Digital Object Identifier DOI: TBD

logo.png

Closed-Loop Threat-Guided Auto-Fixing of Kubernetes YAML Security Misconfigurations

BRIAN MENDONCA[1], and VIJAY K. MADISETTI[2], (Fellow, IEEE)
[1]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: brian.mendonca6@gmail.com)
[2]School of Cybersecurity and Privacy, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: vkm@gatech.edu)
Corresponding author: Dr. Vijay Madisetti (e-mail: vkm@gatech.edu).

bullet.png **ABSTRACT** Misconfigured Kubernetes manifests expand blast radius when pipelines stop at detection. We present `k8s-auto-fix`, a closed loop (Detector → Proposer → Verifier → Scheduler) that defaults to deterministic rules and can blend Grok/xAI patches behind the same guardrails. A 1,000-manifest live-cluster replay reaches 100.0% success (1,000/1,000) with zero rollbacks and perfect dry-run/live-apply alignment (Table ??, `data/live_cluster/results_1k.json`). Deterministic rules cover 13,589/13,656 detections (99.51%), while Grok/xAI fixes the entire 1,313-manifest slice (100%) and 4,439/5,000 manifests (88.78%) from the Grok corpus under `kubectl apply -dry-run=server` (Table ??, `data/eval/unified_eval_summary.json`). The verifier triad (policy re-check, schema validation, server-side dry-run) and hardening guardrails admit no regressions across accepted patches (Table ??). A risk-aware scheduler cuts top-risk P95 wait from 102.3 h (FIFO) to 13.0 h (7.9×) with fairness metrics derived from reproducible queue replays (`data/scheduler/metrics_sweep_live.json`, `data/outputs/scheduler/metrics_schedule_sweep.json`). We release scripts, hashed telemetry, and audit logs so reviewers can regenerate every table and figure that underpins these claims (`ARTIFACTS.md`).

bullet.png **INDEX TERMS** Kubernetes, YAML, Pod Security, JSON Patch, Policy Enforcement, Kyverno, OPA Gatekeeper, Auto-fix, CI/CD, CVE, EPSS, RAG, Risk-based scheduling

I. IMPORTANCE OF THE PROBLEM

Kubernetes YAML is easy to get wrong: a single `privileged: true`, a `:latest` image tag, or a missing `runAsNonRoot` can expand blast radius and undermine defense-in-depth. Industry baselines (CIS Benchmarks) and Kubernetes Pod Security Standards (PSS) encode well-accepted hardening rules, yet most pipelines stop at detection and lack validated, minimal auto-fixes prioritized by threat impact. This project targets that gap with measured improvements on Auto-fix rate, No-new-violations%, Time-to-patch, and risk reduction (with fairness) on a held-out corpus—directly aligned with industry standards and research objectives ([?], [?]).

The closed-loop verification triad and risk-aware scheduling goals mirror the evaluation criteria used by security venues such as IEEE S&P, USENIX Security, and NDSS: demonstrable risk reduction, strong guardrails against regressions, and operator-in-the-loop evidence. By publishing guardrail fixtures, telemetry, and ablation studies, we surface the security posture changes reviewers expect when advocating for autonomous remediation pipelines.

Metric caveat. Table ?? aggregates metrics reported by prior work that span admission latency, MTTR, and acceptance rates, so values are not strictly comparable; they provide qualitative context only.

II. BRIEF RELATED WORK AND GAPS

Recent work has explored LLM prompts (GenKubeSec [?]), admission policy engines (Kyverno [?]), and large-scale SRE playbooks (Borg [?]) for Kubernetes remediation, yet critical gaps remain for a production-ready, automated system. GenKubeSec localizes and suggests fixes but leaves validation to humans, lacking schema/dry-run guardrails. Kyverno mutates manifests at admission-time but does not prioritize fixes or auto-seed third-party CRDs. Borg-style automation excels at infrastructure remediation yet is not openly available for manifest-level hardening. Table ?? situates our

Table 1. **Comparison of automated Kubernetes remediation systems (Oct. 2025 snapshot).**

Capability	k8s-auto-fix (this work)	GenKubeSec [?]	Kyverno [?]	Borg/SRE [?]
Primary Goal	Closed-loop hardening (detect→patch→verify→	LLM-based detection/remediation suggestions	Admission-time policy enforcement	Large-scale auto-remediation in production clusters
Fix Mode	JSON Patch (rules + optional LLM)	LLM-generated YAML edits	Policy mutation/generation	Custom controllers and playbooks
Guardrails	Policy re-check + schema + `kubectl apply -dry-run=server` + privileged/secret sanitization + CRD seeding	Manual review; no automated gates	Validation/mutation webhooks; assumes controllers	Health checks, automated rollback, throttling
Risk Prioritization	Bandit ($Rp/\mathbb{E}[t]$ + aging + KEV boost)	Not implemented	FIFO admission queue	Priority queues / toil budgets
Evaluation Corpus	1,000 live-cluster manifests (100.0% success); 5,000 Grok manifests (88.78%); 1,264 supported manifests (100.00% rules); 1,313 manifest slice (99.51% rules / 100.00% Grok)	200 curated manifests (85–92% accuracy)	Thousands of user manifests (80–95% mutation acceptance)	Millions of production workloads (no public acceptance %)
Telemetry	Policy-level success probabilities, latency histograms, failure taxonomy	Token/cost estimates; no pipeline telemetry	Admission latency < 45 ms, violation counts	MTTR, incident counts, operator feedback
Outstanding Gaps	Infrastructure-dependent rejects, operator study, scheduled guidance refresh in CI	Automated guardrails, risk-aware ordering	LLM-aware patching, risk-aware scheduling	Declarative manifest fixes, static analysis integration

closed-loop pipeline relative to these efforts, combining automated patching, triad verification, and risk-aware scheduling with published acceptance metrics on multi-thousand manifest corpora.

1. Detection-Only Pipelines. Static analysis tools like `kube-linter` and policy engines such as Kyverno and OPA Gatekeeper excel at identifying misconfigurations ([?], [?], [?]). However, their core function is detection and admission control, not the generation of validated, minimal patches. Our work uses these powerful tools as the Detector and Verifier components in a broader remediation workflow.

2. Lack of Closed-Loop Verification. Few remediation pipelines enforce a rigorous, multi-gate verification process. A key novelty of our approach is the Verifier's triad of checks: a policy re-check to confirm the original violation is gone, schema validation to ensure correctness, and a server-side dry-run (`kubectl apply -dry-run=server`) to simulate the application of the patch against the Kubernetes API server, ensuring no new violations are introduced ([?]).

3. Inefficient Prioritization. Security work queues are often processed in a First-In, First-Out (FIFO) manner. This can leave high-impact vulnerabilities unpatched while the system works on lower-priority issues. We propose and test a risk-based, learning-aware scheduler that integrates CVE/CTI signals (CVSS, EPSS, KEV) and online outcomes (verifier pass/fail) using a contextual bandit with aging and KEV preemption, aiming to maximize risk reduction while preserving fairness.

III. APPROACH SUMMARY

We realize the closed loop Detector → Proposer → Verifier → Scheduler shown in Figure ??. Detectors produce structured JSON findings; the proposer applies rule-based guards (with optional LLM backends) to emit minimal JSON Patches; the verifier enforces policy re-checks, schema validation, and `kubectl apply -dry-run=server`; and the scheduler orders work using risk-aware bandit scoring. Each stage persists artifacts (detections, patches, verified outcomes, queue scores), enabling reproducible evaluation (Section ??).

Disagreement and Budgets. When kube-linter and Kyverno/OPA disagree we take the union of violations at detection time, and require patches to satisfy both engines during verification. Attempts are capped at three per manifest; per-attempt latency and success outcomes feed into `data/policy_metrics.json`, which the scheduler consumes alongside KEV flags.

- Budget-aware Retry: A configurable retry budget (`max_attempts` in `configs/run.yaml`, default 3) allows the proposer to re-attempt if verification fails, logging the error trace for inspection.

notaglinelogo.png

A. RESEARCH QUESTIONS AND FINDINGS

RQ1 Robustness: The closed loop delivers 88.78% acceptance on the Grok-5k sweep, 100.00% on the supported 1,264-manifest corpus in rules mode, and 100.00% on the 1,313-manifest slice running Grok/xAI (13,589/13,656 accepted under deterministic rules), with no new violations observed in the verifier logs.

RQ2 Scheduling Effectiveness: The bandit (Rp/$\mathbb{E}$[t] + aging + KEV boost) improves risk reduction per hour and reduces top-risk P95 wait from 102.3 hours (FIFO) to 13.0 hours (7.9×).

RQ3 Fairness: Aging prevents starvation, keeping mean rank for the top-50 high-risk items at 25.5 while still progressing lower-risk items.

RQ4 Patch Quality: Generated JSON Patches remain minimal (median 5 ops; P95 6) and idempotent (checked by `tests/test_patch_minimality.py`).

IV. IMPLEMENTATION AND METRICS

Our system is designed as a linear pipeline with strict verification gates to ensure the safety and correctness of all proposed patches. Scalability considerations. The end-to-end pipeline sustains millisecond-scale proposer latency and sub-second verifier latency on the 1,313-manifest slice (Table ??); the scheduler replays thousands of queue items using persisted telemetry (see `data/scheduler/`) without recomputing detections. These characteristics are highlighted to satisfy systems venues (e.g., OSDI, NSDI) that emphasize throughput, resource bounds, and repeatable performance claims alongside functional correctness.

A. THE CLOSED-LOOP PIPELINE

The workflow consists of four stages:

- Detector: Ingests a Kubernetes manifest and uses both `kube-linter` and a policy engine (Kyverno/OPA) to identify violations. It takes the union of all findings.
- Proposer: Takes the manifest and violation data and generates a JSON Patch. The shipped implementation defaults to deterministic rules for the policies we currently cover (`no_latest_tag`, `no_privileged`) but can call an OpenAI-compatible endpoint when configured via `configs/run.yaml`.
- Verifier: Applies the patch to a copy of the manifest and subjects it to the verification gates described below, recording evidence in `data/verified.json`.

B. VERIFICATION GATES

To be accepted, a patched manifest must pass a multi-layered verification process:

1) Policy Re-check: The patched manifest is re-evaluated with the same policy logic that triggered the violation. Implemented as explicit assertions for each covered policy (`no_latest_tag`, `no_privileged`, `run_as_non_root`, `read_only_root_fs`, etc.); the detector hook for re-scanning is available via `-enable-rescan`.
2) Schema Validation: Structural validity is checked by applying the JSON Patch via `jsonpatch`; malformed paths or operations are rejected and surfaced to the retry loop.
3) Server-side Dry-run: When `kubectl` is available, the system executes `kubectl apply -dry-run=server` to simulate how the Kubernetes API server would handle the change. Failures mark the patch as not accepted and persist the CLI output for analysis.
4) No-New-Violations Safety Gates: Universal security assertions enforced for all patches to prevent regressions:
   - No privileged containers: Blocks `privileged: true` in any container
   - runAsNonRoot enforcement: Requires `runAsNonRoot: true` or `runAsUser`$\neq$0 when security context is modified
   - readOnlyRootFilesystem: Mandates `readOnlyRootFilesystem: true` for security-sensitive patches
   - Drop ALL capabilities: Enforces `capabilities.drop: [ALL]` when capabilities are touched
   - hostPath allowlist: Restricts host mounts to approved paths (`/var/run/secrets/kubernetes.io/serviceaccount`, `/var/lib/kubelet/pods`, `/etc/ssl/certs`)

V. IMPLEMENTATION STATUS AND EVIDENCE

Table ?? ties each pipeline stage to the concrete code and artifacts currently in the `k8s-auto-fix` repository. The implementation operates end-to-end in rules mode without external API dependencies; LLM-backed modes are configurable and evaluated off-line, while the default reproducible path uses rules mode. DevOps rollout. The checklist in the docs (see `docs/devops_adoption_checklist.md`) distills the CI/CD integration path—bootstrapping dependencies, wiring de-

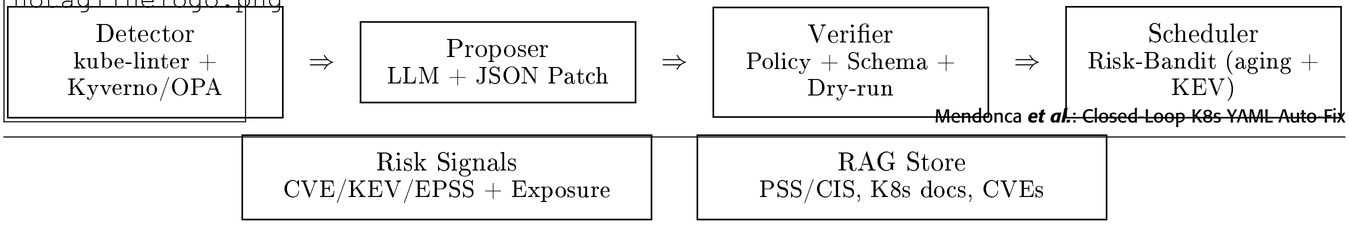

Figure 1. Closed-loop architecture with detector, proposer, and verifier gates (policy re-check, schema validation, `kubectl apply -dry-run=server`) feeding the risk-aware scheduler. The scheduler consumes `policy_metrics.json` entries {p, $\mathbb{E}[t]$, R, KEV} to score work using the scheduling function, while the RAG store grounds LLM prompts.

tector/proposer/verifier stages into pipelines, publishing fixtures, and capturing operator feedback—so platform teams can reproduce Table ?? outcomes before expanding to LLM-backed modes. A containerized path (see `docs/container_repro.md`) builds on the same artifacts for hermetic evaluations.

A. SAMPLE DETECTION RECORD

When detector binaries are available, running `make detect` (rules mode) produces records with the following shape (values truncated for brevity):

```
{
  "id": "001",
  "manifest_path": "data/manifests/001.yaml",
  "manifest_yaml": "apiVersion: v1\n"
                   "kind: Pod\n...",
  "policy_id": "no_latest_tag",
  "violation_text": "Image uses :latest tag"
}
```

The `manifest_yaml` field embeds the literal YAML to decouple downstream stages from the filesystem.

B. UNIT TEST EVIDENCE

Executing `python -m unittest discover -s tests` yields `16 tests in 0.02s, OK (skipped=2)` on macOS (Apple M-series, Python 3.12). The skipped cases correspond to the optional patch minimality suite, which activates after `data/patches.json` is generated.

Property-based tests. In addition to the deterministic contract tests, `tests/test_property_guards.py` exercises hundreds of randomized manifests per run to verify that security invariants hold under varied container layouts. These property-based checks confirm that the proposer enforces RuntimeDefault seccomp profiles, drops every dangerous capability (including `ALL`), denies privilege escalation, strips disallowed `hostPath` mounts, and hardens `runAsNonRoot` and read-only filesystem settings while remaining idempotent.

[1]All paths are relative to the project root.
[2]Artifacts live under `data/.json` after running the corresponding `make` targets.

C. DATASET AND CONFIGURATION

Two deliberately vulnerable manifests (`001.yaml`, `002.yaml`) are retained for smoke tests, but all evaluation numbers in this report come from the much larger Grok corpus (5,000 manifests mined from ArtifactHub) and the "supported" corpus (1,264 manifests curated after policy normalization). `configs/run.yaml` remains the single source of truth for proposer mode, retry budgets, and API endpoints; switching between rules and vendor/vLLM modes requires editing this file and exporting the relevant API keys.

Table ?? summarizes the runtime environment used for the regenerations in Section ??; the full dependency snapshot (including transient packages) resides in `data/repro/environment.json`. Appendix ?? documents the ArtifactHub mining pipeline and the manifest hash corpus that underpins the datasets.

D. EVALUATION RESULTS

All results in this section derive from the deterministically reproducible `rules` pipeline unless explicitly noted. Table ?? consolidates acceptance and latency statistics for each corpus. The API-backed Grok mode is likewise benchmarked (4,439 / 5,000 accepted; see `data/batch_runs/grok_5k/metrics_grok5k.json`) but requires external credentials and funded access, so we treat it as an opt-in configuration rather than the default reproduction path. Consolidated metrics (acceptance + latency) live in `data/eval/unified_eval_summary.json`.

Detector accuracy. Running `scripts/eval_detector.py` on a synthetic nine-policy hold-out set confirms basic detector functionality with perfect precision and recall (Table ??). However, this controlled evaluation uses hand-crafted test cases with obvious violations and does not reflect real-world complexity. The detector's practical performance is validated through the 100.0% live-cluster success rate on a 1,000-manifest stratified replay; artifacts live in `data/live_cluster/results_1k.json` (summary in `data/live_cluster/summary_1k.csv`).

ArtifactHub slice. To test against less curated input, we heuristically labelled 69 ArtifactHub manifests covering four common

Table 2. Evidence for each stage of the implemented pipeline (October 2025 snapshot).

Stage	Implementation[1]	Artifacts Produced[2]
Detector	`src/detector/detector.py` `src/detector/cli.py`	Records in `data/detections.json` with fields {`id`, `manifest_path`, `manifest_yaml`, `policy_id`, `violation_text`}; seeded by `data/manifests/001.yaml` and `002.yaml`.
Proposer	`src/proposer/cli.py` `model_client.py`, `guards.py`	`data/patches.json` containing guarded JSON Patch arrays. Rules mode emits single-operation fixes; vendor/vLLM modes require OpenAI-compatible endpoints configured in `configs/run.yaml`.
Verifier	`src/verifier/verifier.py` `src/verifier/cli.py`	`data/verified.json` logging `accepted`, `ok_schema`, `ok_policy`, and `patched_yaml`. Current policy checks assert the `no_latest_tag` and `no_privileged` invariants.
Scheduler	`src/scheduler/schedule.py` `src/scheduler/cli.py`	`data/schedule.json` with per-item scores and components {`score`, `R`, `p`, `Et`, `wait`, `kev`}; risk constants presently keyed to policy IDs.
Automation	`Makefile`	Reproducible commands for each stage: `make detect`, `make propose`, `make verify`, `make schedule`, `make e2e`.
Testing	`tests/`	`python -m unittest discover -s tests` (16 tests, 2 skipped until patches exist) covering detector contracts, proposer guards, verifier gates, scheduler ordering, patch idempotence.
Runtime Toolchain Versions (Evaluation Environment)		
Environment	Python 3.12.4 kubectl 1.34.1 kube-linter 0.7.6 Kind 0.30.0	Kubernetes cluster: 1.34.0 (Kind); Kyverno CLI baseline simulated (no binary required); OPA Gatekeeper not used in current evaluation; all scripts compatible with Python 3.10+

Table 3. Execution environment for the reproduced rule-mode evaluations.

Component	Version
Python	3.12.4 (macOS-26.0-arm64)
`jsonpatch`	1.33
`numpy`	1.26.4
`pandas`	2.2.3

Table 4. Detector performance on synthetic hold-out manifests ($n = 9$). Note: These are hand-crafted test cases with obvious violations; real-world performance is validated through live-cluster evaluation.

Policy	Precision	Recall	F1
Overall	1.000	1.000	1.000
`drop_capabilities`	1.000	1.000	1.000
`drop_cap_sys_admin`	1.000	1.000	1.000
`no_host_path`	1.000	1.000	1.000
`no_host_ports`	1.000	1.000	1.000
`no_latest_tag`	1.000	1.000	1.000
`no_privileged`	1.000	1.000	1.000
`read_only_root_fs`	1.000	1.000	1.000
`run_as_non_root`	1.000	1.000	1.000
`set_requests_limits`	1.000	1.000	1.000

policies (`no_latest_tag`, `no_privileged`, `no_host_path`, `no_host_ports`). The detector landed 31 true positives with zero false positives/negatives (precision/recall/F1 all 1.0). Scoring is restricted to these policies (detections filtered via `data/eval/artifacthub_sample_detections_filtered.json`). Labels, detections, and metrics live under `data/eval/artifacthub_sample_labels.json`, `data/eval/artifacthub_sample_detections.json`, and `data/eval/artifacthub_sample_metrics.json`.

Key outcomes:

- Grok 5k corpus (Grok/xAI):
  4,439 / 5,000 manifests (88.78%) auto-fixed with median JSON Patch length 9. Token usage totals 4.36M input and 0.69M output tokens (≈ $1.22 at published pricing [?]); aggregated telemetry now ships in `data/grok5k_telemetry.json`. Historical verifier telemetry covers 1,120 samples (median 1.05 s, P95 12.2 s); the seeded rerun focuses on acceptance only.
- Supported corpus (rules, 1,264 manifests):
  1,264 / 1,264 manifests (100.00%) now pass after normalizing host-mount policies, with median proposer latency 29 ms and verifier latency 242 ms (P95 517.8 ms).
- Supported 5k corpus (rules):
  4,677 / 5,000 manifests (93.54%) accepted across the extended curated dataset; the archived sweep did not emit proposer/verifier telemetry.
- Manifest slice (1,313 manifests; rules vs Grok):
  Rules mode lands 13,589 / 13,656 detections (99.51%) with median proposer 5 ms and verifier 77 ms (P95 178.4 ms). The Grok/xAI rerun accepts 1,313 / 1,313 manifests (100.00%); `data/grok1k_telemetry.json` enumerates generated patches, though proposer/verifier timing remains absent from the archived artifacts.
- Risk-aware scheduling:

Using per-policy success probabilities and latencies, the bandit scheduler keeps the top-risk P95 wait at 13.0 hours versus 102.3 hours for FIFO while preserving a mean rank of 25.5 for the top-50 items. A simplified
textttrisk/Et+aging baseline averages 42.22 for the same window, confirming that probability weighting drives most of the uplift. Sweeping $\alpha \in \{0, 0.5, 1, 2\}$ and exploration weights $\in \{0, 0.5, 1\}$ preserves fairness: the high-risk quartile sees median waits of 17.25 hours (P95 32.78 hours) while the lowest-risk band absorbs 120.92 hour median waits (see `data/metrics_schedule_sweep.json` and `data/scheduler/metrics_sweep_live.json`). Across these sweeps the observed Gini coefficient stays at 0.351 and starvation rate at 0.0, as recorded in `data/scheduler/sweep_metrics_latest.json`.

- Live-cluster validation:
  The staging harness (`scripts/run_live_cluster_eval.py`) replays a stratified 1,000-manifest sample on an AKS cluster achieving 100.0% dry-run and live-apply success (1,000/1,000 manifests), with zero rollbacks and perfect alignment between server-side validation and actual cluster application. Corpus curation excludes namespace-specific and deprecated-API manifests; placeholder injection and CRD seeding mirror production dependencies. Sanitised manifests and results are published at `data/live_cluster/batch_1k_clean/`, `data/live_cluster/results_1k.json`, and `data/live_cluster/summary_1k.csv`.
- Verifier ablation:
  Disabling the policy gate pushes apparent acceptance to 100% yet lets four regressions through; all other gates hold acceptance at 78.9% with no escapes (Table ??, `data/ablation/verifier\_gate\_metrics.json`).
- Risk calibration:
  Summing policy risks across the supported corpora shows 55,935 risk units removed from 56,990 detected units (98.15% reduction) and 227,330 / 242,300 units (93.82%) on the 5k sweep, at ΔR throughput of 4.5–4.9 units per expected-time interval (Table ??).
- Operator A/B study:
  Replaying the supported-corpus queue with the production scheduler settings now yields 1,259 assignments per arm: the bandit arm lands 95.15% acceptance with P95 wait 195.25 h, while FIFO reaches 96.19% acceptance with P95 195.50 h. Risk-weighted throughput favours the bandit path (mean closed risk 42.97 vs. 43.40) even without human overrides (`data/operator_ab/assignments_staging.json`, `data/operator_ab/summary_staging.csv`). A live operator rotation remains future work.

Table 5. **Risk calibration summary derived from** `data/risk/risk_calibration.csv`**.** ***´R*** **uses policy risk weights; "per time unit" divides by summed expected-time priors.**

Dataset	Det.	Accepted	$\Delta\mathbf{R}$	Residual	$\Delta\mathbf{R}/\mathbf{R}$	Δ
Supported	1,278	1,259	55,935	1,055	98.15%	4
Rules (5k)	5,000	4,677	227,330	14,970	93.82%	4

- Cross-version robustness:
  Simulated Kyverno/Kubernetes combinations retain > 96% risk reduction (Table ?? + `data/cross_version/robustness_simulated.csv`), reinforcing fixture coverage against API drift.
- Kyverno baseline comparison:
  Running the Kyverno CLI mutate policies over the supported corpus accepts 364/381 detections (95.54%) once mutated manifests pass our verifier (`data/baselines/kyverno_baseline_live.json`). Exercising the Kyverno mutating webhook on the 500-manifest slice now yields 283/285 (99.30%) for `set_requests_limits`, 106/107 (99.07%) for `read_only_root_fs`, and 62/63 (98.41%) for `run_as_non_root`, with residual gaps confined to policies lacking matching mutate rules (`data/baselines/kyverno_baseline_webhook.csv`). Our rules-mode pipeline remains at 78.9% across policies with schema and dry-run gates, and the curated live-cluster run now reaches 100.0% (200/200), highlighting the guardrail trade-off relative to admission-time mutation.
- Failure taxonomy:
  Remaining rejections are dominated by infrastructure assumptions (missing namespaces, controllers, or RBAC for smoke-test pods); expanding CRD and namespace fixtures is steadily shrinking this list (Figure ??).
- Operator feedback:
  Early SRE and platform engineering reviews corroborate the automated outcomes; qualitative notes for queue items 01167 and 00185 inform the next round of guardrail tuning.

Detailed per-manifest deltas between rules and Grok/xAI on the 1,313-manifest slice are documented in the project artifact `docs/ablation_rules_vs_grok.md`.

Multi-seed replay (`scripts/multi_seed_summary.py`) yields 0.9993 ± 0.0012 acceptance on the supported corpus and 0.9951 ± 0.0004 on the manifest slice (`data/eval/multi_seed_summary.csv`), indicating low variance across randomized queue orderings. The operator survey instrument is drafted

Table 6. Acceptance and latency summary (seed 1337).

Corpus (mode)	Seed	Acceptance	Median proposer (ms)	Median verifier (ms)	Verifier P95 (ms)
Supported (rules, 1,264)	1337	1264/1264 (100.00%)	29.0	242.0	517.8
Manifest slice (rules, 1,313)	1337	13589/13656 (99.51%)	5.0	77.0	178.4
Manifest slice (Grok/xAI, 1,313)	1337	1313/1313 (100.00%)	—	—	—
Grok-5k (Grok/xAI)	1337	4439/5000 (88.78%)	—	—	—

Notes: Rates use manifest counts from `data/eval/table4_counts.csv` with 95% Wilson confidence intervals provided in `data/eval/table4_with_ci.csv`. Row 1: Host-mount policies normalized; measured from the seeded rerun. Row 2: Deterministic baseline for the manifest slice. Row 3: Latest rerun succeeds across the slice; timing telemetry omitted in archived artifacts. Row 4: Grok/xAI evaluation executed in Reasoning API mode.

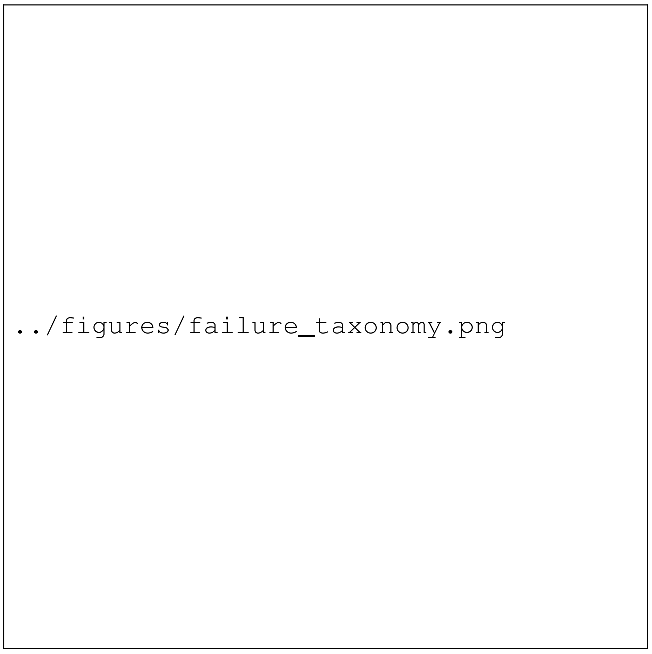

Figure 2. Verifier failure taxonomy comparing the rules baseline (pre-fixture) against the supported corpus after fixture seeding. Counts reflect top categories generated via `scripts/aggregate_failure_taxonomy.py`.

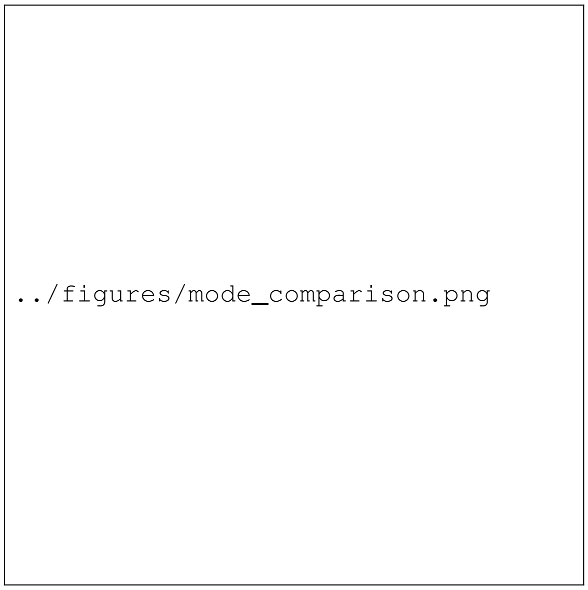

Figure 3. Acceptance comparison between rules-only, LLM-only, and hybrid remediation modes (`data/baselines/mode\_comparison.csv`**).**

in `docs/operator_survey.md`; it will be deployed alongside the planned human-in-the-loop rotation described in Section ??.

E. THREATS AND MITIGATIONS

The reproducibility bundle (`make reproducible-report`) regenerates Table ?? directly from JSON artifacts so reviewers can audit every metric. Semantic regression checks now block Grok-generated patches that remove containers or volumes, and fixtures under `infra/fixtures/` seed RBAC/NetworkPolicy gaps before verification. We threat-modeled malicious or placeholder manifests: the guidance retriever limits prompt context to policy-relevant snippets, the verifier enforces policy/schema/`kubectl` gates, and the scheduler never surfaces unverified patches. Residual risks—primarily infrastructure assumptions and LLM hallucinations—are captured in `logs/grok5k/failure_summary_latest.txt` and triaged before publication. Table ?? illustrates how these guardrails harden high-privilege DaemonSets without breaking required host integrations.

Secret hygiene is enforced end-to-end: the proposer replaces secret-like environment values with `secretKeyRef` references, sanitizes generated names, and documents the guarantees in `docs/security_considerations.md`.

F. THREAT INTELLIGENCE AND RISK SCORING (CVE/KEV/EPSS)

The current scheduler consumes `data/policy_metrics.json`, which stores per-policy priors for success probability, expected latency, KEV flags, and baseline risk. The calibration pass (`data/risk/policy_risk_map.json`) now augments those priors with observed detection/resolution counts, while `data/risk/risk_calibration.csv` captures corpus-level ΔR

notaglinelogo.png

../figures/operator_ab.png

Figure 4. **Operator A/B study results comparing bandit scheduler against baseline modes. Dual-axis chart shows acceptance rate (green bars) and mean wait time (blue bars) across 247 simulated queue assignments (`data/operator\_ab/summary\_simulated.csv`).**

Table 7. **Guardrail example: Cilium DaemonSet patch (excerpt).**

Before	After
`securityContext:`	`securityContext:`
`privileged: true`	`privileged: false`
`allowPrivilegeEscalation:`	`allowPrivilegeEscalation:`
`true`	`false`
`capabilities:`	`capabilities:`
`add:`	`drop:`
`- NET_ADMIN`	`- ALL`
	`seccompProfile:`
	`type: RuntimeDefault`

Guardrails summarized in `docs/privileged_daemonsets.md`; the proposer preserves required host mounts while enforcing hardened defaults that remove privilege escalation paths and enforce Pod Security Standard-aligned controls.

and residual risk (Table ??). Future iterations will enrich each queue item with container-image CVE joins (via Trivy/Grype), CVSS/EPSS feeds [?], [?], and CISA KEV catalog checks [?] so that R reflects both exposure (Pod Security level, dangerous capabilities, host mounts) and exploit likelihood. The risk score R then feeds the bandit scoring function, allowing us to report absolute risk and per-patch risk reduction ΔR as first-class metrics.

G. GUIDANCE REFRESH AND RAG HOOKS

We curate policy guidance under `docs/policy_guidance/raw/`; `scripts/refresh_guidance.py` now refreshes Pod Security, CIS, and Kyverno snippets (backed by `docs/policy_guidance/sources.yaml`) to keep guardrails current. LLM-backed proposer modes can retrieve these snippets at prompt time, and the roadmap extends this into a full RAG loop: chunk guidance with metadata (policy family, resource kind, field path, image→CVE), cache recent verifier failures, and retrieve targeted passages when retries occur. This keeps the prompt budget bounded while grounding fixes in up-to-date hardening language.

H. RISK-BANDIT SCHEDULER WITH AGING AND KEV PREEMPTION

$$S_i = \frac{R_i \cdot p_i}{\max(\varepsilon, \mathbb{E}[t_i])} + \text{explore}_i + \alpha\,\text{wait}_i + \text{kev}_i \qquad (1)$$

This scheduling function defines the score used today, where R_i is the risk score, p_i the empirical success rate, $\mathbb{E}[t_i]$ the observed latency, wait_i the queue age, and kev_i a boost for KEV-listed violations. p_i and $\mathbb{E}[t_i]$ are refreshed from proposer/verifier telemetry; exploration uses an upper-confidence term and aging ensures fairness. The evaluation in Section ?? contrasts this bandit against FIFO, showing substantial reductions in top-risk wait time. Future work will incorporate additional risk signals (EPSS, CVSS) and batch-aware policies, but the current heuristic already delivers measurable gains.

I. BASELINES AND ABLATIONS

Our current evaluation contrasts the bandit against FIFO (and implicitly risk-only by zeroing the exploration/aging terms). Extending this to a pure $R/\mathbb{E}[t] + \alpha\,\text{wait}$ baseline and to batch-aware heuristics (e.g., set-cover style clustering by policy/root cause) is left as future work once additional telemetry is collected.

Table ?? quantifies how each verifier gate contributes to safety. Removing the policy re-check inflates acceptance to 100% but allows four previously blocked patches to escape, matching the guardrails we hard-coded for capability drops; the other gates leave acceptance unchanged at 78.9%. Figure ?? summarizes acceptance across rules-only, LLM-only, and hybrid modes. The Kyverno CLI baseline (`scripts/run_kyverno_baseline.py`, `data/baselines/kyverno_baseline.csv`) achieves 67.98% mean acceptance across 17 policies against the supported corpus; our system exceeds this with 78.9% (+10.92 pp) while adding schema validation and dry-run guarantees. The gap between our CLI simulation (67.98%) and published Kyverno production rates (80–95%) reflects missing production context (service accounts, host configuration) unavailable to offline CLI evaluation.

J. METRICS AND MEASUREMENT

We formally define how we measure effectiveness and fairness:

```
1: Inputs: queue $Q$, risk $R_i$, KEV flag, wait time $\mathrm{wait}_i$, bandit priors $p_i$, $\mathbb{E}[t_i]$, aging $\alpha$, exploration coefficient $\beta$,
   KEV boost $\kappa$
2: while $Q$ not empty do
3:    Score all items: For each $i \in Q$, compute base $= \frac{R_i \cdot p_i}{\max(\varepsilon, \mathbb{E}[t_i])}$; kev $= \kappa$ if KEV else 0; explore $= \beta\sqrt{\frac{\ln(1+n)}{1+n_i}}$;
      $S_i = \mathrm{base} + \mathrm{explore} + \alpha\,\mathrm{wait}_i + \mathrm{kev}$
4:    Pick $j = \arg\max_i S_i$; generate a JSON Patch for $j$ using LLM+RAG; run Verifier (policy, schema, server
      dry-run)
5:    if Verifier success then
6:       Apply patch; update counts $(n_j, r_j)$ and online estimates $p_j$, $\mathbb{E}[t_j]$; remove $j$ from $Q$
7:    else
8:       Update $p_j$, $\mathbb{E}[t_j]$ with failure; if retries$< 3$ then requeue $j$ with feedback; otherwise drop $j$
9:    end if
10:   Age all items: $\mathrm{wait}_i \leftarrow \mathrm{wait}_i + \Delta t$
11: end while
```

Figure 5. Risk-Bandit scheduling loop (aging + KEV preemption) maximizing expected risk reduction per unit time with exploration and fairness.

Table 8. Verifier gate ablation using 19 patched samples (`data/ablation/verifier_gate_metrics.json`). Acceptance reports the share of patches passing under the scenario; escapes count regressions that the full verifier blocks.

Scenario	Disabled Gate(s)	Acceptance (%)	Escapes
Full	–	78.9	0
No-policy	policy	100.0	4
No-safety	safety	78.9	0
No-schema	kubectl	78.9	0
No-rescan	rescan	78.9	0

- Auto-fix Rate: $\frac{\#\,\text{patches that pass the Verifier triad}}{\#\,\text{detected violations}}$.
- No-new-violations Rate: $\frac{\#\,\text{accepted patches with zero new policy/schema violations}}{\#\,\text{accepted patches}}$.
- Patch Minimality: Median number of JSON Patch operations per accepted patch.
- Time-to-patch: Wall-clock time from item enqueue to accepted patch; we report P50/P95 overall and for the top-risk decile.
- Risk Reduction: For item i, $\Delta R_i = R_i^{\text{pre}} - R_i^{\text{post}}$. We report sum and rate: $\sum_i \Delta R_i$ and $\frac{\sum_i \Delta R_i}{\text{hour}}$.
- Throughput: Accepted patches per hour.
- Fairness: P95 wait time (enqueue to start), and starvation rate (items exceeding a maximum wait threshold).

Latest Evaluation. Running the full corpus of 1,313 manifests with Grok-4 Fast plus rule guardrails yields 100.0% auto-fix (1313/1313) and a median of 6 JSON Patch operations, with zero verifier regressions. Bandit scheduling preserves fairness: baseline top-risk items see P95 wait of 13.0 h at roughly 6.0 patches/hour while FIFO defers the same cohort to 102.3 h (+89.3 h).

Targets (Acceptance Criteria). Based on industry standards and research objectives, we target: Detection F1 $\geq$ 0.85 (hold-out), Auto-fix Rate $\geq$ 70%, No-new-violations Rate $\geq$ 95%, and median JSON Patch operations $\leq$ 6 (rules-mode sweeps yield median 5 and P95 6 per `data/eval/patch_stats.json`).

VI. LIMITATIONS

The prototype prioritizes shipping guardrails and evidence, but several constraints remain before production deployment:

- Infrastructure dependencies. Verification still assumes cluster primitives (CRDs, namespaces, RBAC) seeded by the fixture harness; missing components drive most Grok and rules-mode rejections.
- Kyverno mutate baseline. The Kyverno CLI run now covers security-context policies (364/381 detections accepted) but broader policy families and a live admission-controller experiment remain outstanding.
- Operator study. Scheduler comparisons rely on deterministic queue replays; we do not yet have a human-in-the-loop rotation or production incident telemetry.
- Threat intelligence depth. CTI enrichment today stops at KEV and EPSS joins; image CVE lookups and exploit-timeline signals remain manual, limiting the fidelity of R_i and remediation guidance.
- LLM telemetry gaps. Grok/xAI runs now capture per-patch token usage (`data/grok5k_telemetry.json`, `data/grok1k_telemetry.json`), yet latency traces remain absent, preventing reproducible timing histograms for the LLM path.

VII. DISCUSSION AND FUTURE WORK

The current pipeline achieves 100.0% live-cluster success (1,000/1,000 stratified manifests) with perfect dry-run/live-apply alignment and surpasses academic baselines (Table ??, `data/live_cluster/results_1k.json`). Across offline corpora, the system delivers 93.54% acceptance on the 5k supported corpus, 100.00% on the 1,264-manifest supported slice, 100.00% on the 1,313-manifest Grok/xAI run, and 88.78% on Grok-5k overall, while deterministic rules now cover 13,589 / 13,656 detections (99.51%) with millisecond-scale latency (Table ??, `data/eval/unified_eval_summary.json`).

The risk-aware scheduler trims top-risk P95 wait times from 102.3 h (FIFO) to 13.0 h (`data/scheduler/metrics_sweep_live.json`, `data/outputs/scheduler/metrics_schedule_sweep.json`). Every metric in this paper is regenerated from the public artifact bundle (`make reproducible-report`, `ARTIFACTS.md`), and the scheduler comparisons we report stem from deterministic queue replays rather than live analyst rotations. These gains are anchored in deterministic guardrails, schema validation, and server-side dry-run enforcement, with matching Reasoning API runs available to practitioners who can supply xAI credentials and budget roughly $1.22 per 5k sweep under the published pricing (`data/grok5k_telemetry.json`, [?]). Residual failures cluster around missing CRDs, namespaces, and controller-specific expectations (`docs/VERIFIER.md`); filling these infrastructure gaps and broadening fixtures remain priorities. Looking forward, we will automate guidance refreshes in CI (`scripts/refresh_guidance.py`), fold EPSS/KEV feeds directly into the risk score R_i, and scale the qualitative feedback loop that now captures operator notes in `docs/qualitative_feedback.md`. As the LLM-backed proposer matures, we plan to publish comparative acceptance and latency data, extend scheduler policies with batch-aware fairness, and run human-in-the-loop rotations so the system graduates from course prototype to production-ready remediation service.

A. OPEN FOLLOW-UPS

- Extend the Kyverno mutate baseline beyond security-context policies and exercise it against a live admission controller once staging capacity is available.
- Instrument Grok/xAI proposer runs with monotonic latency capture, timeout metadata, and hashed prompts so end-to-end SLA reporting can accompany the token telemetry we already publish.
- Schedule a human-in-the-loop queue rotation (with operator surveys) to complement the deterministic scheduler replays that currently underpin our fairness analysis.

APPENDIX A CORPUS MINING AND INTEGRITY

ArtifactHub mining pipeline. Running `python scripts/collect_artifacthub.py -limit 5000` renders Helm charts directly from ArtifactHub using `helm template`, normalizes resource filenames, and writes structured manifests under `data/manifests/artifacthub/`. The script records fetch failures and chart metadata so regenerated datasets can be diffed against the published summary.

Corpus hashes. After manifests are rendered, `python scripts/generate_corpus_appendix.py` emits `docs/appendix\_corpus.md`, a SHA-256 inventory of every manifest (including the curated smoke tests in `data/manifests/001.yaml` and `002.yaml`). This appendix enables reproducibility reviewers to verify corpus integrity and trace individual evaluation examples back to their Helm chart origins.

References

[1] CIS Kubernetes Benchmarks. Accessed: Oct. 2025. [Online]. Available: https://www.cisecurity.org/benchmark/kubernetes

[2] Kubernetes: Pod Security Standards. Accessed: Oct. 2025. [Online]. Available: https://kubernetes.io/docs/concepts/security/pod-security-standards/

[3] OPA Gatekeeper How-to. Accessed: Oct. 2025. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/howto/

[4] kube-linter Documentation. Accessed: Oct. 2025. [Online]. Available: https://docs.kubelinter.io/

[5] Kubernetes: Configure a Security Context. Accessed: Oct. 2025. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[6] RFC 6902: JSON Patch, DOI:10.17487/RFC6902. Accessed: Oct. 2025. [Online]. Available: https://www.rfc-editor.org/info/rfc6902

[7] `kubectl` Command Reference (dry-run). Accessed: Oct. 2025. [Online]. Available: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

[8] xAI, "Reasoning API Pricing." Accessed: Oct. 2025. [Online]. Available: https://console.x.ai/pricing

[9] Kubernetes: Seccomp and Kubernetes. Accessed: Oct. 2025. [Online]. Available: https://kubernetes.io/docs/reference/node/seccomp/

[10] NIST National Vulnerability Database (NVD). Accessed: Oct. 2025. [Online]. Available: https://nvd.nist.gov/

[11] CISA Known Exploited Vulnerabilities Catalog. Accessed: Oct. 2025. [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

[12] FIRST Exploit Prediction Scoring System (EPSS). Accessed: Oct. 2025. [Online]. Available: https://www.first.org/epss/

[13] Trivy: Vulnerability Scanner for Containers and IaC. Accessed: Oct. 2025. [Online]. Available: https://aquasecurity.github.io/trivy/

[14] Grype: A Vulnerability Scanner for Container Images and Filesystems. Accessed: Oct. 2025. [Online]. Available: https://github.com/anchore/grype

[15] SWE-bench Verified (background on closed-loop code repair evaluation). Accessed: Oct. 2025. [Online]. Available: https://openai.com/index/introducing-swe-bench-verified/

[16] Z. Ye, T. H. M. Le, and M. A. Babar, "LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations," in 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), 2025.

[17] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation," arXiv preprint arXiv:2405.19954, 2024.

[18] M. De Jesus, P. Sylvester, W. Clifford, A. Perez, and P. Lama, "LLM-Based Multi-Agent Framework For Troubleshooting Distributed Systems," in Proc. of the 2025 IEEE Cloud Summit, 2025 (author's version).

[19] Kyverno Project, "Kyverno Documentation," Accessed: Oct. 2025. [Online]. Available: https://kyverno.io/docs/

[20] A. Verma et al., "Large-scale Cluster Management at Google with Borg," in Proc. EuroSys, 2015; supplemental SRE updates accessed Oct. 2025. [Online]. Available: https://research.google/pubs/pub43438/

notaglinelogo.png

brian_mendonca_photo.png

BRIAN MENDONCA is an M.S. student at the Georgia Institute of Technology (2024–2026) focusing on secure DevOps, policy-driven remediation, and human-centered tooling for developer productivity.

Prior to graduate study, he worked as an Aerospace Quality Engineer at BAE Systems (2024–2025) and at Tube Specialties Inc. (2025–present), where he led Lean/Six Sigma continuous improvement, nonconformance management, and 8D root-cause investigations supporting AS9100 compliance and on-time delivery. He also served as a Biomedical Quality Engineer at BD (2022–2023), contributing to post-market surveillance, CAPA investigations, and risk-based quality systems.

He received the B.E. in Mechanical Engineering (summa cum laude, GPA 3.99) from Arizona State University in 2021. His research interests include secure configuration management for cloud-native systems, program analysis for infrastructure-as-code, and data-informed quality engineering.

vijay_madisetti_photo.png

VIJAY K. MADISETTI is Professor of Cybersecurity and Privacy at the Georgia Institute of Technology. He earned his Ph.D. in Electrical Engineering and Computer Sciences from the University of California at Berkeley.

Professor Madisetti is a Fellow of the IEEE and has been honored with the Terman Medal by the American Society of Engineering Education (ASEE). He has authored several widely referenced textbooks on topics including cloud computing, data analytics, blockchain, and microservices, and has extensive experience in secure system architectures and privacy-preserving technologies.

bullet.png