CS8903 – OVM – Fall 2025

Closed-Loop Threat-Guided Auto-Fix for Kubernetes Misconfigurations

Brian Mendonca – bmendonca3@gatech.edu

Problem Statement

Static scanners for Kubernetes manifests routinely identify high-impact misconfigurations but rarely produce validated, minimal patches or prioritize remediation by threat. We propose a threat-guided, budget-aware, closed-loop auto-fixer targeting PSS/CIS-aligned policies (e.g., no privileged containers, runAsNonRoot, CPU/memory limits). A formal schema maps findings to severities (low/medium/high/critical) and policies (e.g., no_privileged). Closed-loop verification—policy re-check, schema validation, and server-side dry-run (with client-side precheck)—ensures fixes remove violations without introducing new ones.

Why This Problem?

Every minute a risky manifest remains unpatched extends exposure. The Kubernetes Pod Security Standards and CIS Benchmarks encode actionable guardrails; aligning auto-fixes to these rules makes impact measurable: a misconfig (dataset label) → detection (per-policy F1) → patch proposal → verified compliance. We quantify value with Auto-fix rate, No-new-violations %, and Time-to-patch, all on a hold-out set to prevent overfitting. ([Kubernetes][2], [CIS][1])

Is the Problem Meaningful?

Yes—meaning is evidenced by metrics tied to accepted baselines/standards. If Detection F1 ≥ 0.85 (hold-out) and ≥70% of detected violations are auto-fixed with ≥95% No-new-violations, teams can ship safer configs faster while spending less per successful fix (tokens/latency/steps). These thresholds map to the course Acceptance Targets and directly reflect “safer, cheaper, faster” remediation.

Is the Problem Novel?

We intentionally keep novelty incremental: (i) fuse two independent detectors (kube-linter + Kyverno/OPA) with (ii) a minimal-diff patcher and (iii) closed-loop gates (policy + schema + server-side dry-run with client precheck) plus (iv) budgeting and a threat-guided queue. Prior systems emphasize detection/admission; LLM patchers lack rigorous gates or budgets. Unlike LLM-driven code repair frameworks such as SWE-bench Verified, which focus on general software bug fixes, our approach integrates a simulated CTI feed to prioritize high-severity Kubernetes misconfigurations and enforces strict verification gates tailored to Kubernetes-specific policies. This targeted synthesis enhances reliability under acceptance criteria while optimizing for resource-constrained environments. ([docs.kubelinter.io][5], [Kyverno][3], [Open Policy Agent][4], [Kubernetes][8], [OpenAI][10])

Gaps in Existing Work

- Detection-only pipelines: kube-linter and policy engines flag issues but do not propose validated minimal patches. [docs.kubelinter.io][5]
- Lack of closed-loop verification: Few pipelines enforce a triad of checks—policy re-check, schema validation, and server-side dry-run—before accepting a patch. [Kubernetes][8]
- Cost/latency creep: Limited evidence on budget caps (attempts/tokens/time) versus quality trade-offs in YAML remediation workflows.
- Prioritization: Security work queues often run FIFO; we test a threat-guided queue for Time-to-patch improvements.
- Additional gaps include limited literature on tools like KubeSec, which focus on scanning without patch generation, and potential alignment with NIST 800-53 (CM, SI) families, to be explored.

Proposed Research Questions

- **RQ1 (Robustness):** Does the closed loop improve Auto-fix rate and No-new-violations % vs. detection-only and a naïve single-shot patcher? Metric/Design: On hold-out, compare closed-loop vs. {kube-linter+Kyverno detection-only, naïve LLM patcher}. Success if Auto-fix ↑ and No-new-violations ≥95% while meeting Patch validity ≥ threshold (Verifier pass rate).
- **RQ2 (Efficiency):** Budget caps reduce cost per successful fix (tokens/latency/steps) while Auto-fix rate and No-new-violations stay within ±2% of an unconstrained run (non-inferiority). Metric/Design: Cost/efficiency tracked per file; quality safeguarded within ±2%.
- **RQ3 (Throughput):** Does a threat-guided queue reduce Time-to-patch vs. FIFO and random? Metric/Design: Per-file and P95 queue Time-to-patch; report overall throughput (verified fixes/hour).
- **RQ4 (Patch Quality):** Are fixes minimal while passing all gates? Metric/Design: Diff minimality = median JSONPatch op count and normalized text diff (lines touched ÷ file lines); track no-op patch rate and idempotence (reapply patch ⇒ zero diff).
- **RQ5 (Scalability):** Does the system generalize across varied Kubernetes clusters? Metric/Design: Define 2-3 cluster profiles (e.g., different admission setups, API versions v1.25+); validate in future phases.

Risks

- Patch breakage / regressions: Mitigation: strict Verifier gates (policy re-check, schema, server-side dry-run) and a “no-new-violations” acceptance gate; prefer minimal diffs (JSONPatch). [RFC Editor][7]
- Overfitting to a toolchain: Mitigation: two detectors (kube-linter + Kyverno/OPA) and per-engine breakdowns in Detection F1. [docs.kubelinter.io][5], [Kyverno][3], [Open Policy Agent][4]

- Token/latency budgets exceeded: Mitigation: cap attempts (median ≤2 for successes), early-exit on first verified pass.
- Data leakage / provenance: Mitigation: fixed policy set, offline manifests, documented sources, no external solution mining; hold-out isolation; disable web tools, pin model (e.g., GPT-3.5).
- Adoption challenges: Mitigation: Design user-friendly outputs and validate with a subset (n≈20) of human-curated fixes for DevOps integration.

Acceptance Checklist

- Detection F1 ≥ 0.85 (hold-out)
- Auto-fix ≥ 70% of detected findings
- No-new-violations ≥ 95% (verifier triad + server-side dry-run)
- Median JSONPatch ops ≤ 3; idempotence ≥ 99%
- P95 Time-to-patch improves vs. FIFO under equal load
- Cost per successful fix down vs. unconstrained (±2% quality)

References

[1] CIS Kubernetes Benchmarks. https://www.cisecurity.org/benchmark/kubernetes

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

[3] Kyverno Documentation. https://kyverno.io/docs/

[4] OPA Gatekeeper How-to. https://open-policy-agent.github.io/gatekeeper/website/docs/howto/

[5] kube-linter Docs. https://docs.kubelinter.io/

[6] Kubernetes: Configure a Security Context. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[7] RFC 6902 (JSON Patch), DOI:10.17487/RFC6902. https://www.rfc-editor.org/info/rfc6902

[8] kubectl Command Reference (dry-run). https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

[9] Kubernetes: Seccomp and Kubernetes. https://kubernetes.io/docs/reference/node/seccomp/

[10] SWE-bench Verified (background on closed-loop code repair evaluation). https://openai.com/index/introducing-swe-bench-verified/