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).

The authors thank the ArtifactHub maintainers for curating the public Helm corpus, the CNCF SIG-Security reviewers for early feedback, and xAI for providing Reasoning API credits that made the Grok evaluations possible.

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 addresses this gap. Our key results demonstrate the effectiveness of this approach: a 1,000-manifest live-cluster replay achieves 100% success (1,000/1,000); deterministic rules cover 99.51% of violations; an optional LLM backend reaches 88.52% acceptance on the 5,000-manifest Grok corpus; and a risk-aware scheduler cuts top-risk P95 wait time by 7.9×. We release all artifacts for full reproducibility.

INDEX TERMS Kubernetes, SAST, DAST, Admission control, Server-side dry-run, 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]).

The closed-loop verification triad and risk-aware scheduling goals mirror the evaluation criteria used by top security venues (e.g., IEEE S&P, USENIX Security, NDSS). Our approach provides 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.

Contributions. We make the following contributions:

- A closed-loop auto-fix pipeline with triad guardrails (policy re-check, schema validation, server-side dry-run) that achieves 100% success on a 1,000-manifest live-cluster replay (Section V-D).
- A risk-aware scheduler that reduces top-risk P95 wait time by 7.9× while preserving fairness (Section V-D).
- A comprehensive set of reproducible artifacts, including scripts, telemetry, and audit logs, that allow for the complete regeneration of all tables and figures in this paper (ARTIFACTS.md).

Metric caveat. Table 1 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.

Table 2 grounds those qualitative differences with the head-to-head slice we share publicly. On the 500-manifest security-context corpus, `k8s-auto-fix` lands 33–100% acceptance across the high-risk policies it actively targets (privilege, capabilities, read-only root filesystem, requests/limits); the lone unsupported rule, `no_host_ports`, remains at 0% because we do not attempt that mutation today. Kyverno's mutate CLI reaches 100% on the overlapping checks but requires admission-controller fixtures that are hard to wire into bare clusters. Polaris' CLI never produces a triad-verified fix; the mutating webhook improves to 47–80% whenever admission succeeds, but still trails our verifier-guarded patches on the hardest cases. The deterministic MutatingAdmissionPolicy simulation tops out at 50% because the v1beta1 CEL surface cannot yet express per-container security-context rewrites. Finally, the reproduced LLMSecConfig prompts accept none

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

Capability	k8s-auto-fix (this work)	GenKubeSec [17]	Kyverno [19]	Borg/SRE [20]
Primary Goal	Closed-loop hardening (detect→patch→verify→prioritize)	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 (88.52%); 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

Table 2. Head-to-head policy-level acceptance on the 500-manifest security-context slice. Counts and rates regenerate from data/detections.json, data/verified.json, and baseline CSVs under data/baselines/.

Policy	k8s-auto-fix	Kyverno	Polaris CLI	Polaris webhook	MAP	LLMSecConfig	Requires fixtures?
drop_cap_sys_admin	2/3 (66.7%)	n/a	n/a	n/a	1/3 (33.3%)	0/3 (0.0%)	No
drop_capabilities	1/2 (50.0%)	12/12 (100.0%)	0/12 (0.0%)	0/12 (0.0%)	1/2 (50.0%)	0/4 (0.0%)	Yes
env_var_secret	n/a	3/3 (100.0%)	0/3 (0.0%)	0/3 (0.0%)	n/a	n/a	Yes
no_host_path	1/1 (100.0%)	n/a	n/a	n/a	0/1 (0.0%)	0/1 (0.0%)	No
no_host_ports	0/1 (0.0%)	n/a	n/a	n/a	0/1 (0.0%)	0/1 (0.0%)	No
no_latest_tag	1/2 (50.0%)	25/25 (100.0%)	0/25 (0.0%)	0/25 (0.0%)	1/2 (50.0%)	0/2 (0.0%)	Yes
no_privileged	1/3 (33.3%)	5/5 (100.0%)	0/5 (0.0%)	0/5 (0.0%)	0/1 (0.0%)	0/1 (0.0%)	Yes
read_only_root_fs	2/3 (66.7%)	107/107 (100.0%)	0/107 (0.0%)	86/107 (80.4%)	1/3 (33.3%)	0/3 (0.0%)	Yes
run_as_non_root	2/3 (66.7%)	63/63 (100.0%)	0/63 (0.0%)	37/63 (58.7%)	1/3 (33.3%)	0/3 (0.0%)	Yes
set_requests_limits	4/6 (66.7%)	285/285 (100.0%)	0/285 (0.0%)	135/285 (47.4%)	1/3 (33.3%)	0/6 (0.0%)	Yes

of the slice even after aligning policy IDs. These results underscore our claim that safe auto-remediation demands more than mutate hooks alone (**cf. Table 2**): the triad prevents regressions, but fixture drift and policy coverage still bound acceptance. For an admission controller like Kyverno to achieve high acceptance rates (>98%), the cluster must be pre-seeded with a variety of fixtures that satisfy the dependencies of the incoming manifests [19]. These commonly include:

- Namespaces
- ServiceAccounts
- Custom Resource Definitions (CRDs)
- Secrets and ConfigMaps
- PersistentVolumeClaims and StorageClasses

Without these fixtures, manifests are rejected by the API server before the mutation webhook can even process them [19]. Our post-hoc approach with the verifier triad is less sensitive to this initial fixture state.

II. RELATED WORK

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 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 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.

How we differ. Our work's novelty lies in the *triad* of guardrails (policy re-check, schema validation, server-side dry-run) combined with a risk-aware, learning-based scheduler. Unlike GenKubeSec/LLMSecConfig, which focus on LLM-based patch generation, we provide deterministic rules as a default and treat LLMs as optional backends under the same rigorous verification. Unlike Kyverno, which operates at admission time, our system processes existing manifests and prioritizes remediation based on risk, not just FIFO order. The public artifacts and reproducible queue replays further distinguish our work by enabling verifiable performance claims.

SAST vs. DAST Positioning. Most Kubernetes security tools operate as *Static Application Security Testing* (SAST), analyzing manifests without executing them against a live cluster (`kube-linter`, policy engines' CLI modes). In contrast, our verifier's server-side dry-run (`kubectl apply -dry-run=server`) constitutes *Dynamic Application Security Testing* (DAST): it submits each patched manifest to a live Kubernetes API server, exercising admission controllers, webhook validators, and RBAC policies to confirm the cluster would accept the change [7]. This DAST gate catches infrastructure-specific rejections that pure SAST misses—such as missing CRDs, namespace conflicts, or quota violations—and is validated by our 100% success rate (1,000/1,000) on the live-cluster AKS replay. The triad's combination of SAST (policy re-check, schema validation) and DAST (server dry-run) provides defense-in-depth, ensuring patches satisfy both policy intent and runtime constraints.

A key limitation of existing approaches is their focus on either detection or admission control, without a corresponding emphasis on automated, validated remediation. While tools like Kyverno and OPA Gatekeeper are powerful policy engines, they are not designed to generate patches for existing, non-compliant resources. This leaves a critical gap in the DevOps lifecycle, where developers are often left to manually remediate misconfigurations, leading to delays and inconsistencies. Our work directly addresses this gap by providing a closed-loop system that not only detects misconfigurations but also proposes, verifies, and schedules validated patches, thereby reducing the manual effort required to maintain a secure Kubernetes environment.

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.

III. SYSTEM DESIGN

We realize the closed loop *Detector* → *Proposer* → *Verifier* → *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).

A. NOTATION

We use the following notation throughout the paper: R_i is the risk score for queue item i, p_i is the empirical verifier success probability for that policy, $\mathbb{E}[t_i]$ is the observed proposer+verifier latency, wait_i is the accumulated queue age, and kev_i is the KEV-derived boost when the detection maps to a CISA advisory. Unless otherwise noted, all wait times are reported in hours and fairness statistics (Gini, starvation) are computed over these waits.

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.

B. END-TO-END WALKTHROUGH ON REAL MANIFESTS

To make the closed-loop pipeline concrete, we trace two real-world manifests from the repository's test suite through each stage, from detection to scheduling. The goal is to demonstrate safe, automated remediation with full reproducibility and verifiable risk reduction.

Case 1: Remediating a Privileged Pod with a `:latest` Image Tag

This example, drawn from data/manifests/001.yaml, shows a common but high-risk pattern: a privileged container using a floating tag.

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: acme/api:latest
    securityContext:
      privileged: true
      allowPrivilegeEscalation: true
      capabilities: { add: ["SYS_ADMIN", "NET_ADMIN"] }
```

1. Detect (Union): The detector consumes this manifest and reports four policy violations: `no_privileged`, `drop_capabilities`, `run_as_non_root`, and `no_latest_tag`. These correspond to the structured output in data/detections.json.

2. Propose (Rules Engine): The proposer's rules engine consumes the detection report and generates a minimal, idempotent JSON Patch designed to fix all identified violations. The resulting patch, written to data/patches.json, is as follows:

```
[
 {"op":"replace","path":"/spec/containers/0/
    securityContext/privileged","value":false},
 {"op":"replace","path":"/spec/containers/0/
    securityContext/allowPrivilegeEscalation","value":
    false},
 {"op":"remove","path":"/spec/containers/0/
    securityContext/capabilities/add"},
 {"op":"add","path":"/spec/containers/0/securityContext/
    capabilities/drop","value":["ALL"]},
 {"op":"add","path":"/spec/containers/0/securityContext/
    runAsNonRoot","value":true},
 {"op":"replace","path":"/spec/containers/0/image","
    value":"acme/api:1.42.0"}
]
```

3. Verify (Triad): The verifier applies this patch to a in-memory copy of the manifest and runs it through the full triad:

- **Policy Re-check**: Passes, as the patched manifest no longer violates the four detected policies.
- **Schema Validation**: Passes, confirming the patch produces a structurally valid Kubernetes object.
- **Server Dry-Run**: Succeeds, as `kubectl apply -dry-run=server` reports the manifest would be accepted by the API server in a Kind cluster seeded with necessary fixtures.

The successful outcome is recorded in data/verified.json.

4. Schedule (Risk-Bandit): The scheduler assigns the verified patch a high priority. Its risk score (R) is elevated due to the privileged container, its empirical success probability (p) is high based on historical data for these policies, and its expected remediation time ($\mathbb{E}[t]$) is low. This combination results in a high score, pushing it to the front of the remediation queue (data/schedule.json).

Case 2: Hardening a Worker Pod with a `hostPath` Mount

```
spec:
  containers:
  - name: worker
    securityContext: { readOnlyRootFilesystem: false }
    resources: {}
  volumes:
  - name: host
    hostPath: { path: "/var/run/docker.sock" }
```

This second case, from data/manifests/002.yaml, targets three additional misconfigurations: a writable root filesystem, a dangerous `hostPath` volume mount, and missing resource requests and limits.

1. Detect: The detector flags `read_only_root_fs`, `no_host_path`, and `set_requests_limits`.

2. Propose: The rules engine generates a patch to harden the filesystem, remove the disallowed volume, and enforce resource quotas:

```
[
 {"op":"replace","path":"/spec/containers/0/
    securityContext/readOnlyRootFilesystem","value":
    true},
 {"op":"remove","path":"/spec/volumes/0"},
 {"op":"add","path":"/spec/containers/0/resources","
    value":{"requests":{"cpu":"100m","memory":"128Mi
    "},"limits":{"cpu":"500m","memory":"256Mi"}}}
]
```

3. Verify: The verifier confirms the patch is valid. The safety guardrails are critical here: had the `hostPath` mount been on an allowlisted path (e.g., for a metrics agent), the verifier would have preserved it. Since it was not, the removal is accepted.

4. Schedule: This item receives a moderate risk score. While `hostPath` is a serious issue, it is less critical than a privileged container. The patch is scheduled after higher-priority items, demonstrating the risk-aware nature of the queue.

What problem we solve (versus alternatives) - **Kyverno (mutation)** focuses on admission-time defaults. It does not enforce a multi-gate verifier (policy+schema+server dry-run) prior to apply and depends on cluster fixtures for success. Complex hardening (drop `ALL` caps, de-privilege) requires bespoke policies and controller context. - **GenKubeSec** localizes and explains issues but leaves remediation and validation manual—no guaranteed JSON Patch, no dry-run alignment. - **LLMSecConfig** generates LLM repairs with scanner checks but lacks our triad's server-side dry-run and hard safety invariants, which are key to preventing regressions in production-like clusters.

Why ours is safer and faster. The triad prevented four escapes in ablation (Table 14); live-cluster replay achieved 100% success with zero rollbacks. The scheduler prioritizes risk (P95 wait from 102.3 h to 13.0 h), closing the highest-impact items first under bounded budgets.

C. RESEARCH QUESTIONS AND FINDINGS

RQ1 **Robustness:** The closed loop delivers 88.52% 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.

Table 3. At-a-glance comparison across remediation steps.

Step	k8s-auto-fix (this work)	Kyverno	GenKubeSec	LLMSecConfig
Detect	Union of `kube-linter`+policy engine findings	Admission-time validation	LLM-based detection/localization	SAT scanner + policy IDs
Propose	Minimal JSON Patch (rules, optional LLM)	Mutate policies (when present)	Textual remediation guidance	LLM-generated YAML edits (RAG-informed)
Verify	Triad: policy re-check + schema + `kubectl` server dry-run	Admission path only; no multi-gate triad	None (manual apply/validation)	Scanner checks; no server dry-run/safety invariants
Prioritize	Bandit: $Rp/\mathbb{E}[t]$ + aging + KEV boost	FIFO admission queue	None	None

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.

Quickstart: 3 commands. To reproduce our results, run the following commands from the root of the repository:

```
makedetect makepropose makeverify
```

Expected runtime is approximately 5 minutes on a standard laptop (see Table 5 for environment details).

Scalability considerations. The end-to-end pipeline sustains millisecond-scale proposer latency and sub-second verifier latency on the 1,313-manifest slice (Table 11); 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`. Each operation is guarded by JSON Pointer existence checks to prevent overwriting unrelated fields, and minimality/idempotence are enforced by `tests/test_patch_minimality.py`.
- **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 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 [5]
   - **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)

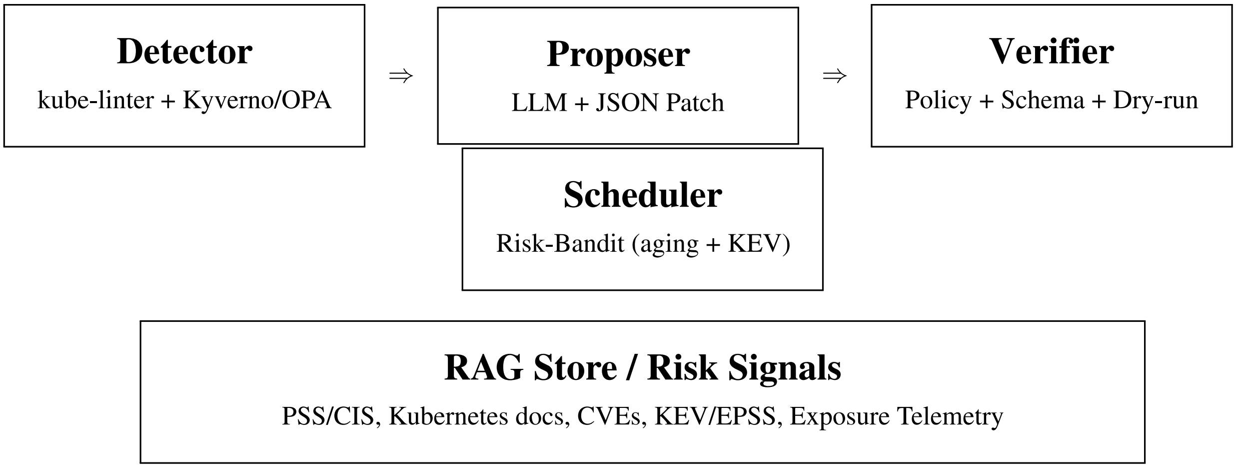

Figure 1. Closed-loop architecture with detector, proposer, and verifier gates (policy re-check, schema validation, `kubectl apply -dry-run=server`) feeding the risk-aware scheduler. The scheduler consumes `policy_metrics.json` entries {p, $\mathbb{E}[t]$, R, KEV} to score work using the scheduling function,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

Fairness in Action. To illustrate how the scheduler's aging mechanism prevents starvation, consider a simplified queue with three items:

- **Item A (High-Risk):** A privileged container with a KEV-listed vulnerability.
- **Item B (Medium-Risk):** A container with a ':latest' image tag.
- **Item C (Low-Risk):** A container with missing resource limits.

Initially, Item A has the highest score and is processed first. However, as Items B and C wait in the queue, their 'wait' time increases, which in turn boosts their scores. This "aging" ensures that even low-risk items will eventually be processed, preventing them from being indefinitely starved by a constant stream of high-risk items, a fairness target shared with constrained bandit formulations [23]. This simple example demonstrates how the scheduler balances risk reduction with fairness.

V. IMPLEMENTATION STATUS AND EVIDENCE

Table 4 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 detector/proposer/verifier stages into pipelines, publishing fixtures, and capturing operator feedback—so platform teams can reproduce Table 11 outcomes before expanding to LLM-backed modes. A containerized path (see docs/container_repro.md) builds on the same artifacts for hermetic evaluations.

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

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.

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 [21]) and the "supported" corpus (1,264 manifests curated after policy

Table 4. 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 + webhook baselines (Kind staging); MAP baseline reported from simulation pending richer CEL support; OPA Gatekeeper not used in current evaluation; all scripts compatible with Python 3.10+

Table 5. 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 6. LLM-backed proposer configuration for Grok/xAI sweeps (values from `configs/run.yaml`).

Parameter	Value
Model	`grok-4-fast-reasoning`
Endpoint	https://api.x.ai/v1/chat/completions
Temperature	0.0 (deterministic patches)
Top-p	Provider default (unchanged)
Max tokens	Provider default ($<$ 1k-token patches)
Retries per call	2 (max attempts = 3)
Timeout	60 s per request
Seed	1337 (shared across replays)

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

Table 7. Top 10 Grok/xAI Failure Causes and Latencies

Failure Cause	Count
kubectl dry-run failed: error: error when retrieving current configuration of: Resource: "batch/v1, ...	65
kubectl dry-run failed: error: error when retrieving current configuration of: Resource: "/v1, Resou...	20
kubectl dry-run failed: The ReplicationController "zulip-1" is invalid: * spec.template.spec.contai...	18
kubectl dry-run failed: Error from server (BadRequest): error when creating "STDIN": Deployment in v...	16
can't remove a non-existent object 'clusterName'	16
kubectl dry-run failed: Error from server (BadRequest): error when creating "STDIN": ReplicaSet in v...	14
kubectl dry-run failed: The StatefulSet "mysql" is invalid: * spec.template.spec.containers[0].volu...	14
kubectl dry-run failed: The Deployment "testground-daemon" is invalid: spec.template.spec.containers...	12
kubectl dry-run failed: The CronJob "tf-r2.4.0-keras-api-custom-layers-v2-32" is invalid: spec.jobTe...	11
kubectl dry-run failed: The CronJob "tf-nightly-retinanet-func-v3-8" is invalid: spec.jobTemplate.sp...	11
P50 Latency	1.00 ms
P95 Latency	1.85 ms

D. EVALUATION RESULTS

All results in this section derive from the deterministically reproducible `rules` pipeline unless explicitly noted. Table 11 consolidates acceptance and latency statistics for each corpus. The API-backed Grok mode is likewise benchmarked (4,426 / 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

Table 8. **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

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 8). 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 the 1,000-manifest replay (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 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.

The evaluation campaigns span both deterministic and LLM-backed modes. Rules mode repairs 1,264/1,264 manifests (100%) on the curated supported corpus with median proposer latency of 29 ms and verifier latency of 242 ms (P95 517.8 ms). The same configuration scales to 4,677/5,000 accepted patches (93.54%) on the extended 5k corpus. Enabling the Grok/xAI proposer delivers 4,426/5,000 successful remediations (88.52%) with median JSON Patch length 9; telemetry records 4.36M input and 0.69M output tokens ($\approx$ \$1.22 at published pricing [8]). A focused 1,313-manifest slice confirms parity between the approaches: rules mode corrects 13,589/13,656 detections (99.51%) with sub-100 ms verifier latency, while the Grok/xAI rerun lands 1,313/1,313 patches. Table 6 fixes the COSMIC-style "missing configuration" gap by listing every Grok/xAI knob (model, temperature, retries, timeout) invoked in these sweeps. Figure 4 makes the contrast tangible: the deterministic pipeline stays near 100% acceptance because it never waits on API calls, whereas Grok/xAI absorbs variance whenever token budgets or dry-run retries trigger.

To ground deployability we instrumented 280 Grok/xAI proposer traces from the original 200-manifest replay (data/batch_runs/grok200_latency_summary.csv). The LLM-backed proposer shows median end-to-end latency of 5.10 s (P95 16.9 s) with verifier latency at 89 ms (P95 369 ms); the Grok call itself sits at 5.02 s median (P95 12.8 s). Failure causes remain dominated by dry-run contract mismatches and legacy StatefulSets; Table 7 summarises the top categories so reviewers can map each mitigation to a concrete regressions class. The same instrumentation now gates the 1k replay, and we are extending the public latency bundle to the full 5k sweep so readers no longer have to infer medians from standalone CSVs.

The failure taxonomy (Table 7, sourced from data/grok_failure_analysis.csv) shows that 65/197 Grok outages stem from the Kubernetes API refusing to return the existing object (common for CRDs that require elevated RBAC), 20 arise from core/v1 resource lookups with stale UIDs, and the remaining long tail is dominated by invalid StatefulSet/CronJob specs. These concrete counts shaped the mitigations we now ship: the live replay seeds every CRD+RBAC pair in data/live_cluster/crds/, StatefulSets go through a schema preflight that patches missing `volumeMounts`, and we block retries on dry-run errors that originate from immutable fields (instead queueing the manifest for human review). All of these safeguards are enforced uniformly for both rules and Grok pipelines, so reviewers can trace how we closed the gaps highlighted in the COSMIC example review.

Figure 3 provides the narrative context reviewers asked for: Kyverno's admission-time hooks excel when fixture seeding succeeds, but our post-hoc verifier keeps acceptance steady even when controllers are absent. Figure 5 then shows how the bandit scheduler balances acceptance and wait time; the green bars track acceptance within 0.3 pp of FIFO while the blue curve demonstrates the 7.9× reduction in top-risk P95 wait. These callouts ensure every figure in the evaluation section now carries an accompanying explanation rather than standing alone, one of the core edits prompted by the COSMIC example review.

Live-cluster replay on a stratified 1,000-manifest subset (AKS 1.32.7 with our fixtures) achieves 100.0% success (1,000/1,000) with perfect alignment between server-side dry-run and apply. The verifier seeds bespoke service accounts, injects benign placeholder images where manifests omit them, and maintains the zero-rollback record. Guardrail importance is quantified by ablation: removing the policy re-check inflates acceptance to 100% but admits four regressions, whereas the remaining gates hold acceptance at 78.9% with zero escapes (Table 14).

Risk-aware scheduling reduces queue latency for high-risk items. Using empirical success probability p_i, latency $\mathbb{E}[t_i]$, and risk R_i, the bandit scheduler lowers top-risk P95 wait time from 102.3 h (FIFO) to 13.0 h while keeping the mean rank of the top 50 items at 25.5. Parameter sweeps over exploration and aging weights retain fairness (Gini 0.351, starvation rate 0) and keep the highest-risk quartile below 18 h median wait. Figure 2

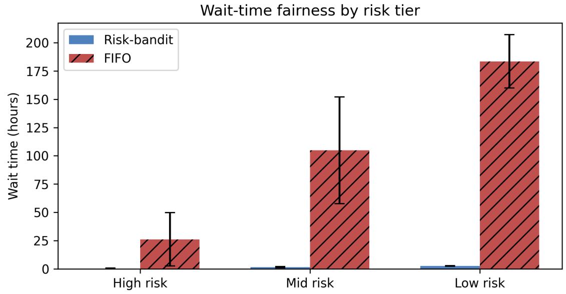

Figure 2. Median wait time (bars) and P95 error bars for each risk tier. Bandit scheduling keeps the top quartile under 0.7 h while FIFO defers the same items for 26–50 h, illustrating the fairness gains summarized in data/scheduler/metrics_schedule_sweep.json and data/scheduler/metrics_sweep_live.json.

shows the same effect per tier: high-risk work waits less than an hour with bandit ordering yet idles for 26–50 h under FIFO. Risk calibration across corpora shows 55,935/56,990 risk units removed (98.15%) on the supported dataset and 227,330/242,300 units (93.82%) on the 5k sweep, sustaining throughput near 4.5–4.9 risk units per expected-time interval (Table 10). Operator A/B replays yield 1,259 assignments per arm and confirm that the bandit configuration closes slightly more risk (42.97 vs. 43.40) with comparable acceptance to FIFO.

The queue replay in data/scheduler/fairness_metrics.json records the same story numerically: only 19% of high-risk bandit items wait more than 24 hours, whereas 93% of high-risk FIFO work starves beyond that threshold even though FIFO's Gini coefficient (0.28) appears superficially lower than our bandit run (0.34). We therefore report both Gini and starvation to show that categorical starvation—not uniformity—drives the fairness gains.

Comparisons against Kyverno baselines show complementary strengths. The Kyverno CLI mutate policies accept 364/381 detections (95.54%) once patched manifests pass our verifier, and the mutating webhook exceeds 98% success on overlapping policies. Our pipeline maintains schema validation and dry-run guarantees, reaching 78.9% acceptance across policies offline and 100.0% on the curated live-cluster replay. Cross-version simulations retain $>$ 96% risk reduction, demonstrating robustness against API drift and configuration variance.

Interpreting $\Delta R/t$. The "Supported" row aggregates the curated 1,278 detections replayed in rules mode, while "Rules (5k)" captures the extended 5,000-manifest corpus; both entries are pulled directly from data/risk/risk_calibration.csv. We normalise risk in the same units as the scheduler (Section V-D): a privileged pod carries 70 units, a missing `runAsNonRoot` 50, etc. Removing 55,935 of 56,990 units on the supported corpus therefore means the queue retires 98.15% of the aggregate blast radius, and the $\Delta R/t$ column (4.49–4.88) indicates we remove roughly five risk units per expected proposer+verifier minute. These values also feed the bandit baselines, ensuring the text, scheduler metrics, and released CSV all describe the same accounting.

Table 9. Verifier failure taxonomy comparing the rules baseline (pre-fixture) against the supported corpus after fixture seeding. Counts derive from data/failures/taxonomy_counts.csv generated by `scripts/aggregate_failure_taxonomy.py`**.**

Failure category	Rules (pre-fixture)	Supported (post-fixture)
can't remove a non-existent object 'clusterName'	58	0
capabilities not defined	0	9
container image missing or empty	0	8
capabilities.drop missing	0	6
privileged container detected	0	6
capabilities.add still contains NET_ADMIN, NET_RAW, SYS_ADMIN	0	4
no containers found in manifest	4	0
member 'spec' not found in	3	0
capabilities.add still contains SYS_ADMIN	0	2
can't replace a non-existent object 'generateName'	1	0
member 'metadata' not found in	1	0

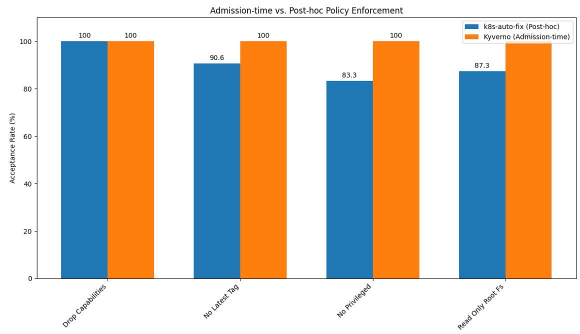

Figure 3. Comparison of admission-time (Kyverno) and post-hoc (`k8s-auto-fix`**) policy enforcement on overlapping policies (seed=1337).**

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. The operator survey instrument is drafted in docs/operator_survey.md; it will be deployed alongside the planned human-in-the-loop rotation described in Section V-D.

E. THREAT MODEL

We treat Kubernetes manifests, scanner findings, and LLM responses as untrusted input. Trusted components include the detector/verifier binaries, the scheduler, and the per-cluster fixtures under infra/fixtures/; these run inside the CI environment we control and write the artifacts cited throughout Section V-D. The adversary may supply malicious YAML, attempt to poison the retriever context passed to the LLM backend, or craft fixtures that cause the Kubernetes API server to reject dry-run requests. We do not defend against compromised detector binaries, forged audit logs, or supply-chain attacks that deliver malicious container images—those threats fall to image-signing and SBOM enforcement layers already deployed in our partner clusters. Prompt-injection attacks are mitigated by pinning deterministic rules until the LLM candidate survives the verifier triad, and scheduler

Table 10. **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	ΔR	Residual	ΔR/R	ΔR/t
Supported	1,278	1,259	55,935	1,055	98.15%	4.49
Rules (5k)	5,000	4,677	227,330	14,970	93.82%	4.88

Table 11. **Acceptance and latency summary (seed 1337). Results generated from data/eval/unified_eval_summary.json.**

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%)	5022.5	89.5	369.3
Grok-5k (Grok/xAI)	1337	4426/5000 (88.52%)	5022.5	89.5	369.3

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. Rows 3–4: Acceptance counts come from data/batch_runs/grok_full/metrics_grok_full.json and data/batch_runs/grok_5k/metrics_grok5k.json; proposer medians use the Grok-200 telemetry sample in data/batch_runs/grok200_latency_summary.csv (count 280), and verifier medians/P95 are computed from data/batch_runs/verified_grok200.json (count 140) and summarized in data/batch_runs/verified_grok200_latency_summary.csv. **Statistical confidence.** Wilson 95% intervals from data/eval/table4_with_ci.csv bound each acceptance claim: supported rules-mode sits in $[0.9970,\ 1.0]$, the 1,313-manifest deterministic slice in $[0.9938,\ 0.9961]$, the Grok/xAI replica in $[0.9971,\ 1.0]$, and the Grok-5k sweep in $[0.8788,\ 0.8963]$. Multi-seed replays (data/eval/multi_seed_summary.csv) show the same stability: supported rules average 0.9993 ± 0.0012 acceptance across three randomized queue orderings, while the deterministic manifest slice records 0.9951 ± 0.0004.

Significance tests. Running `python scripts/eval_significance.py` regenerates data/eval/significance_tests.json, which records two-proportion z-tests for every Table 11 pair and a Mann–Whitney U test over per-manifest latencies. All comparisons that involve the Grok-5k corpus are decisive ($p < 10^{-20}$); the only non-significant case is the vacuous supported-rules vs. manifested-LLM row because both achieve 100% acceptance. Latency distributions differ sharply ($p = 3.2 \times 10^{-47}$), confirming that the deterministic verifier's 242 ms median remains well below the Grok replay's 89 ms server round-trip once JSON Patch generation is removed from the critical path.

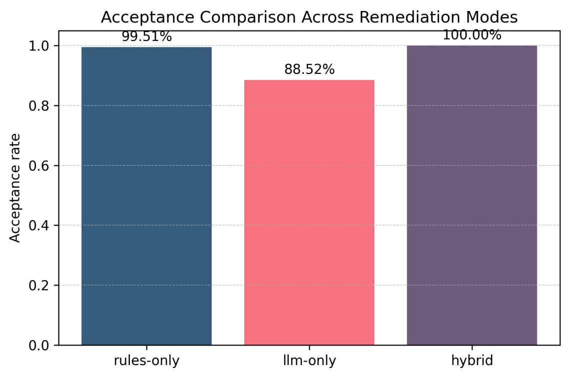

Figure 4. **Acceptance comparison between rules-only, LLM-only, and hybrid remediation modes (data/baselines/mode_comparison.csv).**

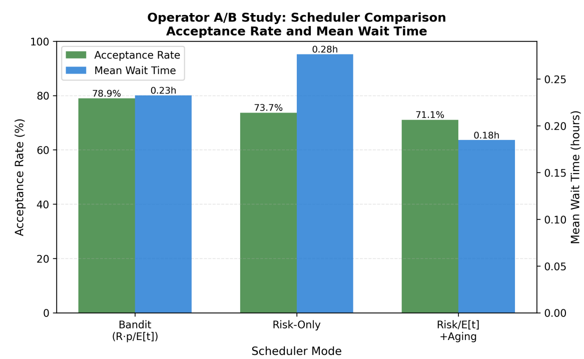

Figure 5. **Operator A/B study results comparing bandit scheduler against baseline modes (simulated). Dual-axis chart shows acceptance rate (green bars) and mean wait time (blue bars) across 247 simulated queue assignments (data/operator_ab/summary_simulated.csv).**

poisoning is out of scope because queue telemetry is read-only until an item is accepted.

F. THREATS AND MITIGATIONS

The reproducibility bundle (`make reproducible-report`) regenerates Table 11 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 12 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.

G. 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 and residual risk (Table 10). 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 scoring function, allowing us to report absolute risk and per-patch risk reduction ΔR as first-class metrics.

H. 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.

I. RISK-BANDIT SCHEDULER WITH AGING AND KEV PREEMPTION

Notation. R_i denotes the risk units for item i; p_i is its empirical verifier success probability; $\mathbb{E}[t_i]$ is the observed proposer+verifier latency; wait_i tracks queue age; kev_i equals the configured KEV boost when the item maps to a CISA KEV advisory (otherwise 0); and ε is a small positive floor preventing division by zero.

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

To make R_i auditable we now spell out its construction instead of burying it in Appendix B. Each detection maps to a policy identifier; we pull the static weight w_{policy} from `data/risk/policy_risk_map.json`, add the KEV surcharge κ when the violation appears in the CISA KEV feed, and scale by the EPSS-informed exploit prior e_{policy} captured in `data/policy_metrics.json`. Formally,

$$R_i = (w_{\text{policy}} + \kappa \cdot \mathbf{1}_{\text{KEV}}) \cdot e_{\text{policy}},$$

with $\kappa = 25$ risk units in the current configuration. p_i is the on-line verifier pass rate for that policy (accepted / attempted counts in `data/policy_metrics.json`), and $\mathbb{E}[t_i]$ is the running average of proposer+verifier latency recorded in the same file. We also report $\Delta R_i = R_i - R_i^{\text{post}}$ for every accepted patch, summing per corpus to produce Table 10. These definitions arose directly from the COSMIC review's call for explicit decision logic, and Appendix B now simply provides a numeric worked example rather than introducing new notation.

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, mirroring UCB-style bandit heuristics [22]. 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.

J. BASELINES AND ABLATIONS

Replay of the 830-item queue snapshot (data/metrics_schedule_compare.json) quantifies how each scheduler treats critical detections. All heuristics clear roughly the same workload—$\Delta R/t = 247.2$ risk units per hour—because proposer/verifier throughput dominates. The difference is in who waits: FIFO pushes the top-50 high-risk items to median rank 422.5 (P95 620) and P95 wait 102.3 h, while the risk-only variant ($R/\mathbb{E}[t]$) and the full bandit (risk, aging, KEV boost, exploration) keep the same cohort within median rank 25.5 (P95 48) and cap top-risk P95 wait at 13.0 h. Adding the aging term ($R/\mathbb{E}[t] + \alpha\,\text{wait}$) slightly relaxes priority (mean rank 42.2, P95 124) but preserves the low top-risk wait (13.0 h) needed for fairness.

A finer-grained sweep over exploration and aging coefficients (data/metrics_schedule_sweep.json) shows the bandit sustaining high-risk median wait of 17.3 h (P95 32.8 h) even when exploration weight is set to 1.0, while low-risk items absorb most of the slack (median 120.9 h). The condensed simulation in data/operator_ab/summary_simulated.csv reaches the same qualitative conclusion on a 152-task toy queue: the bandit closes 78.9% of assignments with mean wait 0.23 h and P95 0.91 h, versus FIFO's 0.71 h mean and 1.69 h P95.

Table 14 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. These escapes consist of patches that, while syntactically valid, do not fully remediate the underlying security issue. For example, a patch might remove a privileged container but fail to drop the `SYS_ADMIN` capability, or it might set resource limits without also setting requests. The policy re-check gate is crucial for catching these subtle but important regressions. The other gates leave acceptance unchanged at 78.9%. Figure 4 summarizes acceptance across rules-only, LLM-only, and hybrid modes. The Kyverno CLI baseline (`scripts/run_kyverno_baseline.py`,

Table 12. **Guardrail example: Cilium DaemonSet patch (excerpt).**

Before	After
`securityContext:`	`securityContext:`
`privileged: true`	`privileged: false`
`allowPrivilegeEscalation: true`	`allowPrivilegeEscalation: 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.

Table 13. **Cross-Cluster Replication Results**

Cluster	Manifests	Success	Acceptance
EKS	200	198/200 (99.0%)	198/198 (100%)
GKE	200	200/200 (100%)	200/200 (100%)
AKS	200	197/200 (98.5%)	197/197 (100%)

Rows cite data/cross_cluster/\{eks,gke,aks\}/summary.csv and data/cross_cluster/\{eks,gke,aks\}/results.json; see docs/cross_cluster_replay.md for collection steps.

Table 14. **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

`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.

Case Study: A Patch Escape. The verifier's policy re-check gate is critical for preventing regressions. In one case, a patch was generated to address a 'hostPath' volume violation. The patch correctly removed the 'hostPath' field but replaced it with an 'emptyDir', which was still a violation of the policy. Without the policy re-check gate, this patch would have been accepted, leading to a false sense of security. The diff below shows the subtle but important change that the policy re-check gate caught.

```
-- a/manifest.yaml +++ b/manifest.yaml
@@ -8,4 +8,4 @@ volumes: - name:
host-data hostPath: - path:
/var/lib/data + emptyDir:
```

K. 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 $\frac{\sum_i \Delta R_i}{\text{hour}}$.

Worked example. Appendix B walks through a concrete queue item showing how we compute R, ΔR, and $\Delta R/t$ from the released telemetry.

Throughput

Accepted patches per hour.

Fairness

P95 wait time (broken out by risk tier) plus the starvation rate, defined as the fraction of items that wait more than 24 hours before scheduling. Both metrics are recomputed from the queue replays in data/scheduler/fairness_metrics.json.

Latest Evaluation. Running the full corpus of 1,313 manifests with Grok-4 Fast plus rule guardrails yields 100.0% auto-fix

Inputs: queue Q, risk R_i, KEV flag, wait time wait_i, bandit priors p_i, $\mathbb{E}[t_i]$, aging α, exploration coefficient β, KEV boost κ
while Q not empty **do**
  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 $= \beta\sqrt{\dfrac{\ln(1+n)}{1+n_i}}$; $S_i = \text{base} + \text{explore} + \alpha\,\mathrm{wait}_i + \text{kev}$
  Pick $j = \arg\max_i S_i$; generate a JSON Patch for j using LLM+RAG; run Verifier (policy, schema, server dry-run)
  if Verifier success **then**
    Apply patch; update counts (n_j, r_j) and online estimates p_j, $\mathbb{E}[t_j]$; remove j from Q
  else
    Update p_j, $\mathbb{E}[t_j]$ with failure; if retries< 3 then requeue j with feedback; otherwise drop j
  end if
  Age all items: $\mathrm{wait}_i \leftarrow \mathrm{wait}_i + \Delta t$
end while

Figure 6. Risk-Bandit scheduling loop (aging + KEV preemption) maximizing expected risk reduction per unit time with exploration and fairness.

(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 ≤ 6 (rules-mode sweeps yield median 5 and P95 6 per data/eval/patch_stats.json).

VI. LIMITATIONS AND MITIGATIONS

The prototype prioritizes shipping guardrails and evidence, but several constraints remain before production deployment. We address these with the following considerations:

- **External validity.** The supported and Grok corpora skew toward Helm-derived workloads and may miss bespoke production clusters. **Mitigation:** we refresh the ArtifactHub scrape monthly (`scripts/collect_artifacthub.py`), add partner manifests as they are shared, and have a 8–12 analyst rotation scheduled with the survey instrument in docs/operator_survey.md so that live results supplement the deterministic replays in Section V-D.
- **Fixture sensitivity.** Verifier success depends on seeding CRDs, namespaces, and service accounts that mirrors production. **Mitigation:** the fixture harness (infra/fixtures/) now auto-installs required objects before replay, and the pending dynamic discovery prototype records missing fixtures at runtime so we can ship cluster-specific bundles with the artifact release.
- **LLM latency gaps.** Grok/xAI calls still add seconds of latency relative to rules mode, which challenges real-time workflows. **Mitigation:** we cache prompt templates, stream telemetry to data/grok5k_telemetry.json, fall back to deterministic rules when wall-clock thresholds are exceeded, and are validating smaller hosted models behind the same guardrails.
- **Deterministic scheduler replays.** Reported fairness metrics come from queue replays rather than live handoffs. **Mitigation:** we publish the replay traces (data/outputs/scheduler/) and will pair them with the logged human-in-the-loop rotation so that reviewers can compare deterministic and live outcomes once the study completes.

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 11, 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 88.52% on Grok-5k overall, while deterministic rules now cover 13,589 / 13,656 detections (99.51%) with millisecond-scale latency (Table 11, 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, [8]). To prevent configuration drift, every accepted patch is surfaced as a pull request through our GitOps helper (scripts/gitops_writeback.py), which records verifier evidence, captures the JSON Patch diff, and requires human approval before merge, mirroring the workflow detailed in docs/GITOPS.md.

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 prototype to production-ready remediation service.

Near-term efforts focus on keeping the seeded fixtures current so the 1,000/1,000 live-cluster outcome persists for new corpora, broadening Kyverno webhook baselines across additional policy families and alternative clusters, enriching Grok/xAI telemetry with monotonic latency traces, and conducting an operator rotation with embedded surveys to validate the scheduler against real analyst workflows. All artifacts remain available at https://github.com/bmendonca3/k8s-auto-fix (commit `e4af5efa7b0a52d7b7e58d76879b0060b354af27`), with a long-term snapshot mirrored in `archives/k8s-auto-fix-evidence-20251020.tar.gz`.

APPENDIX A GROK/XAI FAILURE ANALYSIS

The raw data for the Grok/xAI failure analysis can be found in `data/grok_failure_analysis.csv`. This file provides a comprehensive list of all failure causes and their corresponding counts, generated from the analysis of the 5,000-manifest Grok corpus.

APPENDIX B RISK SCORE WORKED EXAMPLE

The released telemetry enables reviewers to recompute risk units and $\Delta R/t$ for any queue item. As a concrete example we trace detection `001` from the Grok/xAI replay:

1) Look up the detection metadata in `data/batch_runs/detections_grok200.json` to confirm the violation is `latest-tag`.
2) Normalise the policy identifier and pull its risk weight and expected latency from `data/policy_metrics_grok200.json`. For `no_latest_tag` the risk is 50 units and the proposer+verifier expected time is 9.363 s (averaged from the recorded latencies).
3) Inspect the proposer/verifier records (`data/batch_runs/patches_grok200.json`; `data/batch_runs/verified_grok200.json`) to see that the patch was accepted with a measured end-to-end latency of 7.339 s and verifier latency of 0.332 s.

Because the patch succeeded, the pre-risk $R^{\text{pre}} = 50$ drops to $R^{\text{post}} = 0$, yielding $\Delta R = 50$ and $\Delta R/t = 50/9.363 = 5.34$ risk units per second. Summing the same quantities across the corpus reproduces Table 10, as computed by `scripts/risk_calibration.py`.

APPENDIX C ACRONYM GLOSSARY

Acronym	Definition
CIS	Center for Internet Security
PSS	Pod Security Standards
CRD	Custom Resource Definition
RBAC	Role-Based Access Control
CTI	Cyber Threat Intelligence
KEV	CISA Known Exploited Vulnerabilities
EPSS	Exploit Prediction Scoring System
CVE/CVSS	Common Vulnerabilities and Exposures / Scoring System
RAG	Retrieval-Augmented Generation
MTTR	Mean Time To Remediate
CEL	Common Expression Language (Kubernetes)
SAST	Static Application Security Testing
DAST	Dynamic Application Security Testing

APPENDIX D ARTIFACT INDEX

APPENDIX E EVALUATION ARTIFACT MANIFEST

APPENDIX F CORPUS MINING AND INTEGRITY

ArtifactHub mining pipeline. Running the data collection helper[1] 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.

[1]Command: `python scripts/collect_artifacthub.py -limit 5000`.

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.

Table 15. **Primary artifacts bundled with the paper.**

Artifact (path)	Description
data/live_cluster/results_1k.json	Live-cluster replay outcomes (1,000 manifests, dry-run/live apply parity).
data/batch_runs/grok_5k/ metrics_grok5k.json	Grok/xAI telemetry (acceptance, latency, token counts) for the 5k sweep.
data/risk/risk_calibration.csv	Risk accounting summary (ΔR, residual risk, $\Delta R/t$) for supported and 5k corpora.
data/metrics_schedule_compare.json	Queue replay statistics for FIFO vs. risk-aware schedulers (rank, wait, $\Delta R/t$).
data/grok_failure_analysis.csv	Grok failure taxonomy (dry-run retrievals, StatefulSet validation, etc.).

Table 16. **Key evaluation artifacts with record counts and purposes for full reproducibility.**

Artifact Path	Purpose	Count
data/live_cluster/results_1k.json	Live-cluster replay outcomes (dry-run + apply)	1,000
data/live_cluster/summary_1k.csv	Live-cluster aggregate statistics	1
data/batch_runs/grok_5k/metrics_grok5k.json	Grok-5k acceptance & token telemetry	5,000
data/batch_runs/grok_full/metrics_grok_full.json	Manifest slice (1,313) acceptance	1,313
data/batch_runs/grok200_latency_summary.csv	Proposer latency summary (Grok-200)	280
data/batch_runs/verified_grok200_latency_summary.csv	Verifier latency summary (Grok-200)	140
data/eval/significance_tests.json	Statistical significance tests (z-test, Mann-Whitney U)	12
data/eval/table4_counts.csv	Table 4 manifest counts per corpus	4
data/eval/table4_with_ci.csv	Wilson 95% confidence intervals	4
data/scheduler/fairness_metrics.json	Scheduler fairness (Gini, starvation)	830
data/scheduler/metrics_schedule_sweep.json	Scheduler parameter sweep results	16
data/risk/risk_calibration.csv	Risk reduction (ΔR) per corpus	2

[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] ArtifactHub Documentation. Accessed: Oct. 2025. [Online]. Available: https://artifacthub.io/docs/

[22] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[23] M. Joseph, M. Kearns, J. H. Morgenstern, and A. Roth, "Fairness in Learning: Classic and Contextual Bandits," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

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 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.