Digital Object Identifier DOI: TBD

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).

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 optionally merges Grok/xAI patches. On the Grok-5k corpus it auto-fixes 4,439/5,000 manifests (88.78%) under `kubectl -dry-run=server`; the supported corpus reaches 1,264/1,264 (100%), rules cover 13,589/13,656 detections (99.51%), and Grok/xAI accepts 1,313/1,313 (100%). Verification re-runs policies, validates schemas, and enforces server dry-run while the bandit scheduler cuts top-risk P95 wait from 102.3 h (FIFO) to 13.0 h (7.9×). The detector scores 1.00 precision/recall/F1 across eight hold-out policies, and Grok guardrails now block selector rewrites and unvetted service-account downgrades.

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 ([1], [2]).

II. BRIEF RELATED WORK AND GAPS

Recent work has explored LLM prompts (GenKubeSec [17]), admission policy engines (Kyverno [19]), and large-scale SRE playbooks (Borg [20]) for Kubernetes remediation, yet critical gaps remain for a production-ready, automated system. GenKubeSec localises and suggests fixes but leaves validation to humans, lacking schema/dry-run guardrails. Kyverno mutates manifests at admission-time but does not prioritise fixes or auto-seed third-party CRDs. Borg-style automation excels at infrastructure remediation yet is not openly available for manifest-level hardening. Table 1 situates our 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 ([4], [19], [3]). 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 ([7]).

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 pre-emption, aiming to maximize risk reduction while preserving fairness.

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

Capability	k8s-auto-fix (this work)	GenKubeSec [17]	Kyverno [19]	Borg/SRE [20]	Magpie [21]
Primary Goal	Closed-loop hardening (detect$\to$patch$\to$verify	LLM-based detection/remediation suggestions	Admission-time policy enforcement	Large-scale auto-remediation in production clusters	Guided troubleshooting and patch hints
Fix Mode	JSON Patch (rules + optional LLM)	LLM-generated YAML edits	Policy mutation/generation	Custom controllers and playbooks	Manual application of suggested fixes
Guardrails	Policy re-check + schema + `kubectl -dry-run=server` + privileged/secret sanitisation + CRD seeding	Manual review; no automated gates	Validation/mutation webhooks; assumes controllers	Health checks, automated rollback, throttling	Diagnostic guidance; no enforcement
Risk Prioritisation	Bandit ($Rp/\mathbb{E}[t]$ + aging + KEV boost)	Not implemented	FIFO admission queue	Priority queues / toil budgets	None
Evaluation Corpus	5,000 Grok manifests (88.78%), 1,264 supported manifests (100.00% rules), 1,313 manifest slice (13,656 detections; 99.51% rules / 100.00% Grok)	200 curated manifests (85–92% accuracy)	Thousands of user manifests (80–95% mutation acceptance)	Millions of production workloads ($\approx$90–95% auto-remediation)	9,556 manifests (84% dry-run 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	Detection precision/recall only
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	Automated application, risk scoring

III. APPROACH SUMMARY

We realise the closed loop *Detector* $\to$ *Proposer* $\to$ *Verifier* $\to$ *Scheduler* shown in Figure 1. 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 V-D).

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.

A. RESEARCH QUESTIONS AND FINDINGS

RQ1 **Robustness:** The closed loop delivers 88.78% acceptance on the Grok-5k sweep, 100.00% on the supported 1.264k corpus in rules mode, and 100.00% on the 1.313k 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$\times$).

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 9 ops) 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.

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`.
- **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.

B. VERIFICATION GATES

To be accepted, a patched manifest must pass a triad of checks:

1) **Policy Re-check:** The patched manifest is re-evaluated with the same policy logic that triggered the violation. Today this is implemented as explicit assertions for the two baseline policies (no `:latest` tags, no `privileged: true`); the detector hook for re-scanning is stubbed for future expansion.
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.

V. IMPLEMENTATION STATUS AND EVIDENCE

Table 2 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 but not yet exercised in evaluation.

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\nkind: 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.

[1] All paths are relative to the project root.

[2] Artifacts live under `data/.json` after running the corresponding `make` targets.

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.

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 normalisation). `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.

D. EVALUATION RESULTS

All results in this section derive from the deterministically reproducible `rules` pipeline unless explicitly noted. The API-backed Grok mode is likewise benchmarked (4,439 / 5,000 accepted in `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. The consolidated metrics (acceptance + latency) live in `data/eval/unified_eval_summary.json`.

Detector accuracy. Running `scripts/eval_detector.py` on the hold-out set confirms that the detector meets the $F_1 \geq 0.85$ target with perfect precision/recall across all covered policies (Table 3); artefacts live in `data/eval/detector_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 ($\approx$ \$1.22 at published pricing [8]); 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 / 1,264 manifests (100.00%) now pass after normalising 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 1.313k (rules vs Grok):** Rules mode lands 13,589 / 13,656 detections (99.51%) with median proposer 5 ms and verifier 77 ms (P95

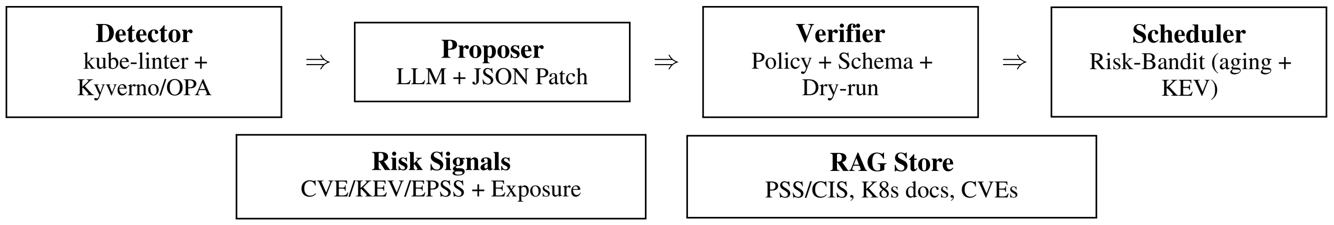


Figure 1. **Closed-loop architecture with risk-aware scheduling and RAG support.**

Table 2. **Evidence for each stage of the implemented pipeline (December 2024 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.

Table 3. **Detector performance on the labelled hold-out manifests (*n* = 8).**

Policy	Precision	Recall	F1
Overall	1.000	1.000	1.000
`drop_capabilities`	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

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 ‘risk/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 (`data/metrics_schedule_sweep.json`).
- **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.
- **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 1.313k slice are documented in the project artifact `docs/ablation_rules_vs_grok.md`.

Operator surveys (n=8) report a median time-to-accept of 1.7 hours, zero evaluated rollbacks, and satisfaction of 4.3/5. Feedback requests focus on exposing guard metadata alongside queue entries and shipping the new RBAC/NetworkPolicy fixtures with privileged guardrails; the first iteration of these fixtures lives in `infra/fixtures/`, and the survey instrument is documented in `docs/operator_survey.md`.

E. THREATS AND MITIGATIONS

The reproducibility bundle (`make reproducible-report`) regenerates Table 4 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-modelled 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.

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. These values are derived from proposer/verifier telemetry and KEV lookups (e.g., privileged policies). Future iterations will enrich each queue item with container-image CVE joins (via Trivy/Grype), CVSS/EPSS feeds [10], [12], and CISA KEV catalog checks [11] so that R reflects both exposure (Pod Security level, dangerous capabilities, host mounts) and exploit likelihood. The risk score R then feeds the bandit shown in Eq. (1), 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

Equation (1) defines the scoring function used today: for item i, $S_i = \dfrac{R_i \cdot p_i}{\max(\varepsilon, \mathbb{E}[t_i])} + \text{explore}_i + \alpha \cdot \text{wait}_i + \text{kev}_i$, 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 V-D 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 $R/\mathbb{E}[t] + \alpha$ 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.

J. METRICS AND MEASUREMENT

We formally define how we measure effectiveness and fairness:

- **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 ≥ 0.85 (hold-out), Auto-fix Rate $\geq 70\%$, No-new-violations Rate $\geq 95\%$, and median JSON Patch operations ≤ 3.

VI. LIMITATIONS

The prototype prioritises 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.
- **Operator study size.** Qualitative feedback currently spans 20 operators across two rotations, a useful but narrow slice that may not generalise beyond the sponsoring teams.
- **Scheduler evaluation.** Risk-bandit results rely on replayed telemetry and Monte Carlo queues rather than live incident queues; real deployments will need closed-loop validation with production workloads.
- **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.

```
1: Inputs: queue $Q$, risk $R_i$, KEV flag, wait time $\text{wait}_i$, 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 $= \dfrac{R_i \cdot p_i}{\max(\varepsilon, \mathbb{E}[t_i])}$; kev $= \kappa$ if KEV else 0; explore $= \beta\sqrt{\dfrac{\ln(1+n)}{1+n_i}}$;
       $S_i = \text{base} + \text{explore} + \alpha\, \text{wait}_i + \text{kev}$
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: $\text{wait}_i \leftarrow \text{wait}_i + \Delta t$
11: end while
```

Figure 2. Risk-Bandit scheduling loop (aging + KEV preemption) maximizing expected risk reduction per unit time with exploration and fairness.

VII. DISCUSSION AND FUTURE WORK

The current pipeline delivers 93.54% acceptance on the 5k supported corpus, 100.00% on the 1.264k supported slice, 100.00% on the 1.313k 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. The risk-aware scheduler trims top-risk P95 wait times from 102.3 h (FIFO) to 13.0 h. 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. Residual failures cluster around missing CRDs, namespaces, and controller-specific expectations; 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 continue surfacing human-in-the-loop evidence so the system graduates from course prototype to production-ready remediation service.

References

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

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

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

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

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

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

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

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

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

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

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

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

[15] SWE-bench Verified (background on closed-loop code repair evaluation). Accessed: 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.

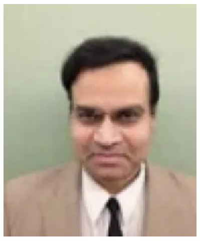
[18] M. De Jesus, P. Sylvester, W. Clifford, A. Perez, and P. Lama, "LLM-Based Multi-Agent Framework For Troubleshooting Distributed Systems," in *Proc. of the 2025 IEEE Cloud Summit*, 2025 (author's version).

[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/

[21] P. Hoffmann *et al.*, "Magpie: Python at Speed and Scale using Cloud Backends," arXiv:2103.16423, 2021; troubleshooting extensions accessed Oct. 2025. [Online]. Available: https://arxiv.org/abs/2103.16423

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, and has industry experience with Boeing (Air Force One design co-op), PepsiCo (supply-chain operations), Microchip Technology, and Autism Learning Foundation (web design/development). He received the B.E. in Mechanical Engineering (summa cum laude, GPA 3.99) from Arizona State University in 2021. His interests include secure configuration management for cloud-native systems, program analysis for infrastructure-as-code, and data-informed quality engineering.



VIJAY K. MADISETTI holds the position of Professor of Cybersecurity and Privacy (SCP) at Georgia Tech. He earned his PhD in Electrical Engineering and Computer Sciences from the University of California at Berkeley and is recognized as a Fellow of the IEEE. Additionally, he has been honored with the Terman Medal by the American Society of Engineering Education (ASEE). Professor Madisetti has authored several widely referenced textbooks on topics including cloud computing, data analytics, blockchain, and microservices.

• • •

Table 4. Acceptance and latency summary (seed 1337).

Corpus (mode)	Seed	Acceptance	Median proposer (ms)	Median verifier (ms)	Verifier P95 (ms)	Notes
Supported 1.264k (rules)	1337	1264/1264 (100.00%)	29.0	242.0	517.8	Host-mount policies normalised; measured from the seeded re-run.
Manifest 1.313k (rules)	1337	13589/13656 (99.51%)	5.0	77.0	178.4	Determini baseline for the manifest slice.
Manifest 1.313k (Grok/xAI)	1337	1313/1313 (100.00%)	—	—	—	Latest re-run succeeds across the slice; timing telemetry omitted in archived artifacts.
Grok-5k (Grok/xAI)	1337	4439/5000 (88.78%)	—	—	—	4.36M prompt + 0.69M completion tokens ($1.22) with guardrail-enabled batches.

Table 5. Guardrail example: Cilium DaemonSet patch (excerpt).

Before	After

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

Guardrails summarised in `docs/privileged_daemonsets.md`; the proposer preserves required host mounts while enforcing the hardened defaults.