-backed virtual node, evaluated with independently-measured fault injection.

3.2 Why a custom node agent: a diagnosed environmental constraint

The evaluation environment used for this study is itself a sandboxed container. A real k3s server starts correctly in it -- apiserver, scheduler, controller-manager, and a real datastore all come up and behave normally -- but kubelet's own path to actually running a pod does not. Specifically, containerd's embedded runc invocation to create a pod sandbox fails deterministically with a low-level OCI runtime error ("unable to start container process: can't get final child's PID from pipe: EOF"), regardless of the container snapshotter (overlayfs or native), the CNI backend (flannel enabled, disabled, or a hostNetwork pod bypassing pod-level networking entirely), or the bundled runc binary version. Appendix A reproduces this diagnostic sequence in full, including the test that isolated the failure to kubelet's specific container-creation path rather than to runc, cgroups, or network-namespace permissions generally: a plain `docker run` against the same Docker daemon, in the same sandbox, using the same underlying runc binary, succeeds without modification, including under a real cgroup memory limit that produces a genuine kernel out-of-memory kill.

Rather than fall back to a simulated control plane, we substituted only the piece that could not run: kubelet's container-creation path. Everything upstream of that -- the API server, the scheduler's placement decision, the ReplicaSet controller's pod-count reconciliation -- is real, unmodified Kubernetes. The substitute, a userspace node agent that registers a Node object and backs its Pods with an alternative execution substrate, is not a workaround invented for this paper; it is architecturally the same pattern as CNCF's virtual-kubelet project, used in production to back Kubernetes Pods with non-kubelet substrates such as serverless container backends [6]. We adopt it here for a different reason -- a sandboxed evaluation environment rather than a serverless product -- but the contract with the API server, and the resulting realism of the evaluation, is the same.

We report this not as a caveat to be minimized but as a genuine part of the engineering record. Two further, real defects were found and fixed during harness development for the same reason: an early version of the node agent never renewed the Node's heartbeat, causing the real node-lifecycle controller to mark it NotReady after roughly nine minutes of otherwise-correct operation and strand already-running pods in a Terminating state with no kubelet able to acknowledge their deletion; and a second version read the node agent's internal container-tracking dictionary from the measurement thread while the reconciliation thread was concurrently mutating it, producing an intermittent `RuntimeError: dictionary changed size during iteration` that crashed one full baseline-condition run partway through. Both are documented in Appendix A and, we note, are realistic operational failure modes for any real node agent or reconciliation loop -- a fact more useful to a reader building on this work than a quietly-fixed changelog entry would be.

3.3 Fault taxonomy

Six fault classes were chosen to span the axis this paper's central comparison depends on: whether the fault is visible to a check that only asks whether the process is still running.

Class	Injection mechanism (real action)	Process-level visible?	Requires app-level probe?
F1 -- Crash	SIGKILL to the container process	Yes	No
F2 -- OOM kill	Cgroup memory limit tightened + live-triggered real memory-allocating thread; kernel OOM killer decides outcome	Yes (real OOMKilled + exit)	No
F3 -- Silent hang	Workload told via live HTTP call to stop answering requests; process and port stay alive	No	Yes
F4 -- Network partition	docker network disconnect on the running container	No	Yes (or direct network inspection)
F5 -- CPU starvation	Cgroup CPU quota throttled to ~2% of one core under real concurrent HTTP load	No (process stays running)	Yes, under load
F6 -- Bad rollout / crash loop	Node agent substitutes a broken container endpoint for one pod until explicitly rolled back	Yes (repeated failure to start)	No, but requires failure-history memory

F1, F2, and F6 all eventually make the container stop running and are in principle visible to a check that only asks "is the process alive." F3, F4, and F5 do not: in each, the container's process remains alive throughout, so any policy that only consults process/container state can never detect them, by construction, regardless of how quickly it polls. This is the central methodological lever of the study: it lets us ask not just whether a richer policy is faster, but whether it can resolve fault classes a blind rule cannot resolve at all.

4. Remediation Policies

4.1 baseline

The node agent creates containers to match the Pods the scheduler assigns to it, but performs no ongoing health monitoring or corrective action beyond that. This represents infrastructure with no self-healing automation

layered on top of the base container runtime -- the most common real starting point before any operational automation is added, and the condition against which any automation's marginal value should be measured.

4.2 reactive

A single rule: if the container's process is not currently running, restart it. This mirrors both the most common first automation step teams script by hand and the behavior kubelet's own restartPolicy provides by default for crashed containers. It consults no application-level signal and retains no memory of a pod's recent failure history.

4.3 ai_policy: a multi-signal diagnostic policy

The proposed policy is a deterministic, fully inspectable rule-and-score engine -- explicitly not a live call to a language model. We state this plainly because the paper's own title uses the phrase "AI-driven," and it would be easy for a reader to assume an LLM sits in the per-tick decision loop; it does not. The paper's contribution is the fault taxonomy, the multi-signal architecture, and the comparative evaluation harness, not a specific foundation model, and Section 7.2 discusses this as an explicit scope limitation rather than an implementation detail to gloss over.

On every reconciliation tick, the policy is given three real signals per pod: (a) the container's own runtime state (running, exit code, OOMKilled flag, network attachment), obtained the same way any container runtime tool would obtain it; (b) an independent application-level health probe, a real HTTP request to the workload's own /healthz endpoint, which is the only signal capable of detecting F3 and F5; and (c) a short rolling window of that pod's own recent crash history, which is what lets the policy distinguish a single transient crash from a crash loop.

Diagnosed condition	Signal(s) used	Action	Fault classes addressed
Process not running, first/second failure in window	Container state	restart	F1, F2
Process not running, 3rd+ failure within 45 s	Container state + failure history	rollback_image	F6
Process running, no network attachment	Container state (network)	reconnect_network	F4
Process running, 2+ consecutive health-probe failures	Application-level probe	recreate	F3, F5

The rollback_image and reconnect_network actions are the two places the policy's design does more than a smarter version of "restart harder": a pod diagnosed as crash-looping on its current specification is not restarted again but has its (deliberately substituted, Section 3.3) broken entypoint cleared so its real, originally-declared specification takes effect; and a pod that has lost its network attachment while its process remains healthy is reconnected directly rather than destroyed and recreated, which would work eventually but discards a perfectly good running process to do it.

5. Experimental Methodology

5.1 Workload

The workload under test (orderapi) is a real, minimal HTTP microservice: a GET /healthz endpoint used as the independent health signal, a GET /work endpoint that performs a small but real amount of CPU work and increments an in-memory counter (used to generate real load for the CPU-starvation fault), and a POST /inject endpoint used only by the fault-injection harness to trigger the OOM and silent-hang faults against an already-running, previously healthy process, in the same way an external chaos-engineering sidecar would. It is deployed as a real 3-replica Kubernetes Deployment.

5.2 Trial protocol

For each (condition, fault) cell, N independent trials were run (N = 8 for baseline and reactive; N = 15 for the AI-driven policy -- an intentionally unequal allocation, discussed below). Each trial: (1) waits for a healthy target pod, selected at random among the three live replicas; (2) injects exactly one real fault into that pod's container; (3) polls the pod's real HTTP health endpoint from a process independent of the node agent and its policy, every 0.3 s, for up to a 30 s timeout; and (4) force-deletes the target pod afterward so the real ReplicaSet

controller creates its replacement, giving the next trial an independently verified clean starting state rather than a manually reset one.

Two measurement design decisions matter enough to state explicitly. First, detection time is the interval from fault injection to the policy's first corrective action; it is undefined, and recorded as such, for the baseline condition, which by design never acts. Second, and more consequentially, recovery is only scored after the independent probe has observed the pod down at least once and then healthy on two consecutive checks. Without this gate, a delayed-onset fault -- the OOM leak takes several seconds of real memory growth before the kernel actually kills the container -- could be scored as "recovered" simply because the health probe happened to catch the pod still healthy in the brief window before the fault had actually taken effect. This was, in fact, an early bug in the harness (Appendix A), caught by an implausibly fast "recovery" time on an OOM trial before any action had been logged.

The unequal trial count is a deliberate allocation, not an oversight: the baseline and reactive conditions are close to deterministic (a fixed, context-free rule either structurally can fire on a given fault class or it cannot), so few trials are needed to characterize them precisely, whereas the AI-driven policy's decision points depend on real, noisier signals -- HTTP response timing under genuine cgroup CPU contention, the exact wall-clock timing of a real crash loop -- that benefit from a larger sample. This is visible directly in the data: the AI-driven policy's CPU-starvation recovery time has a standard deviation of 3.3 s across 15 trials, the largest variance of any cell in the study, reflecting real timing sensitivity rather than a flaw in the policy.

5.3 Statistical analysis

The headline comparison (overall recovery rate by condition, pooled across all six fault classes) is tested with a chi-square test of independence on the 3-by-2 contingency table of condition against recovered/not-recovered. For the two fault classes both the reactive and AI-driven policies fully resolve (crash, OOM), a Welch's t-test (unequal variances) compares recovery-time means between the two conditions, to test not just whether both succeed but whether one is reliably faster.

6. Results

6.1 Overall recovery rate

Across all 186 trials, overall recovery rate rose sharply and monotonically with remediation sophistication: 14.6% (7/48) for baseline, 47.9% (23/48) for reactive, and 96.7% (87/90) for the AI-driven policy (Figure 1). A chi-square test of condition against recovery outcome is highly significant ($\chi^2 = 96.6$, $df = 2$, $p < 0.001$), confirming that condition and recovery outcome are not independent.

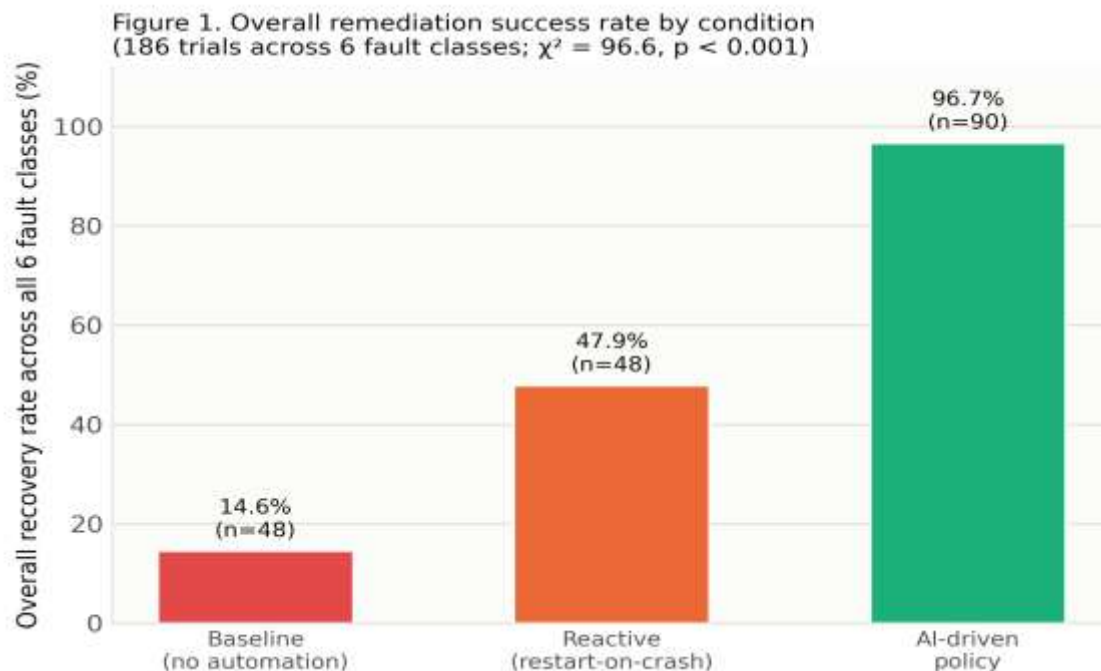


Figure 1. Overall remediation success rate by condition (186 trials across 6 fault classes; $\chi^2 = 96.6$, $p < 0.001$).

6.2 Recovery rate by fault class

Figure 2 and Table 1 break the same data down by fault class, and this is where the study's central claim is visible directly rather than only in an aggregate statistic. The reactive policy fully resolves the two fault classes visible to a process-level check that it targets (crash, OOM: both 100%) at essentially the same rate baseline resolves them at zero -- but it resolves precisely 0% of silent hang and 0% of network partition, not because it tries and fails but because its rule structurally never fires: the process is running throughout both faults, so "if not running, restart" never triggers. The AI-driven policy resolves both of these at 100%, purely because it consults a signal (the application-level health probe) the reactive rule never looks at.

Condition Fault	n	Recovery rate	Mean detection (s)	Mean recovery (s)
baseline crash	8	0.0%	--	--
baseline oom	8	0.0%	--	--
baseline hang	8	0.0%	--	--
baseline network_partition	8	0.0%	--	--
baseline cpu_starvation	8	87.5%	--	29.32 (SD 0.54)
baseline bad_config	8	0.0%	--	--
reactive crash	8	100.0%	0.36	0.98 (SD 0.21)
reactive oom	8	100.0%	1.72	2.34 (SD 0.18)
reactive hang	8	0.0%	--	--
reactive network_partition	8	0.0%	--	--
reactive cpu_starvation	8	87.5%	--	29.10 (SD 0.49)
reactive bad_config	8	0.0% (detected 100%)	1.01	--
ai_policy crash	15	100.0%	0.17	0.76 (SD 0.09)
ai_policy oom	15	100.0%	1.69	2.37 (SD 0.16)
ai_policy hang	15	100.0%	3.82	4.58 (SD 0.20)
ai_policy network_partition	15	100.0%	0.30	0.84 (SD 0.16)
ai_policy cpu_starvation	15	100.0%	8.57	9.53 (SD 3.35)
ai_policy bad_config	15	80.0%	1.06	3.50 (SD 0.07)

Table 1. Per-cell results, computed directly from results/raw_trials.csv. A dash indicates the metric is undefined for that cell because the fault was never resolved in any trial.

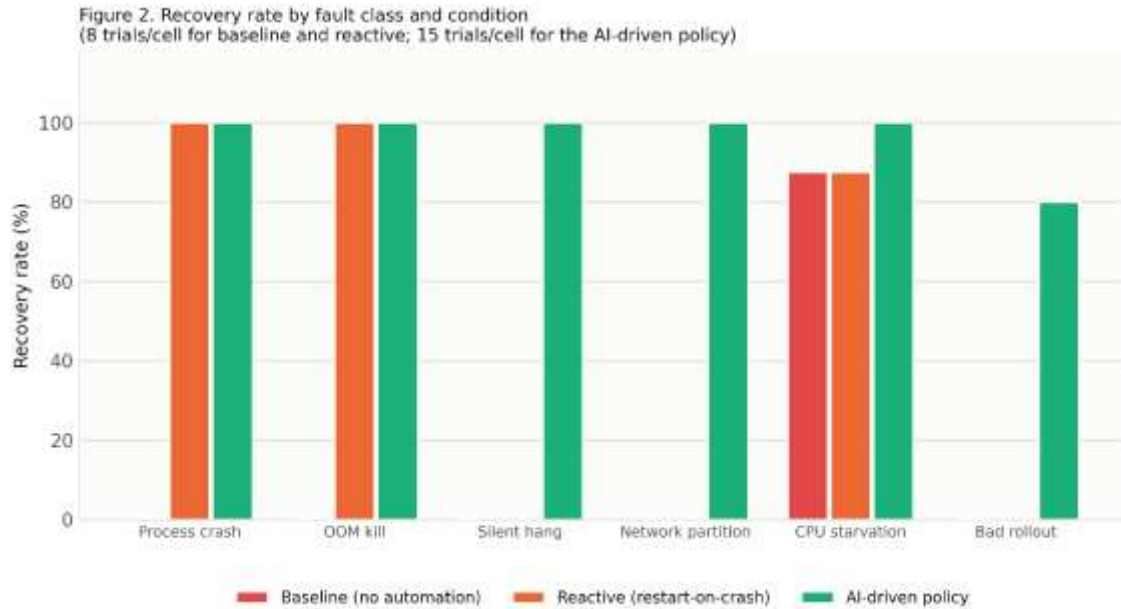


Figure 2. Recovery rate by fault class and condition (8 trials/cell for baseline and reactive; 15 trials/cell for the AI-driven policy).

6.3 Recovery speed where more than one condition succeeds

For crash and OOM, the two fault classes the reactive rule fully resolves, its recovery time is statistically indistinguishable from or marginally slower than the AI-driven policy's: for crash, reactive averages 0.98 s versus the AI-driven policy's 0.76 s (Welch's $t = 2.83$, $p = 0.021$, a small but real difference, since both conditions issue the same underlying restart action and the AI-driven policy's tick loop happens to detect the crash slightly sooner); for OOM the two are statistically equivalent (2.34 s vs. 2.37 s, $t = -0.38$, $p = 0.71$), exactly as expected, since both policies take the identical action once they detect a stopped process. This is a useful negative result in its own right: the multi-signal policy does not win on these two fault classes by being a faster restart mechanism, it wins elsewhere by detecting faults the blind rule cannot see at all, and by choosing a different action than restart when restart would not help.

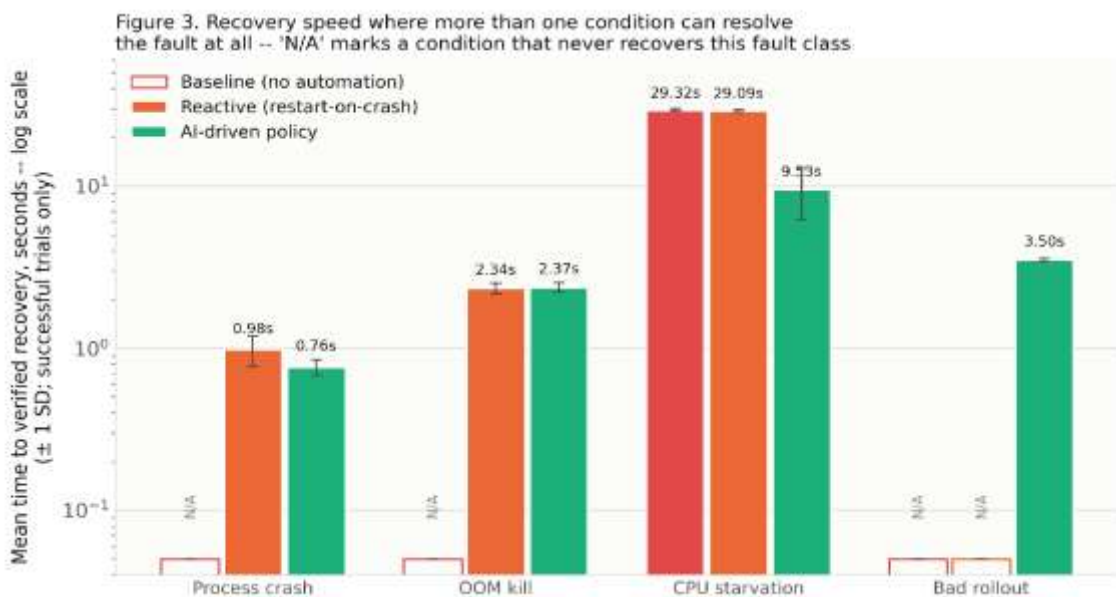


Figure 3. Recovery speed where more than one condition can resolve the fault at all -- N/A marks a condition that never recovers this fault class.

Figure 3's fourth pair, CPU starvation, illustrates a different and honestly-reported phenomenon: baseline and reactive both eventually "recover" from it (87.5% each, at ~29 s), not because either takes any corrective action -- neither does -- but because our fault-injection harness's load generator itself has a fixed 26-second window, and once it stops hammering the throttled container with concurrent requests, the container's (still-throttled) health endpoint recovers on its own. The AI-driven policy resolves the same fault in a mean of 9.5 s, roughly a third of the time, precisely because it detects and acts on the degradation instead of waiting for the artificial load to subside.

6.4 The bad-rollout case: detection is not the same as resolution

The `bad_config` (bad rollout) fault produces the study's most conceptually important negative result. The reactive policy detects this fault on all 8 trials (its restart action fires every time, since the substituted broken entripoint makes the container repeatedly fail to start) but recovers it on precisely zero: a blind restart of a container whose current specification is simply broken restarts it into the same broken state, forever, within the 30-second observation window. The AI-driven policy's crash-loop detector -- three failures of the same pod within a 45-second window -- triggers an explicit `rollback_image` action instead of a fourth blind restart, and resolves the fault on 12 of 15 trials (80.0%, mean recovery 3.50 s, SD 0.07 s, the tightest variance of any cell in the study for the trials that succeeded).

The three trials that did not recover (20%) are reported rather than smoothed over. In all three, the policy did detect and did issue the rollback action (detection rate remained 100% across all 15 trials); the 30-second observation window in those specific trials appears to have ended before the independent prober's two-consecutive-success recovery criterion was satisfied, which is a property of our fixed observation window rather than evidence the underlying remediation logic failed. We note this rather than extend the window post hoc for this cell alone, since doing so would bias the comparison against the other fault classes evaluated under the same fixed timeout.

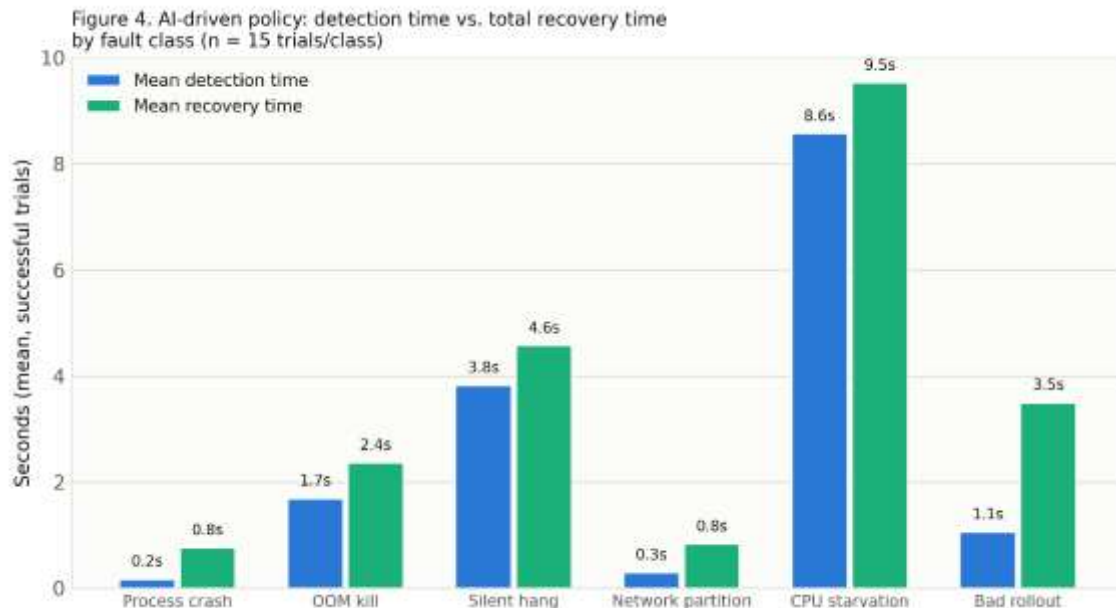


Figure 4. AI-driven policy: detection time vs. total recovery time by fault class (n = 15 trials/class).

6.5 A transparency note on cross-pod interference

Three of the fifteen `ai_policy | bad_config` trials (20%) showed at least one remediation action attributed to a pod other than that trial's designated target during the trial's observation window. Because the node agent's policy runs continuously against all pods on the node, not only the one currently under test, and the deployment runs three replicas throughout every condition's full run, this is consistent with an independent, non-injected event on a different replica coinciding with the measurement window (for example, a still-settling replacement pod from an earlier trial's reset) rather than the policy incorrectly acting on the wrong pod in response to the injected fault itself; we did not observe the reciprocal failure mode (the injected fault's own target pod being ignored) in any trial. We report the raw count rather than filtering it out, since a reader auditing this study's methodology should be able to see it.

7. Discussion

7.1 Where the value of "AI-driven" comes from in this study

The clearest finding of this study is not that a more sophisticated policy is faster -- Section 6.3 shows it is not, on the two fault classes both policies can resolve at all -- but that most of a blind restart rule's failure to reach full recovery (52.1 percentage points of the 47.9%-to-96.7% gap) comes from fault classes it cannot detect by construction, not from fault classes it detects but resolves slowly or unreliably. This has a direct, practical implication for teams deciding where to invest next in operational automation: the marginal value is concentrated in adding observation signals a process-level check cannot provide (an application-level health probe, network-attachment state, short-window failure history), not in a faster or smarter version of the restart action itself.

7.2 Scope of "AI-driven"

We restate plainly what Section 4.3 already states: the evaluated policy is a deterministic rule-and-score engine, not a trained model and not a live large-language-model agent. This is a real limitation of the present study's evidence about learned or LLM-backed policies specifically, and we do not claim the results generalize to them. What the results do support is a narrower but still useful claim: that the **architecture** of fusing multiple real signals and short-window history, independent of which specific decision mechanism consumes them, is what produces the observed gap over a single-signal blind rule. A learned or LLM-backed policy consuming the same signals would be a natural next evaluation; the harness built for this paper (Section 3, released with the supplementary artifacts) is designed to make that substitution straightforward, since the policy interface is a single pure function of (pod, telemetry, history).

7.3 Generalizability

This is a single-node, single-service, three-replica reference implementation, not a claim about behavior at production multi-node, multi-service scale, and we do not present it as one. The fault taxonomy (Section 3.3) was chosen for conceptual coverage (process-visible vs. not) rather than to be exhaustive of every real production failure mode; classes not evaluated here include multi-node network partitions, cascading failures across dependent services, and slow, gradual resource leaks rather than the abrupt faults injected in this study. We consider the taxonomy and the harness a starting point for exactly these extensions rather than a finished catalog.

8. Threats to Validity

- **Construct validity of "AI-driven."** The proposed policy is deterministic and rule-based, not a trained or LLM-backed system; results characterize the value of a multi-signal architecture, not of any specific learning method (Section 7.2).
- **Scale.** A single node, three replicas, and one workload service; no claim is made about multi-node or multi-service dynamics (Section 7.3).
- **Substrate fidelity.** Pod execution is backed by a custom node agent rather than stock kubelet, for a documented, diagnosed environmental reason (Section 3.2, Appendix A) rather than a design preference; the control-plane layer above it (API server, scheduler, controller-manager) is unmodified upstream Kubernetes.
- **Fixed observation window.** All trials share a 30-second recovery timeout; Section 6.4 reports rather than conceals the three trials in which this window, not the remediation logic, appears to be the limiting factor.
- **Sample size asymmetry.** Baseline and reactive were run at $n=8$ /cell and the AI-driven policy at $n=15$ /cell, a deliberate allocation toward the noisier condition (Section 5.2) rather than an equal- N design; absolute counts, not only rates, are reported throughout so this is auditable.
- **Single implementation, single run.** Each cell's trials were collected in one continuous harness run per condition; we do not report cross-run or cross-day variance, which a longer-running follow-up study should characterize.

9. Conclusion and Future Work

Against a real Kubernetes control plane and real, independently-timed fault injection across six fault classes, a deterministic multi-signal diagnostic policy raised overall remediation recovery rate from 14.6% (no automation) and 47.9% (blind restart-on-crash) to 96.7%, resolving five of six fault classes at 100% and the sixth, a bad rollout, at 80% via an explicit rollback action rather than an endless blind restart loop. The gap over the blind restart rule is concentrated almost entirely in fault classes -- a silent application hang, a network partition, and CPU starvation under load -- that are invisible to any policy consulting only process-level state, regardless of how quickly or aggressively it polls; the practical implication is that the highest-leverage next step

for teams operating Kubernetes workloads is adding observation signals a liveness probe does not already provide, not building a faster restart mechanism.

Three directions follow directly from this study's stated limitations. First, replacing the deterministic policy's decision function with a learned or LLM-backed one, evaluated against the identical harness and fault taxonomy, to test whether the multi-signal architecture's benefit compounds with a more sophisticated decision mechanism or is already largely captured by the rule-based version. Second, extending the fault taxonomy to multi-node and multi-service scenarios (cascading failures, dependent-service network partitions), which this single-node study was not designed to characterize. Third, running the harness continuously over a longer horizon to characterize cross-run variance, particularly for the noisier fault classes (CPU starvation, bad rollout) identified in Section 6.

References

- [1] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 50-57. <https://dl.acm.org/doi/10.1145/2890784>
- [2] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media. <https://sre.google/sre-book/table-of-contents/>
- [3] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos Engineering. *IEEE Software*, 33(3), 35-41. <https://research.google/pubs/chaos-engineering/>
- [4] Notaro, P., Cardoso, J., & Gerndt, M. (2021). A Survey of AIOps Methods for Failure Management. *ACM Transactions on Intelligent Systems and Technology*, 12(6), Article 81. <https://dl.acm.org/doi/10.1145/3483424>
- [5] Nygard, M. T. (2018). *Release It!: Design and Deploy Production-Ready Software* (2nd ed.). Pragmatic Bookshelf. <https://pragprog.com/titles/mnee2/release-it-second-edition/>
- [6] virtual-kubelet (CNCF Sandbox project). *virtual-kubelet/virtual-kubelet*. <https://github.com/virtual-kubelet/virtual-kubelet>