

USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

PatchWeaver: Risk-Bounded Autonomous Vulnerability Remediation Under Change-Management Policies

Rui Li and Shuang Cao, *Dawn Computing*

https://www.usenix.org/conference/usenixsecurity26/presentation/li-rui

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

PATCHWEAVER: Risk-Bounded Autonomous Vulnerability Remediation Under Change-Management Policies

Rui Li
Dawn Computing
mark.li@dawncomputing.com

Shuang Cao
Dawn Computing
scao@dawncomputing.com

Abstract

Security teams increasingly rely on automation to keep pace with vulnerability disclosure and patch deployment, yet real remediation workflows are constrained by change-management policies: maintenance windows, staged rollout rules, blast-radius limits, approval gates, and least-privilege requirements. Today, "autonomous" remediation is brittle. Scripted playbooks are safe but rigid, while LLM-driven agents are flexible but frequently violate operational policies (e.g., patching outside windows, taking too many replicas offline, or skipping mandatory validation), creating outages and compliance risk.

We present PATCHWEAVER, a remediation system that provides *policy-bounded autonomy*. PATCHWEAVER combines (i) an explicit, continuously refreshed graph of Kubernetes assets, dependencies, identities, vulnerabilities, approvals, and evidence; (ii) a typed policy interface (CHANGESPEC) that encodes organization-specific change rules as executable predicates; and (iii) a rollout-based decision layer that scores candidate remediation plans by predicting both progress and policy-violation risk before any action is executed. The core insight is that myopic enforcement cannot avoid *constraint traps*—states where locally admissible actions lead to dead ends that force later violations. PATCHWEAVER detects traps via short-horizon simulation and replans when reality diverges from prediction.

We evaluate PATCHWEAVER on four Kubernetes remediation workloads under explicit governance constraints. Compared to a strong toolchain baseline (Gatekeeper + Argo Rollouts + Cosign), PATCHWEAVER reduces step-level policy violations by 52.5% (4.73% vs. 9.95%, $p < 0.01$) and episode-level violations by 52.2% (5.82% vs. 12.18%). Compared to an LLM-agent baseline, PATCHWEAVER reduces step violations by 79.1% and worst-case (p99) episode violations by 3.1× (16.8% vs. 51.3%). In security stress tests across 10 attack categories, PATCHWEAVER achieves 0.33% aggregate miss rate with mean dwell time 0.84 s, separating 88.7% immediate blocks from 11.0% escalations, versus 28.5% miss rate and 96.4 s for the toolchain. Control-plane overhead remains practical: p99 admission latency is 31 ms at 32 policy predicates, and total resource usage is under 1 CPU core and 1 GB memory.

1 Introduction

Cloud remediation is now a continuous, adversarial race. Exploitation often happens within days (or hours) of disclosure, while organizations must coordinate patches across container images, services, clusters, and cloud identities [7,22]. Modern patch-management guidance treats this as a governed workflow: teams triage affected assets, select scope, create change tickets, obtain approvals, validate in staging, perform progressive rollout, and verify the result [4,26,30]. Kubernetes makes this workflow concrete because many remediation steps are API-object changes: update an image digest, tighten an RBAC binding, add a NetworkPolicy, or advance an Argo canary. These changes are high-impact and policy constrained. Production remediation may be limited to maintenance windows, staged rollout rules, blast-radius caps, approval gates, artifact-provenance checks, and least-privilege boundaries. Violating those rules can cause outages, audit failures, or new security regressions.

This paper studies how to automate this governed workflow without removing the governance that makes production remediation safe. Given a remediation objective, a typed view of current cluster and governance state, and executable change-management policies, the system should produce a sequence of typed actions such as rebuilding an image, signing an artifact, updating a Deployment, advancing a canary, tightening RBAC, or requesting approval. The desired guarantee is not that the system can fix every CVE autonomously. Rather, the goal is *policy-bounded autonomy*: execute only those actions whose current and predicted trajectory remain compliant with declared policies, and otherwise deny, wait for evidence, or escalate with an auditable explanation.

Why today's "autonomous remediation" fails. Scripted runbooks are safe but rigid: they cannot cover the long tail of dependency graphs, heterogeneous environments, and exception handling. LLM-based agents are flexible, but their failure

modes are poorly matched to remediation. The dominant failure is not that an LLM cannot produce a command; it is that the agent loses track of evolving system state and governance constraints. Examples include patching production outside an approved window, taking too many replicas offline simultaneously, skipping a mandatory staging validation, or executing an action with privileges that violate separation-of-duties policies. Such errors compound across multi-step workflows and are amplified by partial observability (delayed signals, transient failures, inconsistent telemetry). In Kubernetes, these failures manifest concretely as: proposing privileged Pods to "debug" a rollout, applying overly broad ClusterRoleBindings to "fix" access, or forcing a deployment update that violates availability budgets and causes cascading outages. These risks are exacerbated by the complexity of Kubernetes networking and its security implications, which are not always visible at the manifest level [24].

Why existing tools are insufficient. One might expect that combining best-in-class policy engines (Gatekeeper/OPA, Kyverno), authorization languages (Cedar), supply-chain verification (Cosign, Sigstore), progressive delivery controllers (Argo Rollouts), and workflow graphs (LangGraph/LangChain) would solve the problem [2, 3, 19, 21, 28, 35]. These systems are useful and we build around them. However, they address different layers. Policy-as-code engines evaluate individual requests or authorization decisions; rollout controllers gate delivery stages; workflow frameworks encode control flow and human-in-the-loop steps. None of these components, by itself, scores a candidate remediation *trajectory* against evolving approvals, maintenance windows, health signals, artifact evidence, and privilege boundaries. As a result, a locally admissible action can lead to a *constraint trap*: a state where all future actions either violate policy or cause availability loss.

Constraint trap example. Consider a 4-step remediation workflow: (1) rebuild image with patched base, (2) sign and push to registry, (3) initiate canary rollout at 10%, (4) promote to production. Each step is individually admissible. However, if the agent executes step (3) before obtaining the required production-approval ticket—which is only issued after step (2) completes and a staging gate passes—then step (4) will be blocked. The agent is now trapped: it cannot proceed (missing approval) and cannot rollback (the approval policy forbids reverting a canary without incident justification). Myopic admission checks pass at every step, yet the trajectory leads to a dead end. In our evaluation, LLM-Agent walks into such traps in 34 of 50 Canary Gating episodes, triggering costly rollbacks that increase TTR by 47%.

This motivates plan-level reasoning that can anticipate trajectory-dependent failures before actions are executed. Meanwhile, recent work has shown that excessive Kubernetes permissions remain detectable and exploitable at scale [12], reinforcing the need for least-privilege enforcement in remediation controllers.

Observation: autonomy must be bounded by governance. Remediation is fundamentally a constrained, multi-step decision problem. When constraints are explicit and auditable, the right question is not "can an agent patch?", but "can an agent *predictably* patch without violating policy under uncertainty?" This motivates a design principle we call *policy-bounded autonomy*: the system may act autonomously, but only when its planned actions are forecasted to remain compliant with executable change-management policies under plausible system evolution.

Our approach. We present PATCHWEAVER, a Kubernetes-focused remediation control plane that couples three elements: (i) a continuously refreshed, typed state graph over Kubernetes objects (Deployments, Pods, ServiceAccounts, Roles/Bindings, NetworkPolicies), artifacts (images, SBOM/provenance metadata), vulnerabilities, and change tickets; (ii) an executable policy layer written in CHANGESPEC that captures admission and change-management constraints as first-class predicates with explanation; and (iii) a rollout-based decision layer that scores candidate multi-step plans by forecasting both progress and policy-violation risk before executing the next action. The core mechanism is simple: proposals are cheap, execution is expensive, so we simulate first and act second. Crucially, PATCHWEAVER is not a monolithic "LLM agent"; it is a policy-centered system that treats the LLM as an *untrusted proposer* and uses admission-like checks, explicit state, and rollouts to bound behavior.

Key insight: trajectory-level compliance. Existing policy-as-code tools answer a point-in-time question: "Is this action admissible now?" PATCHWEAVER answers a trajectory-level question: "Will this plan remain compliant through completion, given what we know about state evolution?" This shift requires (1) formalizing constraint traps as reachability failures, (2) simulating candidate plans under executable policies, (3) tracking policy and evidence versions, and (4) detecting drift between predicted and observed state to trigger replanning. The contribution is therefore not a new general-purpose workflow framework, nor a claim that OPA, Cedar, or LangGraph are unsuitable. It is the integration of executable governance with short-horizon trajectory scoring and fail-closed execution for vulnerability remediation.

Contributions.

1. We formalize autonomous remediation as a constrained multi-step decision problem and introduce the notion of *constraint traps*—states from which all admissible plans eventually violate policy—that myopic enforcement cannot avoid.
2. We design CHANGESPEC, a typed policy interface and evaluation engine that captures common operational and security constraints (windows, staged gates, blast radius, approvals, privilege bounds) and returns actionable violation explanations with audit-grade provenance.
3. We build PATCHWEAVER, a policy-bounded remediation

system that performs short-horizon rollouts to score candidate plans by predicted progress and compliance risk, with drift detection that triggers replanning when reality diverges from prediction.

4. We evaluate PATCHWEAVER on four Kubernetes workloads (image patching, RBAC minimization, progressive rollout, policy drift recovery) across 200 episodes and seven baselines. PATCHWEAVER reduces step-level violations by 52.5% vs. the strongest static baseline (Gatekeeper+Argo+Cosign) and by 79.1% vs. LLM-Agent. In security stress tests across 10 attack categories, PATCHWEAVER achieves 0.33% aggregate miss rate with mean dwell time 0.84 s versus 28.5% miss rate and 96.4 s dwell time for the toolchain baseline.

2 Background and Motivation

2.1 Kubernetes remediation primer

Kubernetes exposes cluster state through declarative API objects. A DEPLOYMENT manages replicated Pods; a SERVICEACCOUNT determines the identity under which Pods run; ROLE, CLUSTERROLE, and binding objects grant permissions; NETWORKPOLICY constrains traffic; admission controllers validate or mutate API requests before they are persisted; and rollout controllers move a service through canary or blue-green stages. A remediation action therefore changes a typed object or external evidence source: an image digest, a signature, an SBOM, a ticket status, an RBAC binding, a rollout weight, or a health gate.

PATCHWEAVER normalizes these objects and evidence sources into a state graph G_t, where t denotes a decision time. Nodes represent resources and evidence; edges represent relationships such as "Deployment uses image," "Pod runs as ServiceAccount," "ticket approves rollout," or "CI job validates artifact." This graph is not a workflow graph in the LangGraph sense: its nodes are operational entities and evidence, not agent computation steps. The workflow is a sequence of candidate actions evaluated over the graph. Introducing G_t early matters because CHANGESPEC rules in Section 4 range over graph nodes, edges, attributes, actions, and external evidence.

2.2 Remediation as a governed workflow

Enterprise patch management is typically implemented as a governed workflow comprising triage, scope selection, change ticket creation, approvals, staging validation, canary rollout, progressive rollout, and post-change verification. The governance surface is large: security policies restrict which changes are permitted; reliability policies bound blast radius; production changes may be restricted to explicit maintenance windows; staged rollout may be mandatory for certain asset classes; and least-privilege rules constrain which principals can touch which resources. These constraints are not merely organizational—they are often required for regulatory compliance (SOC 2, PCI-DSS), incident postmortems, and auditability. They shape what safe automation must do.

Kubernetes shifts the control surface. In Kubernetes and cloud-native stacks, a large fraction of remediation is expressed as changes to declarative objects: image references, Pod specs, RBAC bindings, network policies, and admission configurations. This creates an attractive enforcement point: admission control can accept, mutate, or reject changes before they persist in the API server. However, remediation rarely fits a single admission decision. Safe remediation requires *sequences* of changes (build $\rightarrow$ sign $\rightarrow$ canary $\rightarrow$ progressive rollout $\rightarrow$ verify), and governance constraints depend on evolving state (health signals, approval status, maintenance window, and artifact evidence). This motivates plan-level reasoning rather than only action-level checks.

2.3 Why constraints are hard for flexible agents

Two properties make remediation constraints challenging for flexible automation. First, constraints are *stateful*: approvals can arrive mid-episode; maintenance windows open and close; instances can become unhealthy. Second, constraints *interact*: a safe canary can still violate a maintenance window; a staged rollout can violate a blast-radius cap when dependencies are mis-modeled or health signals are delayed.

Myopic enforcement blocks actions that violate constraints *now*, but it does not reason about trajectories. It cannot detect constraint traps early. In a constraint trap, locally admissible actions lead to a dead end that forces a later violation: skipping staging can make a later production change infeasible; skipping approvals can make later gates impossible; overconsuming availability budget can force rollback. The key challenge is anticipating these multi-step interactions before execution.

Example: constraint trap in RBAC tightening. Consider a workflow to remediate an overprivileged ServiceAccount. A myopic agent may first remove a RoleBinding to reduce access, but this action inadvertently breaks a dependent service that requires that binding. The agent then attempts to restore the binding, but the restored binding violates a "no RBAC escalation" policy because the change ticket only authorized tightening, not loosening. The agent is now trapped: it cannot proceed (broken service) and cannot rollback (policy violation). PATCHWEAVER avoids this by simulating the multi-step trajectory before execution and detecting that the initial removal leads to a trap.

2.4 Design goals

We design PATCHWEAVER around five goals:

1. **Executable governance.** Policies must be executable predicates with structured explanations, not informal natural

language. This enables automated evaluation, explanation, and audit.
2. **Policy-bounded autonomy.** The system should act autonomously only when predicted to remain compliant across the planning horizon.
3. **Drift awareness.** The system must detect divergence between observed and predicted state and replan safely to avoid cascading errors from stale assumptions.
4. **Least privilege and audit.** Actions must execute with scoped credentials and produce an auditable trace linking decisions to policy predicates and evidence.
5. **Practicality.** The system must integrate with existing Kubernetes admission and rollout controllers, scanners, CI, and ticketing without invasive platform changes or custom API servers.

What the system does not guarantee. PATCHWEAVER does not prove that an organization's policies are complete or correct, and it does not guarantee remediation success when every compliant plan requires missing evidence or human approval. Instead, it guarantees that accepted actions pass the declared policy interface over current state and simulated near-future states; when that guarantee cannot be established, the system denies or escalates.

3 Threat Model

3.1 Adversary capabilities

We consider an adversary whose goal is to cause unsafe or non-compliant remediation actions, leading to outages, evasion, or expanded access. The adversary may:

- **Manipulate observations**: induce transient failures, delay signals (e.g., CI status), or create inconsistent telemetry to trigger drift and confuse the decision layer.
- **Achieve partial compromise**: control a subset of endpoints, ServiceAccounts, CI runners, a single namespace, or a GitOps branch.
- **Exploit partial remediation**: induce rollbacks to create inconsistent state.
- **Submit adversarial inputs**: inject malicious change inputs via tickets, metadata, or configuration changes.

Adversary entry points. We explicitly model four entry points: (i) GitOps pull requests submitted by external contributors or compromised accounts; (ii) direct kubectl access by operators with limited RBAC permissions attempting privilege escalation; (iii) compromised CI runners that may inject malicious artifacts or attestations; and (iv) container registries serving images with swapped tags or missing provenance. We do not consider attacks that directly compromise the Kubernetes API server or etcd, which are out of scope for the remediation layer.

3.2 Trust assumptions

We assume the PATCHWEAVER coordinator and policy engine are not fully compromised; if they are, guarantees are void. Action execution uses authenticated channels and scoped credentials; compromise is possible but should be detectable via audit logs. We assume CHANGESPEC policies represent intended change-management rules (policy *correctness* is out of scope; policy *enforcement* is in scope). We assume signature and provenance verification are sound: if evidence is present, it is cryptographically valid. PATCHWEAVER relies on external tooling for verification and does not re-implement cryptography. We assume the Kubernetes API server is an authenticated source of state, watch streams are authenticated, and connector credentials are provisioned correctly. We do not assume that every external evidence source is always fresh or honest; ticketing, CI, registry, and telemetry signals may be stale, delayed, missing, or partially compromised. The state graph therefore carries source, timestamp, version, and evidence hashes so policies can distinguish trusted, stale, conflicting, and unauthenticated observations.

3.3 Security goals

PATCHWEAVER aims to provide:

- **Policy compliance**: prevent executions that violate explicit policies, even under noisy observations and adversarial inputs.
- **Least-privilege execution**: actions run with scoped credentials; actions cannot silently escalate privilege through remediation. If elevated privilege is required, PATCHWEAVER rejects or escalates for human approval.
- **Auditability**: produce an audit trail linking decisions to policy predicates and evidence (state snapshots, signatures, approvals).
- **Robustness to drift**: detect divergence between predicted and observed state, trigger replanning, and avoid cascading errors from stale plans.
- **Attack resistance**: block or escalate adversarial inputs that exploit multi-step trajectories.

We evaluate attack resistance against ten attack categories in Section 7.3, mapped to adversary capabilities as follows. *Privileged Container*, *HostPath Mount*, and *RBAC Wildcard* attacks correspond to adversarial change inputs via GitOps PRs or kubectl access. *Aggregated Role Bypass* and *Multi-Step RBAC Escalation* exploit partial compromise by assembling individually valid bindings into an aggregate privileged role. *Image Tag Swap* and *Staged Provenance Bypass* exploit compromised registries or CI runners. *Policy Drift Mid-Rollout* and *Delayed Health Signal* exploit observation manipulation via signal delays or inconsistent telemetry. *Webhook Degradation* exploits workflow attacks by degrading policy engine observability during remediation. For attacks where PATCHWEAVER cannot definitively distinguish malicious from legitimate inputs (e.g., Webhook Degradation

under network latency), the correct behavior is to escalate for human review rather than proceeding or blocking incorrectly.

3.4 Partial state-graph compromise

The state graph is a derived view, not a root of trust. Attackers may try to corrupt that view by suppressing health signals, replaying stale approval status, presenting inconsistent CI and registry evidence, or racing policy updates against rollout decisions. PATCHWEAVER mitigates these cases in four ways. First, graph facts are source-labeled and versioned; high-impact predicates can require corroboration across independent sources (e.g., registry digest plus signature transparency log plus CI attestation). Second, stale evidence expires: approvals, health gates, and artifact checks include timestamps and maximum age. Third, conflicting facts increase drift and trigger replanning or escalation rather than silent execution. Fourth, the audit log stores state hashes and evidence hashes, enabling post-incident reconstruction when an external source later proves compromised. These mechanisms do not protect against full API-server, etcd, key-management, or PATCHWEAVER coordinator compromise. They are intended to make partial evidence compromise fail closed and visible.

3.5 Out of scope

We exclude: full compromise of the Kubernetes control plane (etcd, API server); attacks that bypass authentication entirely; kernel exploits on worker nodes; hypervisor compromise; supply-chain attacks on the verifier itself; key management compromise; timing, resource, or covert channels; physical attacks; and insider attacks with full admin credentials. We do not treat availability as a primary security goal (an adversary may overload the decision layer; we treat that as an operational issue). We do not verify that policies encode organizational intent; we enforce policies as written.

4 Policy Model: CHANGESPEC

This section defines CHANGESPEC, the policy interface used by PATCHWEAVER to bound autonomous remediation. CHANGESPEC is intentionally narrow. It is not a replacement for OPA/Rego or Cedar, both of which can express many individual authorization and admission checks and can attach useful denial messages [2, 14, 28]. The design goal is different: represent remediation-relevant predicates in a typed action/state interface so the same checks can be evaluated on both current state G_t and predicted states $\hat{G}_{t+h}$ during rollout scoring. Organizations with existing Rego or Cedar policies can keep those policies as predicate implementations. PATCHWEAVER supplies the surrounding adapter: it maps the candidate action and relevant state-graph slice into the policy engine input, records the policy version and evidence hash, and wraps the result as a typed predicate that can be evaluated during both admission and simulated rollouts.

4.1 State, actions, and rule grammar

At decision time t, policies are evaluated over a typed state graph $G_t = (V_t, E_t, X_t)$ and a candidate action a_t. Graph nodes V_t include Kubernetes resources (DEPLOYMENT, POD, SERVICEACCOUNT, ROLE/BINDING, NAMESPACE), artifacts and evidence (IMAGE, SIGNATURE, SBOM, CIRUN), vulnerabilities, tickets, and approvals. Edges E_t encode relationships such as RUNSAS, BOUNDTO, USESIMAGE, APPROVES, VALIDATEDBY, OWNS, and DEPENDSON. Attributes X_t store timestamps, source identifiers, policy versions, vulnerability status, health/readiness, rollout stage, evidence freshness, and evidence hashes. Candidate actions are typed records: ⟨type, target, parameters, preconditions, executor identity⟩.
Grammar. The core syntax is compact:

```
rule       ::= "rule" id "{" match when decision priority?
↪  explain "}"
match      ::= "match:" scope
when       ::= "when:" expr
decision   ::= "decision:" ("ALLOW" | "DENY" | "REQUIRE"
↪  evidence
                          | "ESCALATE" reason)
priority   ::= "priority:" integer
expr       ::= predicate | "NOT" expr | expr ("AND" |
↪  "OR") expr
predicate  ::= node_pred | edge_pred | attr_pred |
↪  action_pred
             | evidence_pred | aggregate_pred
```

The `match` clause is a coarse applicability filter: it selects the action and object classes to which the rule applies. The `when` clause is the actual guard: it evaluates a typed Boolean expression over the bound action, graph neighborhood, attributes, and evidence. The `decision` specifies the policy outcome, and `explain` provides the audit message. Predicates are typed by the data they inspect: node predicates range over graph nodes, edge predicates over graph relationships, attribute predicates over node/edge fields, action predicates over a_t, evidence predicates over external attestations, and aggregate predicates over bounded graph queries.

4.2 Representative rules

We illustrate three rule types from the evaluation policy set.
Deny RBAC wildcards.

```
rule deny_rbac_wildcard {
  match: action.type in {ApplyRole, UpdateClusterRole}
  when: action.object.rules[*].verbs contains "*"
     OR action.object.rules[*].resources contains "*"
  decision: DENY
  priority: 90
  explain: "RBAC wildcards violate least privilege"
}
```

This action predicate denies role changes whose proposed object contains wildcard verbs or resources.
Require staging evidence before production.

```
rule require_staging_before_prod {
  match: action.type == AdvanceRollout
  when: action.target.env == "prod"
    AND NOT edge_exists(ValidatedBy,
        src=action.target.deployment,
        dst=Evidence[type=StagingPass, fresh<30m])
  decision: REQUIRE Evidence[type=StagingPass]
  priority: 80
  explain: "Production rollout requires fresh staging pass"
}
```

Evidence comes from connectors that write signed or versioned facts into G_t, such as ticket-system approvals, CI/staging results, registry signatures, SBOM metadata, and Kubernetes health events. `REQUIRE` means the action is not executable yet, but it may become executable if the requested evidence appears and remains fresh. Execution without that evidence would be a violation; waiting for it or requesting it is not.

Escalate ambiguous webhook degradation.

```
rule webhook_degraded {
  match: action.type in {ApplyManifest, AdvanceRollout}
  when: attr(policy_engine.health) == "degraded"
    OR aggregate(conflicting_sources(action.target)) > 0
  decision: ESCALATE "policy evidence is inconsistent"
  priority: 95
  explain: "Cannot establish policy state confidently"
}
```

This captures cases where fail-closed human review is safer than either automatic allow or automatic deny. `ESCALATE` is used when the controller cannot establish the policy state confidently, for example because evidence sources disagree or a policy conflict is unresolved.

4.3 Semantics and rule evaluation

Let R be the ordered policy set. For each rule r_i, evaluation returns one of $\{\textsc{Allow}, \textsc{Deny}, \textsc{Require}(e), \textsc{Escalate}(q), \bot\}$, where $\bot$ means the rule did not match. A candidate action is executable only if no active rule returns DENY, no active rule returns ESCALATE, and all required evidence is present and fresh. For safety, terminal outcomes are ordered as DENY $>$ ESCALATE $>$ REQUIRE $>$ ALLOW; priority orders explanations and resolves same-class conflicts, but a high-priority escalation cannot override a hard deny. We define two indicators used by the scoring function in Section 5.4. The hard-violation indicator is:

$$v(G_t, a_t) = \mathbb{I}[\textsc{Decision}(G_t, a_t) = \textsc{Deny}].$$

$v = 1$ means the action would violate a hard policy if executed. We separately define $e(G_t, a_t) = \mathbb{I}[\textsc{Decision}(G_t, a_t) \in \{\textsc{Require}, \textsc{Escalate}\}]$, which penalizes missing evidence or human review without counting it as an executed violation. For executed logs, a step-level violation is counted only when an action is executed despite an active deny, unsatisfied requirement, expired evidence, or mandatory escalation. Thus, requesting approval is not itself a violation; bypassing or timing out an approval requirement is.

Algorithm 1 CHANGESPEC Rule Evaluation

Require: State graph G_t, candidate action a_t, ordered rules R

denies $\leftarrow \emptyset$, requires $\leftarrow \emptyset$, escalates $\leftarrow \emptyset$
reasons $\leftarrow \emptyset$
for each rule $r_i \in R$ in descending priority **do**
  if $\textsc{Match}(r_i.scope, a_t, G_t)$ and $\textsc{Eval}(r_i.expr, a_t, G_t)$ **then**
    append $(r_i.id, r_i.explain, \textsc{Bindings}(r_i))$ to reasons
    if $r_i.decision = \textsc{Deny}$ **then**
      add r_i to denies
    else if $r_i.decision = \textsc{Require}(e)$ **then**
      add e to requires
    else if $r_i.decision = \textsc{Escalate}(q)$ **then**
      add q to escalates
    end if
  end if
end for
if denies $\neq \emptyset$ **then**
  return DENY with highest-priority deny and reasons
else if escalates $\neq \emptyset$ **then**
  return ESCALATE with highest-priority reason and reasons
end if
if all evidence in requires is present and fresh in G_t **then**
  return ALLOW with reasons
else
  return REQUIRE(missing evidence) with reasons
end if

Algorithm 1 is used twice: once on the current graph before execution, and repeatedly on predicted graphs during plan simulation. This reuse is what lets PATCHWEAVER detect trap-inducing trajectories before any mutation reaches the cluster.

4.4 Predicate taxonomy and generalizability

Our evaluation uses 32 representative predicates across seven categories (Table 1). The set is not claimed to be complete for all organizations. It was derived by translating a production Kubernetes change-management document into executable checks, then iterating over denial explanations observed during validation. Table 1 is meant as a map of what the evaluation policy set covers: each row groups predicates that inspect the same kind of state, and the range column shows whether the predicates read action fields, graph structure, attributes, evidence, or bounded aggregates.

Using existing policy engines. The categories are meant to be extensible. A new organization can add predicates for its own ticket schema, deployment topology, verifier, or emergency-change policy without changing the planner. If an organization already uses OPA, the practical migration path is a Rego adapter: PATCHWEAVER serializes the bound action, graph neighborhood, evidence timestamps, and policy version into an OPA input document; OPA returns allow/deny plus viola-

Table 1: Policy taxonomy: 32 representative predicates across seven categories. Ranges identify what each predicate inspects.

Category	#	Range	Example Predicate
Time windows	3	attrs/evid.	`window(ns)`
Ordering/gates	6	edges/evid.	`staging_ok(dep)`
Blast radius	5	aggregate	`offline(dep)<k`
Least privilege	6	action/node	`no_wildcard(role)`
Approvals	4	evidence	`approved(id)`
Admission safety	4	action/node	`no_hostpath(pod)`
Artifact trust	4	attrs/evid.	`has_sig(img)`

tion messages; CHANGESPEC records that result as a typed predicate outcome that can be replayed on predicted states. This adapter covered 24 of the 32 predicates in our prototype because they were local checks over action fields, Kubernetes objects, or artifact attributes. The remaining eight needed hand-written bindings because their truth depended on temporal governance state, such as approval freshness, rollout-gate ordering, or drift between predicted and observed graphs. For Cedar-style authorization policies, the analogous integration is to call Cedar for principal/action/resource decisions and embed the result, policy id, and entity bindings as evidence used by CHANGESPEC.

4.5 Policy engineering, conflicts, and auditing

Authoring the 32-predicate set took 4.5 person-hours: 2.5 hours for initial translation, 1.5 hours for validation against denial explanations, and 0.5 hours for conflict resolution. We observed 12 predicate revisions during validation, mostly around ordering/gate predicates, and resolved two conflicts by explicit priority. This cost is comparable to authoring a small policy-as-code bundle; the distinctive benefit is that every decision, including predicted denials during simulation, is logged with the same typed explanation schema.

Policy sets can be incomplete or inconsistent. CHANGESPEC therefore supports explicit rule priorities, static checks for duplicate scopes with contradictory decisions, and runtime fail-closed behavior. If no priority resolves a conflict, PATCHWEAVER returns ESCALATE with the conflicting rule ids and bound entities. The audit log records rule id, policy version, graph hash, bound entities, failed subpredicate, evidence hashes, and human-readable explanation for each allow, deny, requirement, and escalation. In evaluation, audit completeness is 100% because every executed or rejected action flows through this interface; this is a coverage statement about the prototype logs, not a claim that the policy set captures every possible organizational rule.

SBOM evidence is an active research and engineering area; we treat SBOM presence as one input to policy evaluation rather than a complete security guarantee [18, 40].

5 System Design

5.1 Architecture

Figure 1 shows the PATCHWEAVER architecture. An operator (or upstream automation) submits a remediation objective, such as "remediate CVE-2024-21626 for service `api-gateway` in prod." PATCHWEAVER builds a typed Kubernetes state graph from the Kubernetes API (objects and events), scanner results, ticketing state, CI signals, and container registry metadata (signatures/SBOM). A planner generates candidate multi-step plans, which are scored by rollouts under CHANGESPEC; Section 5.4 defines this scoring function. The next action of the best-scoring plan is executed by a connector with scoped credentials and logged provenance.

Figure 1 separates proposal, enforcement, and execution trust boundaries. Inputs are heterogeneous and noisy, so we normalize them into a typed state graph. Policies are evaluated on state and predicted trajectory, with structured explanations. Actions execute only through scoped connectors and admission checks. Every allow/deny/escalation is logged with policy version and evidence hash.

The architecture enforces six key properties: (1) Governance data (approvals, maintenance windows, CI attestations) are first-class inputs to the state graph, preventing the controller from treating governance as unstructured text. (2) The proposer is isolated from execution: it may generate arbitrary plans, but every action is mediated by CHANGESPEC and rollout scoring before any mutation occurs. (3) The trust boundary is explicit: the proposer is treated as untrusted input, while the policy engine and executor form the enforcement boundary. (4) Least privilege is enforced at the connector level, ensuring that even a correct plan cannot execute beyond scoped credentials. (5) The decision log is on the critical path, recording each allow/deny with policy version and evidence for auditability and post-incident analysis. (6) The propose–simulate–act loop anticipates trajectory-dependent failures (constraint traps) that are invisible to point-in-time admission checks.

5.2 State-graph refresh and prediction

Section 4 defines the graph schema used by CHANGESPEC. The design question is how PATCHWEAVER keeps that graph current enough for admission and predictive enough for rollout scoring. The graph is refreshed continuously from Kubernetes watch streams and external connectors using a delta-merge strategy: only changed nodes and edges are updated, preserving graph identity for cache validity. Evidence connectors write approvals, CI attestations, signatures, SBOM presence, and window state with source identifiers and freshness timestamps. During simulation, PATCHWEAVER applies action-specific transition functions to produce predicted graphs $\hat{G}_{t+h}$, while carrying evidence versions forward conservatively. These predicted graphs are checked by the same

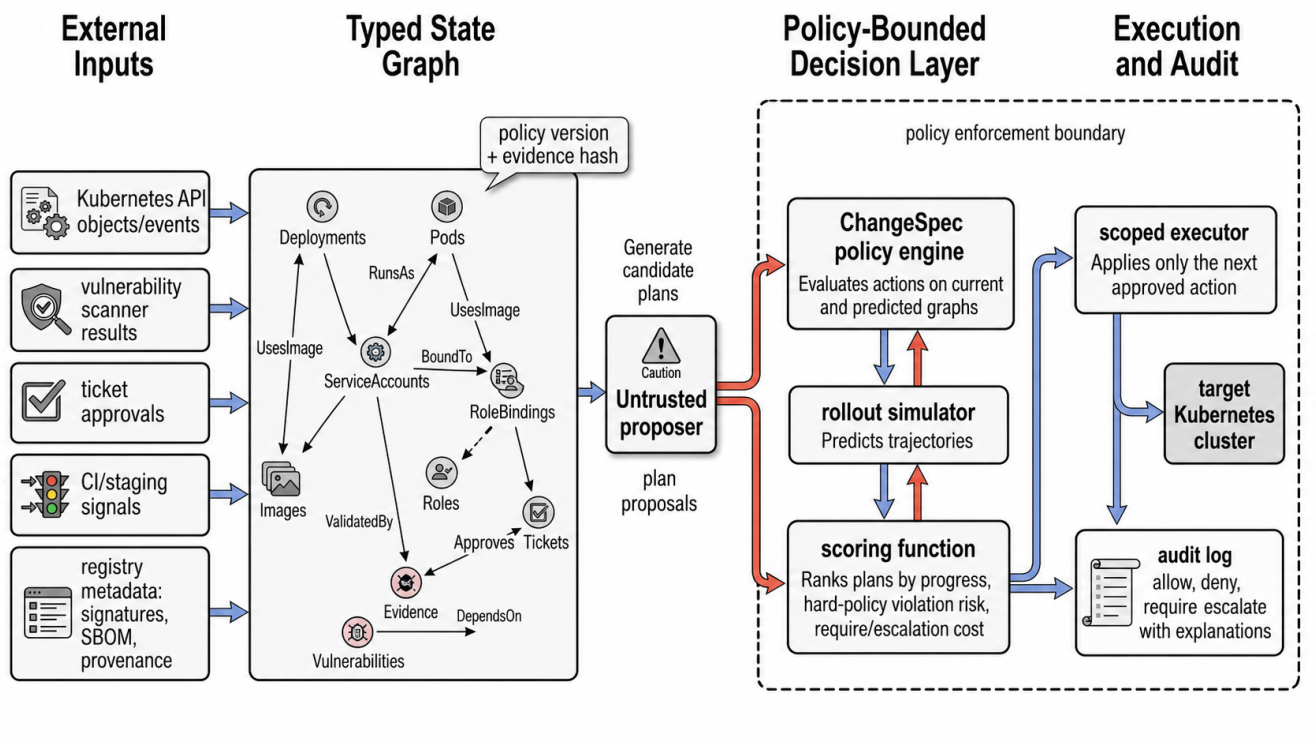

Figure 1: PATCHWEAVER pipeline: propose–simulate–act with executable policy constraints. Inputs flow from external systems (left) through the state graph and policy engine (center) to the executor and target systems (right). All decisions are logged to an audit trail.

CHANGESPEC rules as the current graph, which keeps enforcement, scoring, and audit logs aligned.

5.3 Constraint trap formalization

A central insight is that myopic admission cannot prevent *constraint traps*. We formalize this as follows.

Definition 1 (Constraint Trap)**.** *Given state G_t, policy set $\mathcal{C}$, and planning horizon H, a state G_t is a* constraint trap *if:*

$$\forall \pi \in \Pi(G_t, H): \quad \exists h \in [1, H]: \mathcal{C}(\hat{G}_{t+h}, a_{t+h}) = 0$$

where $\Pi(G_t, H)$ is the set of all admissible plans of length H, and $\hat{G}_{t+h}$ is the predicted state after executing the first h actions. Informally, a trap is a state from which all feasible multi-step plans eventually violate policy.

Detection. PATCHWEAVER approximates this reachability condition online by simulating the top-K candidate plans emitted by the proposer and checking whether all plans have nonzero predicted violation. The system does not enumerate the full action space; instead, it combines typed action schemas, template plans, and LLM proposals to cover the feasible remediation frontier. If no candidate is currently executable, PATCHWEAVER identifies the highest-scoring blocked candidate under the scoring function in Section 5.4. When that candidate is blocked only by missing evidence, PATCHWEAVER returns REQUIRE for that evidence; when every candidate contains a hard denial or ambiguous policy state, it escalates with a trap notification. This turns a latent multi-step failure mode into an explicit, auditable decision point. The key property is *timing*: the trap is detected before execution, and the user sees the violated predicate, required evidence, and required ordering early—supporting safe intervention, policy refinement, and compliance reviews.

Figure 2 illustrates trap avoidance in the Canary Gating workload. LLM-Agent walks into a trap by attempting production rollout before obtaining approval (which requires staging validation to complete first). PATCHWEAVER simulates the plan at horizon H=6, predicts the dead end at step 3, and replans to request approval before proceeding.

Figure 2 shows why point-in-time checks are insufficient. In (a), each step is locally admissible, but the sequence becomes infeasible. The dead end is caused by gate ordering and approval availability, not syntax. In (b), rollouts reject unsafe trajectories before any mutation executes. This avoids rollbacks and turns traps into explicit, auditable escalations.

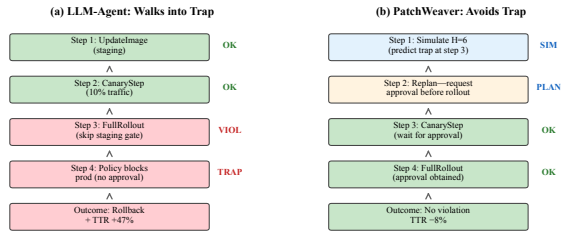

Figure 2: Constraint trap case study. (a) LLM-Agent walks into a trap: steps 1–2 are admissible, but step 3 (full rollout without approval) is blocked, forcing rollback. (b) PATCHWEAVER simulates the trajectory, detects the trap at h=3, and replans to request approval earlier.

5.4 Decision layer: proposing, scoring, and replanning

PATCHWEAVER decouples *proposal* from *scoring*. Any proposer may be used: a deterministic planner, a template library, or an LLM. In our prototype we use an LLM-based proposer (GPT-4-turbo) with a prompt describing available actions and a typed state summary. The proposer outputs a JSON-serializable intermediate representation (action type plus typed arguments) validated before scoring. Action types include: rebuild and sign images, update deployments, advance canary steps, tighten RBAC, apply network policies, rollback, and run verification steps. Crucially, proposals are not executed directly; they are filtered and scored by CHANGESPEC and rollouts. This separation ensures that even if the LLM proposes an unsafe action, the policy layer rejects it before execution.

At decision time, the proposer emits K candidate plans for horizon H. For each plan, PATCHWEAVER predicts trajectory $\hat{G}_{t+1:t+H}$ under conservative assumptions and computes:

$$S(\pi) = \sum_{h=1}^{H} r(\hat{G}_{t+h}, a_{t+h}) - \lambda_v \sum_{h=1}^{H} v(\hat{G}_{t+h}, a_{t+h}) - \lambda_e \sum_{h=1}^{H} e(\hat{G}_{t+h}, a_{t+h}).$$

where r is a bounded progress reward (e.g., vulnerable replicas removed, gates completed, privileges reduced). The v term is the hard-deny indicator from Section 4: it is 1 only when a predicted action would violate a hard policy if executed. The e term is 1 for REQUIRE or ESCALATE, so missing evidence and human review are penalized separately from hard violations. We set λ_v larger than any single-step progress reward, so a plan with a predicted hard violation cannot outrank a compliant plan with slower progress. λ_e is smaller: escalation is allowed but penalized, reflecting the operational cost of human review. The default parameters are H=6, λ_v=1.0, and λ_e=0.25, chosen via sensitivity analysis (Section 7.4): increasing H beyond 6 yields diminishing returns, and the selected penalties balance violation reduction against escalation overhead. We use K=12 candidate plans per decision step and cache policy evaluations per state-graph hash to keep scoring latency practical.

The scoring rule also answers why adding simple preconditions to a workflow graph is insufficient. Preconditions can block an action whose required fact is absent now, but PATCHWEAVER scores whether the proposed ordering will make the required evidence available before a later gate, whether a canary consumes too much availability budget before rollback remains legal, and whether a policy update invalidates cached assumptions mid-rollout. This is trajectory scoring over evolving evidence, not only graph control flow.

We replan when drift exceeds threshold τ=0.15:

$$\text{Drift}(G_t, \hat{G}_t) = 1 - \frac{|V_t \cap \hat{V}_t| + |E_t \cap \hat{E}_t|}{|V_t \cup \hat{V}_t| + |E_t \cup \hat{E}_t|}.$$

PATCHWEAVER versions policies with a monotonic sequence number; when a policy update is detected, PATCHWEAVER invalidates cached scores and triggers replanning. In-flight actions continue under the authorizing policy version. Executors run actions via scoped connectors; each action is checked for policy compliance, precondition satisfaction, and privilege scope. All actions and rejections are logged with explanations. If the policy engine is unavailable, PATCHWEAVER defaults to DENY with escalation.

6 Implementation

6.1 Control plane and connectors

PATCHWEAVER is implemented as a Kubernetes-native control plane deployed as a standard Deployment in a dedicated `patchweaver-system` namespace. The implementation totals approximately 8,400 lines of Go (controller logic, state graph, connectors) and 1,200 lines of Rego-like DSL (policy definitions and evaluation), excluding tests and generated code. We use Go 1.22.5, client-go v0.28.4 for Kubernetes API interactions, Argo Rollouts v1.6.4 for progressive delivery integration, and Cosign v2.2.2 for signature verification.

The control plane comprises four logical components: a graph builder that maintains the typed state graph via Kubernetes watch streams; a policy engine that evaluates CHANGESPEC predicates and produces structured explanations; a proposer interface that accepts candidate plans from an LLM or template library; and an executor coordinator that dispatches actions to scoped connectors and logs outcomes. The system maintains an append-only decision log backed by a PostgreSQL store, recording state snapshot hashes, proposed plans with scores, policy evaluations with explanations, chosen actions, and execution outcomes. The proposer interface is rate-limited (10 requests/s) to avoid saturating control-plane resources. All components share typed schemas for state, ac-

tions, policies, and logs, enabling deterministic replay for audit and debugging.

Connectors. We implement connectors for common remediation surfaces: scanner and inventory ingestion (Trivy, Grype), ticketing and approvals (Jira), CI validation (Tekton), Kubernetes API execution (kubectl/client-go), and container registry metadata (Docker Hub, ECR) for signature and SBOM provenance checks. Each connector runs with a dedicated ServiceAccount whose permissions are scoped to a policy-defined boundary and constrained by namespace where possible. This separation prevents the remediation pathway from becoming a privilege-escalation channel: no connector is authorized to expand its own authority as a side effect of remediation actions. The decision log records which identity executed each action, enabling reviewers to map each action to a bounded verb and resource set.

6.2 Admission, rollout gating, and policy versioning

PATCHWEAVER integrates at two Kubernetes control points: (i) a validating admission webhook that rejects unsafe changes (privileged pods, RBAC wildcards, missing provenance) with a policy explanation, and (ii) a rollout controller integration via Argo Rollouts AnalysisRuns that gate progressive delivery steps (canary/blue-green) on policy satisfaction and verification signals. This split is intentional: admission enforces instantaneous invariants, while rollouts enforce multi-step constraints (staging before prod, canary success before full rollout). Recent work has also studied how to mitigate best-practice violations directly in Kubernetes manifests [33]; PATCHWEAVER is complementary in that it enforces both manifest-level invariants and workflow-level governance constraints.

Admission failure semantics. When the admission webhook times out or returns an error, PATCHWEAVER is designed for fail-closed behavior: the API server denies the request rather than allowing policy-violating actions to proceed unchecked. If an operator configures fail-open admission for availability reasons, PATCHWEAVER records such actions as "unchecked" in the decision log for later review. The webhook is deployed as three stateless replicas behind a Kubernetes Service to ensure transient replica failures do not disable enforcement. We record webhook latency (p50/p95/p99) and timeout counts for monitoring.

Policy update semantics. Policies are stored as ConfigMaps in the `patchweaver-system` namespace with a monotonic version number. When a policy ConfigMap is updated, the policy engine reloads the policy set and invalidates cached plan scores so that subsequent decisions replan under the new policy version. In-flight actions that were already submitted continue under the policy version that authorized them, and outcomes are logged with that version identifier. The audit log records a policy-update event including both old and new policy hashes, enabling diff analysis.

6.3 Policy engine and audit log

CHANGESPEC policies compile to a predicate evaluation graph implemented as a directed acyclic graph (DAG) of predicate nodes. Each predicate node corresponds to a rule condition (e.g., "graph.count(Pod[status=NotReady]) > k") and can be evaluated incrementally when the state graph changes. The engine returns both a boolean and a structured explanation for violated rules; explanations include bound entities, the failed subpredicate, and a human-readable message. The engine is deterministic, side-effect free, cacheable, and versioned. Evaluation latency (p99 $<$ 5 ms for 32 predicates) and denial reasons are recorded per decision. At policy-load time, the engine also runs lightweight consistency checks: duplicate rule ids, unreachable match scopes, contradictory decisions at the same priority, missing evidence schemas, and cyclic `REQUIRE` dependencies. These checks do not prove global policy correctness, but they catch common operator errors before the policy version is admitted.

Decision loop. Algorithm 2 sketches one iteration of the control loop. Proposals are treated as untrusted inputs: the proposer can emit arbitrary plans, but the policy engine and rollouts mediate every action before execution. The simulation step evaluates policy predicates on predicted states (not just the current API object), and drift triggers replanning under partial observability. The audit trail records policy version, evidence, and explanations for both allows and denies, enabling replay and attribution.

Planner and intermediate representation. Plans are expressed in a typed intermediate representation (IR) rather than raw shell commands. Each action in the IR includes an action type, a typed target reference (e.g., Deployment and namespace), action-specific parameters (e.g., image digest, canary weight), and preconditions that must hold before execution. This allows CHANGESPEC to reason about action semantics and lets executors enforce privilege scopes. The IR is compiled into connector calls (e.g., K8s deployment patch, CI trigger) and can be replayed for audit.

Audit log schema. Every decision log entry includes: timestamp, serialized state graph hash, policy version identifier, proposed plans with scores, chosen action IR, policy decision and explanation, connector outcome, and execution identity (ServiceAccount and namespace). The log is append-only to ensure decisions remain auditable even when subsequent actions change cluster state. We also record rollout stage, gate outcomes, approval status, and CI/artifact evidence to support end-to-end attribution.

7 Evaluation

We evaluate PATCHWEAVER on four Kubernetes remediation workloads under explicit change-management constraints. Our experiments answer the following questions: (Q1) Does plan-level reasoning reduce policy violations compared to

Algorithm 2 PATCHWEAVER Decision Loop (one step)

Require: Current state graph G_t, policy set $\mathcal{C}$, proposer Π, rollout horizon H, candidates K, drift threshold τ

(plans) $\leftarrow \Pi(G_t; K, H)$ {LLM or template proposer}
for each plan $\pi \in$ plans **do**
  $(\hat{G}_{t+1:t+H}) \leftarrow \text{SIMULATE}(G_t, \pi)$
  $S(\pi) \leftarrow \sum_{h=1}^{H} r(\hat{G}_{t+h}, a_{t+h}) - \lambda_v v(\hat{G}_{t+h}, a_{t+h}) - \lambda_e e(\hat{G}_{t+h}, a_{t+h})$
end for
$\pi^\star \leftarrow \arg\max_\pi S(\pi)$
if $\forall \pi : S(\pi) < 0$ **then**
  ESCALATEWITHTRAPEXPLANATION($\pi^\star$)
  return
end if
$a_t \leftarrow \text{FIRSTACTION}(\pi^\star)$
$d \leftarrow \text{EVALUATERULES}(G_t, a_t, \mathcal{C})$
if $d = \text{DENY}$ **then**
  REJECTWITHEXPLANATION(a_t, d); **return**
else if $d = \text{REQUIRE}$ or $d = \text{ESCALATE}$ **then**
  ESCALATEWITHEXPLANATION(a_t, d); **return**
end if
EXECUTE(a_t) with scoped connector
Observe G_{t+1} from connectors
if $\text{DRIFT}(G_{t+1}, \hat{G}_{t+1}) > \tau$ **then**
  Invalidate cached plans; replan from G_{t+1}
end if
Log decision to audit trail

admission-only and LLM-agent baselines? (Q2) How does PATCHWEAVER perform under adversarial inputs that exploit multi-step trajectories? (Q3) What is the contribution of each component? (Q4) What is the control-plane overhead?

7.1 Setup and Workloads

We deploy PATCHWEAVER on a 12-node Kubernetes cluster (v1.28.4) on AWS EC2 (m5.2xlarge, 8 vCPU, 32 GB RAM, Intel Xeon Platinum 8259CL @ 2.50 GHz) running Ubuntu 22.04.4 (kernel 6.6.12), containerd 1.7.13, runc 1.1.12, and NVIDIA 550.54.14 / CUDA 12.4, with Istio 1.19.3, Argo Rollouts 1.6.4, Gatekeeper 3.14.0, Kyverno 1.11.1, and Cosign 2.2.2. We use GPT-4-turbo (gpt-4-1106-preview, temperature 0.0, max tokens 4096) as the LLM proposer. Each workload runs 50 episodes with randomized initial conditions; we report means, standard deviations, and 95% bootstrap confidence intervals (10,000 resamples). The CHANGESPEC policy set contains 32 predicates covering maintenance windows (3), staged rollout gates (6), approval requirements (4), blast-radius caps (5), least-privilege constraints (6), admission safety (4), and artifact trust (4). Episodes randomize namespace topology (8–14), RBAC bindings (42–78), rollout stage, and approval arrival times (30–180 s); all components are seeded for replay.

Metric definitions. We define two violation metrics computed from action-level decision logs:

- **Step violation rate** = (total violated actions across all episodes) / (total actions across all episodes). This is a pooled rate.
- **Episode violation rate** = average across episodes of (violated actions in episode / actions in episode). This captures per-episode behavior.

For distribution statistics (median, IQR, p90, p99), we compute per-episode violation percentages and report percentiles. We also report time-to-remediate (TTR), escalation rate (fraction of actions escalated to human review), and rollback rate (fraction of episodes requiring rollback). The primary violation rates are unweighted so that policy failures remain directly auditable from logs; for adversarial tests we report miss rate and dwell time separately so severe persistent unsafe states are not hidden by averaging.

The four workloads are: (1) **ImageRebuild**—patching CVE-2024-21626 in runc, rebuilding 8 containers, verifying Cosign signatures, and rolling out under staged gates (10%/30%/100%); (2) **RBAC Minimization**—reducing excessive permissions for 12 ServiceAccounts under separation-of-duties constraints [12]; (3) **Canary Gating**—progressive delivery with availability budgets (max 20% unavailable) and mandatory staging validation; and (4) **PolicyDrift Recovery**—mid-rollout policy changes (new approval requirement, shifted maintenance window) that force replanning.

We compare against seven baselines: (1) **Admission-only** (Gatekeeper 3.14); (2) **Strong-toolchain** (Gatekeeper + Argo Rollouts + Cosign + Kyverno); (3) **Runbook** (deterministic scripts with static ordering); (4) **LLM-Agent** (GPT-4-turbo with ReAct [41], no policy enforcement); (5) **PATCHWEAVER-H0** (one-step lookahead only, equivalent to PATCHWEAVER with $H = 0$); (6) **LLM+Policy** (LLM proposer with CHANGESPEC enforcement but *no* rollout simulation); and (7) **Template+Rollout** (deterministic template proposer with full rollout simulation). Baselines (6) and (7) isolate the contributions of the proposer and the scoring mechanism.

All methods share identical tool access (K8s API, Jira, Tekton connectors), action schemas (18 types), and per-episode budget (40 actions, where one action is one typed connector invocation). For LLM-Agent, we use a fixed system prompt that enumerates the action schema, specifies that actions must be issued as typed JSON, and instructs the agent to request approvals rather than bypassing gates. To avoid strawman baselines, we implement strict output validation and retries for malformed tool calls for both LLM-Agent and PATCHWEAVER's proposer, counting retries against the action budget.

External benchmarks. We considered ITBench, MicroRemed, and SRE-Bench [1, 17, 23]. They are valuable for evaluating operational task completion, incident diagnosis, and remediation-agent behavior, but they do not directly encode the governance constraints central to this paper: approval freshness, maintenance windows, staged production gates, artifact provenance, least-privilege execution, and mid-rollout policy drift. Using them unchanged would therefore test

whether an agent can complete an operational task, not whether it can remain compliant through a governed remediation trajectory. We view these benchmarks as complementary: adding policy/evidence annotations and rollout constraints to their tasks would enable broader apples-to-apples comparisons in future work.

7.2 Main Results

Table 2 summarizes performance across all four workloads. PATCHWEAVER achieves the lowest step-level violation rate (4.73% ± 1.18%) and episode-level violation rate (5.82% ± 3.94%) among all methods. Compared to the strongest static baseline (GK+Argo+Cosign), PATCHWEAVER reduces step violations by 52.5% (from 9.95% to 4.73%) and episode violations by 52.2% (from 12.18% to 5.82%). Compared to LLM-Agent, PATCHWEAVER reduces step violations by 79.1% (from 22.69% to 4.73%) and episode violations by 79.5% (from 28.46% to 5.82%).

Template+Rollout achieves the second-lowest violation rate (5.38% ± 1.29%), confirming that rollout simulation—not the LLM proposer—is the primary driver of violation reduction. LLM+Policy (LLM proposer with policy enforcement but no rollout simulation) achieves 11.84% ± 2.52%, showing that policy enforcement alone reduces violations compared to LLM-Agent (22.69%) but is insufficient without trajectory reasoning. Runbook achieves 6.09% ± 1.41% but incurs 28.4% longer TTR (624 s vs. 486 s) due to fixed wait intervals and inability to parallelize independent actions.

PATCHWEAVER's escalation rate (7.82% ± 1.56%) is higher than LLM-Agent (1.48% ± 0.52%) because PATCHWEAVER escalates ambiguous situations rather than proceeding optimistically; this is the core safety trade-off. The rollback rate for PATCHWEAVER (1.24% ± 0.68%) is 4.8× lower than LLM-Agent (5.96% ± 1.84%) and 3.6× lower than GK+Argo+Cosign (4.48% ± 1.42%), indicating that plan-level reasoning prevents late-stage failures. For hypothesis testing, we use a paired Wilcoxon signed-rank test over matched episodes with Holm correction; improvements versus GK+Argo+Cosign and LLM-Agent are significant at $p < 0.01$ (effect size $r > 0.5$).

We use the following abbreviations in Table 2: PW=PATCHWEAVER, T+R=Template+Rollout, L+P=LLM+Policy, and PW-H0=PATCHWEAVER with $H = 0$.

Table 2 summarizes safety and operational trade-offs. Step violations capture action-boundary compliance; episode violations capture error accumulation. LLM-Agent vs. L+P vs. PW isolates implicit reasoning, policy enforcement, and trajectory scoring. T+R is a strong non-LLM sanity check for structured workflows. Rollback and escalation quantify late failures and fail-closed caution, respectively.

Figure 3 shows the distribution of episode-level violations across 50 episodes. The CDF plot (left) demonstrates that

Table 2: Main results aggregated over four workloads (50 episodes each). Step Viol. = step-level policy violation rate (%); Ep. Viol. = episode-level violation rate (%); TTR = time-to-remediate (seconds); Esc. = escalation rate (%); Rollback = rollback rate (%). Values are mean ± std.

Method	Step Viol.	Ep. Viol.	TTR (s)	Esc.	Rollback
PW	**4.73**±1.18	**5.82**±3.94	**486**±38	7.82±1.56	**1.24**±0.68
T+R	5.38±1.29	7.16±3.68	502±36	7.24±1.42	1.56±0.72
Runbook	6.09±1.41	8.24±4.28	624±48	3.26±0.94	2.86±1.02
GK+Argo+Cos	9.95±2.21	12.18±5.86	542±46	4.18±1.14	4.48±1.42
Admission	10.92±2.37	14.26±6.42	598±52	2.94±0.86	5.12±1.64
L+P	11.84±2.52	15.62±5.14	528±41	4.86±1.18	4.82±1.48
PW-H0	13.86±2.74	18.24±5.92	556±42	5.14±1.28	4.18±1.36
LLM-Agent	22.69±4.86	28.46±8.92	512±44	1.48±0.52	5.96±1.84

PATCHWEAVER's violation distribution is shifted significantly left, with 90% of episodes having violation rates below 11.4% versus 21.5% for GK+Argo+Cosign and 42.4% for LLM-Agent. The boxplot (right) shows that PATCHWEAVER has the smallest interquartile range (IQR = 4.08%, Q1=2.18%, Q3=6.26%) and the lowest median (4.24%), indicating both lower central tendency and tighter variance. Notably, the tail behavior differs: PATCHWEAVER's p99 violation rate is 16.8%, whereas LLM-Agent's p99 reaches 51.3%, representing a 3.1× reduction in worst-case failures.

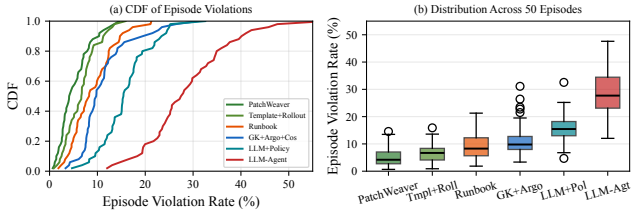

Figure 3: Distribution of episode-level violations across 50 episodes. (a) CDF showing PATCHWEAVER's distribution shifted left compared to all baselines. (b) Boxplot showing PATCHWEAVER's lower median (4.24%) and tighter IQR (4.08%) versus LLM-Agent (median 26.1%, IQR 12.5%).

Figure 3 reports distributions, not only averages. The CDF shows dominance across most of the support. The boxplot shows tighter variance under randomized conditions. Tail reduction matters because rare episodes drive rollbacks and incidents. CDF and boxplot provide cross-checks against cherry-picked statistics.

Per-workload analysis. Figure 4 breaks down step violation rates by workload. PATCHWEAVER achieves the lowest violation rate on all four workloads. The improvement is largest on Canary Gating (5.84% vs. 26.78% for LLM-Agent, −78.2%) and PolicyDrift Recovery (4.28% vs. 24.16% for LLM-Agent, −82.3%), both of which involve multi-step dependencies and mid-episode constraint changes. On ImageRebuild and RBAC Minimization, Template+Rollout performs comparably to PATCHWEAVER, confirming that deterministic proposers suf-

fice for well-defined workflows.

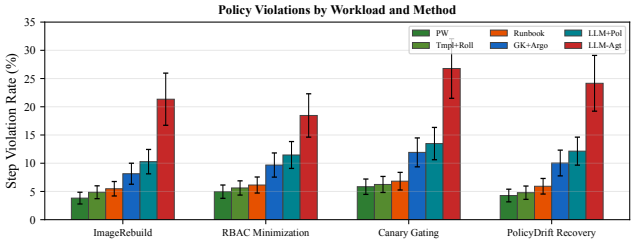

Figure 4: Step violation rate by workload and method. PATCHWEAVER achieves the lowest rate on all four workloads, with the largest improvements on Canary Gating and PolicyDrift Recovery.

Figure 4 localizes gains to specific failure modes. ImageRebuild is structured; T+R is close to PW. Canary Gating is interaction-heavy; trajectory scoring prevents traps. PolicyDrift stresses replanning under changing constraints and state. The breakdown avoids "aggregate-only" skepticism by showing heterogeneity. *Case study: constraint trap avoidance.* We observe constraint traps in 34 of 50 Canary Gating episodes when using LLM-Agent. In this scenario, production rollout requires a signed approval ticket available only after staging validation. LLM-Agent updates staging, initiates canary, then attempts full rollout without approval. The approval predicate evaluates to false; the policy engine blocks; the agent is trapped and triggers rollback, increasing TTR by 47%. PATCHWEAVER simulates the plan before execution, predicts the dead end at horizon $h = 3$, requests approval earlier, waits for approval, then proceeds with staged rollout—completing without a violation and 8% faster.

7.3 Security Stress Tests

We evaluate PATCHWEAVER against ten attack categories derived from real-world Kubernetes incidents and CVEs (e.g., CVE-2020-8554, CVE-2021-25741, CVE-2022-0185). Each attack is instantiated as 30 episodes with randomized timing and injection points. We report miss rate (unsafe state persists until episode end), block rate (immediate rejection), escalation rate (human review triggered), and **dwell time** (seconds the unsafe state persists before detection).

Table 3 shows quantitative results and separates immediate block from escalation. PATCHWEAVER blocks 88.7% and escalates 11.0% of adversarial inputs, with aggregate miss rate 0.33% (1 episode across 300 trials), versus 71.5% safe outcomes for GK+Argo+Cosign and 9.7% for LLM-Agent. Notably, PATCHWEAVER detects multi-step attacks (Aggregated Role Bypass, Multi-Step RBAC Escalation, Staged Provenance Bypass) that evade point-in-time admission checks because each individual step is locally admissible. PATCHWEAVER's mean dwell time is 0.84 s (p99 = 24.6 s) versus 96.4 s (p99 = 246.4 s) for GK+Argo+Cosign and 186.8 s (p99 = 524.8 s) for LLM-Agent.

Table 3 evaluates adversarial safety beyond steady-state outcomes. Block rate captures fully autonomous rejection; escalation rate captures cases routed to human review; miss rate captures worst-case persistence; dwell time captures exposure. Static attacks show parity with admission checks. Multi-step attacks isolate trajectory reasoning and drift awareness. Escalation is a deliberate valve when observability is ambiguous.

Figure 5 shows dwell time analysis. PATCHWEAVER's dwell time is near-zero for most attacks because trajectory simulation detects the unsafe state before execution. For Webhook Degradation, PATCHWEAVER escalates 53.4% of trials and misses 3.3% because the attack exploits telemetry delays indistinguishable from legitimate network latency; this is the main source of human review in Table 3. GK+Argo+Cosign misses dynamic attacks (Policy Drift 63.3%, Webhook 70.0%) because it lacks state-level drift detection. LLM-Agent misses 90.3% of attacks on average because it relies on implicit LLM reasoning without explicit policy predicates.

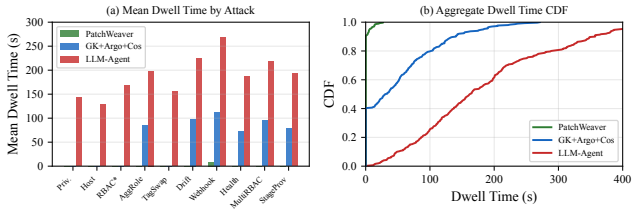

Figure 5: Dwell time analysis. (a) Mean dwell time by attack category. (b) Aggregate dwell time CDF showing PATCHWEAVER near-zero dwell versus extended exposure for baselines.

Figure 5 shows exposure time, not only misses. Baselines may recover eventually but still expose unsafe windows. The aggregate CDF highlights tail risk in dwell time. PW reduces exposure by pre-execution rejection or escalation. The figure cross-checks Table 3 method ordering.

7.4 Ablation and Overhead

Table 4 shows ablation results isolating each major component. Removing rollout simulation (H=0) increases step violations from 4.73% to 13.86% (+193%), confirming multi-step lookahead as the primary driver. Removing drift detection increases violations to 9.74% (+106%), especially for PolicyDrift (4.28% to 14.86%). Removing provenance checks increases violations to 7.82% (+65%), and removing least-privilege enforcement increases violations to 10.48% (+122%). Escalation rates decrease when components are removed (e.g., H=0: 7.82% to 3.26%), reflecting reduced caution rather than improved safety.

Table 3: Attack evaluation over 10 categories (30 episodes each). Block/Esc/Miss are percentages; Dwell is mean seconds unsafe state persists. PATCHWEAVER separates autonomous blocking from human escalation.

	PATCHWEAVER				GK+Argo		LLM-Agt	
Attack Category	Block	Esc	Miss	Dwell	Miss	Dwell	Miss	Dwell
Privileged Container	100.0	0.0	0.0	0.0	0.0	0.0	86.7	142.8
HostPath Mount	100.0	0.0	0.0	0.0	0.0	0.0	83.3	128.6
RBAC Wildcard	100.0	0.0	0.0	0.0	0.0	0.0	90.0	168.4
Aggregated Role Bypass	96.7	3.3	0.0	0.0	23.3	84.2	93.3	198.4
Image Tag Swap	100.0	0.0	0.0	0.0	0.0	0.0	80.0	156.2
Policy Drift Mid-Rollout	86.7	13.3	0.0	0.0	63.3	98.6	96.7	224.6
Webhook Degradation	43.3	53.4	3.3	8.4	70.0	112.8	100.0	268.4
Delayed Health Signal	76.7	23.3	0.0	0.0	30.0	72.4	86.7	186.4
Multi-Step RBAC Escal.	93.3	6.7	0.0	0.0	46.7	94.8	96.7	218.4
Staged Prov. Bypass	90.0	10.0	0.0	0.0	36.7	78.4	90.0	192.6
Aggregate	**88.7**	**11.0**	**0.33**	**0.84**	28.5	96.4	90.3	186.8

Table 4: Ablation study (step violation rate %). Removing rollout simulation has the largest impact (+193%).

Configuration	Step Viol. (%)	Δ
Full System (PATCHWEAVER)	4.73 ± 1.18	—
No Rollout (H=0)	13.86 ± 2.74	+193%
No Drift Detection	9.74 ± 2.08	+106%
No Provenance Check	7.82 ± 1.64	+65%
No Least-Privilege	10.48 ± 2.18	+122%

Table 4 isolates which components are necessary for policy-bounded autonomy rather than merely sufficient. Removing rollout simulation produces the largest regression, confirming that local admissibility checks cannot prevent trap-inducing trajectories. Drift detection has a smaller aggregate effect but is concentrated in PolicyDrift episodes where policies change mid-rollout, which matches the intended design. Provenance and least-privilege checks contribute primarily to adversarial robustness, preventing remediation from becoming a supply-chain or privilege-escalation channel. Importantly, reduced escalation in weaker configurations should not be interpreted as an operational improvement; it is a symptom of unsafe optimism under uncertainty. We study sensitivity to planning horizon H and violation penalty λ_v. Increasing H from 0 to 6 reduces violations (13.86% to 4.73%). Returns diminish beyond H=6 once dominant gates are covered. Increasing λ_v reduces violations but increases escalation. We use H=6 and λ_v=1.0 as practical defaults. *Control-plane overhead.* Figure 6 shows admission latency and resource usage. At 32 predicates (our default policy set), p50/p95/p99 admission latency is 7.42/18.64/31.28 ms, well below the 30 s webhook timeout threshold. Latency scales linearly with policy count; at 48 predicates, p99 remains under 50 ms. Total resource usage is 0.86 CPU cores and 928 MB memory (dominated by LLM API client: 0.43 cores, 486 MB for request queuing and response parsing), 2–3× higher than standalone Gatekeeper but within typical controller budgets. Decision-time latency is 1.2–3.8 s depending on LLM response time, acceptable for remediation workflows where human review often takes minutes. We do not observe webhook timeouts at 32 predicates under bursty workloads (100 concurrent episodes).

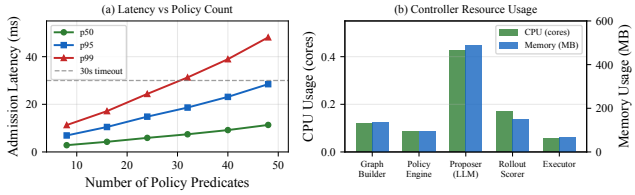

Figure 6: Control-plane overhead. (a) Admission latency scales linearly with policy count; p99 remains under 50 ms at 48 predicates. (b) Resource usage by component; LLM API client dominates.

Figure 6 summarizes deployability overhead. Admission latency scales near-linearly with predicate count. p99 latency stays well below webhook timeouts for fail-closed use. Most CPU/memory cost is in proposer integration, not policies. We report both latency and resources to make trade-offs explicit. *Threats to validity.* Results are scoped to Kubernetes remediation and CHANGESPEC constraints; generalization to other platforms requires adapter work. Approvals can dominate wall-clock time; we control arrival times and log escalations. Cluster load may affect outcomes; we isolate the control plane and report tail latency. LLM baselines depend on prompting; we fix temperature to 0.0 and keep action vocabulary identical. Security tests derive from real CVEs and documented incident patterns; we implement parameterized instantiations to exercise each exploit under controlled timing and injection points.

8 Discussion and Limitations

Policy completeness. PATCHWEAVER is only as strong as the encoded policies. Organizations differ in governance requirements, and CHANGESPEC is designed to be extensible. If policies are mis-specified, PATCHWEAVER may over-escalate (false positives) or permit risky behavior not captured by predicates (false negatives). The explanation mechanism helps identify mis-specifications by surfacing denial reasons, but policy correctness remains the operator's responsibility.

CHANGESPEC *expressiveness and adoption.* CHANGESPEC is not intended to subsume general policy languages. OPA/Rego and Cedar are better choices for organizations that need broad policy ecosystems, mature tooling, or formally analyzed authorization semantics. CHANGESPEC is useful when remediation policies must be evaluated over typed actions, graph relationships, evidence freshness, and predicted future states. Policies that depend on ambiguous human intent, natural-language exceptions, long-term business risk, or cross-organization negotiations are outside its intended scope and should be represented as escalation gates. Inconsistent rules can still occur; PATCHWEAVER detects common syntactic and scope-level contradictions at load time and escalates unresolved runtime conflicts, but it does not prove global satisfiability for all future graph states.

Proposer flexibility vs. reliability. An LLM proposer can generate diverse candidate plans covering edge cases that template libraries miss, but it can also suggest unsafe actions (e.g., overly broad RBAC grants). The policy engine rejects such actions before execution; however, organizations concerned about LLM latency (1.2–3.8 s per decision) or API cost can substitute template-based proposers. Our evaluation shows Template+Rollout achieves comparable violation rates (5.38% vs. 4.73% for PATCHWEAVER) at lower latency, though it may miss novel remediation paths. This result also frames model sensitivity: stronger frontier models may propose better plans, but they do not remove the need for policy evaluation because safety-critical failures arise from missing evidence, stale state, and governance ordering, not only from reasoning quality. We did not evaluate every available model; the proposer interface is intentionally replaceable.

State accuracy and telemetry gaps. Rollouts depend on state graph accuracy. PATCHWEAVER mitigates drift via continuous reconciliation from Kubernetes watch streams and a conservative drift threshold (τ=0.15). Severe telemetry gaps (>30 s) can still lead to unsafe decisions; in those cases, PATCHWEAVER escalates rather than guessing. Adversaries can target observability to increase drift and escalation volume. Defending the governance pathway against denial-of-service (e.g., via rate limiting, deduplication, connector hardening) remains an important direction for future work.

False positives and alert fatigue. Fail-closed systems can block or escalate legitimate changes. In our main workload experiments, PATCHWEAVER escalates 7.82% of actions; in the adversarial stress tests, 11.0% of inputs are escalated and 88.7% are immediately blocked. Most escalations occur when evidence is ambiguous rather than when a static manifest is clearly unsafe. This trade-off is intentional, but it can create alert fatigue if policy freshness windows are too strict or telemetry is unreliable. Deployments should tune escalation thresholds by service criticality and monitor repeated escalation reasons as policy-engineering feedback.

Multi-cluster and federation. Our evaluation uses a single-cluster setup (12 nodes). Production environments often span multiple clusters with multi-tenant namespaces. PATCHWEAVER supports federation via replicated state graphs synchronized through a coordination layer; we have not evaluated cross-cluster coordination overhead or consistency guarantees, which we leave for future work.

Failure modes. PATCHWEAVER can fail due to policy incompleteness, state desynchronization, proposer failure (LLM timeout or malformed output), or executor failure (connector errors). All failure modes are logged with structured explanations, enabling post-incident analysis. The fail-closed design ensures that failures result in escalation rather than uncontrolled execution.

Human oversight. PATCHWEAVER is not intended to remove human oversight for high-impact changes. Instead, it reduces avoidable violations and produces pre-execution explanations that make approval decisions faster and more informed. The escalation rate (7.82% in our evaluation) represents cases where PATCHWEAVER correctly identifies ambiguity and requests human review—a feature, not a limitation.

Benchmark scope. Our custom workloads were necessary to exercise governed remediation constraints not present in current public benchmarks. This creates the usual risk that a system and workload co-evolve. We mitigate this by including strong baselines, ablations, randomized episodes, attack categories tied to documented Kubernetes incidents, and per-workload breakdowns. Future benchmark extensions should add policy/evidence annotations to ITBench, MicroRemed, and SRE-Bench so governed-remediation systems can be compared on shared tasks.

9 Related Work

Patch management guidance. Operational guidance emphasizes that effective patch management requires governance, planning, and verification [26,30]. PATCHWEAVER targets the automation layer that executes governed remediation workflows under explicit constraints, bridging the gap between high-level guidance and low-level execution.

Policy-as-code and authorization languages. Gatekeeper/OPA [14, 28] and Kyverno [19] enforce policies at Kubernetes admission time. Cedar provides a purpose-built, analyzable authorization language for principal/action/resource decisions [2]. These tools can express many of the individual predicates used by PATCHWEAVER

and can provide useful violation messages. Our claim is therefore not that ChangeSpec is more expressive as a standalone language. The missing layer is trajectory evaluation: admission and authorization tools decide whether a request is permitted now, while PatchWeaver simulates whether a remediation plan remains permitted as approvals, health, artifacts, and policy versions evolve. Our evaluation shows that Gatekeeper+Argo+Cosign still misses 28.5% of attacks, particularly dynamic attacks like Policy Drift and Webhook Degradation, because it lacks trajectory-level and drift-aware reasoning.

Kubernetes security analysis. Recent work has developed automated techniques to detect and remediate excessive RBAC permissions [12], protect secrets when permissions are overly broad [31], and analyze Kubernetes networking security [24]. PatchWeaver complements these by enforcing least privilege *during* multi-step remediation workflows, preventing the remediation pathway itself from becoming a privilege-escalation channel.

Attack graphs and security modeling. Structured models such as attack graphs capture reachability and attacker paths [27, 29, 34]. PatchWeaver instead models the *defender's* remediation workflow as a constrained decision process, reasoning about how governance constraints interact across steps. This is a dual perspective: attack graphs model what an attacker can reach; PatchWeaver models what a defender can safely do.

Provenance and intrusion detection. Whole-system provenance supports intrusion detection and investigation [6, 16], though provenance-based detectors can be attacked via mimicry [11], motivating robust monitoring [37]. PatchWeaver uses a decision log to audit *defender* actions and enforce governance, complementing provenance-based detection of attacker activity.

Supply chain security. Modern clusters enforce artifact trust through image signatures and provenance checks: Sigstore [35] provides keyless signing; in-toto [38] specifies supply-chain layouts; SLSA [36] defines security levels. Remediation workflows must ensure that build/sign/rollout gates occur in correct order under approval constraints. PatchWeaver incorporates artifact-trust predicates and uses rollouts to ensure compliant sequencing, complementing progressive delivery controllers like Argo Rollouts [3] and Flagger [10]. The supply chain threat model includes malicious packages in npm and PyPI [8, 9, 13, 15, 42]. SBOMs structure supply-chain evidence [18, 40], and GitHub Actions introduce workflow-level automation with its own failure modes [5, 32]. PatchWeaver treats CI evidence and approvals as first-class, versioned inputs to policy decisions.

Workflow graphs and planning frameworks. LangGraph and related LangChain tooling model stateful agent workflows as graphs with nodes, edges, persistence, and human-in-the-loop control [20, 21]. These frameworks are complementary orchestration substrates. PatchWeaver's graph is different: it is a typed graph of Kubernetes resources, governance evidence, and dependencies, not only a graph of agent computation steps. Recent policy-constrained planning proposals prune non-compliant plans during generation [25]; PatchWeaver instead permits any proposer but evaluates candidate trajectories against executable governance and drift-aware state before execution. This separation lets existing workflow tools serve as proposers while preserving a policy-centered enforcement boundary.

LLM agents for operations. Recent work applies LLM agents to code generation, debugging, and operations. The PAAR framework also studies policy-aware autonomous remediation for Kubernetes, combining LLM/ML diagnosis, an RL action generator, and OPA-based plan validation [39]. PatchWeaver differs by focusing on vulnerability-remediation trajectories under explicit change-management evidence, using the same typed predicates for current-state admission, predicted rollout scoring, drift-triggered replanning, and audit. However, LLM agents lack explicit policy enforcement and trajectory-level reasoning, leading to high violation rates in constrained settings. Our evaluation shows LLM-Agent (GPT-4-turbo with ReAct prompting) achieves 22.69% step violation rate and misses 90.3% of attacks. Adding policy enforcement without rollout simulation (LLM+Policy) reduces violations to 11.84%, but trajectory-level reasoning (PatchWeaver) is needed to achieve 4.73%. PatchWeaver treats the LLM as an untrusted proposer, not a trusted executor.

Operational and remediation benchmarks. ITBench, MicroRemed, and SRE-Bench provide useful task suites for evaluating incident response and remediation agents [1, 17, 23]. They do not currently encode the specific governance evidence needed for our claims, such as approval freshness, maintenance windows, artifact-trust gates, and mid-rollout policy drift. Extending them with policy annotations would make them a natural next benchmark target for PatchWeaver-like systems.

Positioning. Remediation is a strict setting because the same action surface is both operationally high-impact and adversarially contested. PatchWeaver therefore evaluates compliance over trajectories, treating governance as executable, versioned state rather than unstructured instructions. This framing connects policy-as-code, rollout controllers, and agentic proposers into a unified, auditable system.

10 Conclusion

PatchWeaver provides policy-bounded autonomy for vulnerability remediation in Kubernetes environments. We make remediation state explicit via a typed Kubernetes-centric graph. We encode change-management rules as executable predicates in ChangeSpec with audit-grade explanations. We score candidate plans via short-horizon rollouts under policy constraints. We detect drift between predicted and ob-

served state and trigger replanning when divergence exceeds threshold.

With these mechanisms, PATCHWEAVER reduces step-level policy violations by 4.8× compared to an LLM-agent baseline (4.73% vs. 22.69%) and by 2.1× compared to a strong combination of Gatekeeper, Cosign, and Argo Rollouts (4.73% vs. 9.95%). The key insight is that safe remediation requires *trajectory-level* reasoning over executable policies. Point-in-time admission checks cannot detect constraint traps—states where locally admissible actions lead to dead ends that force later violations.

We evaluate on four remediation workloads (image patching, RBAC minimization, canary gating, policy drift recovery) across 200 episodes with randomized initial conditions. In security stress tests across 10 attack categories, PATCHWEAVER blocks 88.7% and escalates 11.0% of adversarial inputs with aggregate miss rate 0.33% and mean dwell time 0.84 s, versus 28.5% miss rate and 96.4 s dwell time for the toolchain baseline. PATCHWEAVER degrades gracefully under telemetry delays by escalating ambiguous cases rather than guessing.

Control-plane overhead remains practical: at 32 policy predicates, admission latency is 7.42 ms p50 and 31.28 ms p99. Decision-time latency is 1.2–3.8 s, dominated by LLM response time; organizations concerned about latency or cost can substitute template-based proposers with comparable violation rates (5.38% for Template+Rollout vs. 4.73% for PATCHWEAVER). The ablation study confirms that rollout simulation and drift detection are the most impactful components: removing rollout simulation (H=0) increases violations from 4.73% to 13.86% (+193%); removing drift detection increases violations to 9.74% (+106%).

We believe the path to safe agentic security automation is not to rely on better prompting or larger models alone. Instead, we couple flexible proposal with explicit, executable governance; conservative trajectory simulation; and auditable decision logs that link every action to policy predicates and evidence. PATCHWEAVER demonstrates this principle in a practical, Kubernetes-native system. The artifact package contains the CHANGESPEC specification, the policy checker, connectors, evaluation scripts, and containerized reproducibility runner.

A Ethical Considerations

This work studies automation for security remediation workflows in Kubernetes-like environments. We do not collect data from human subjects and do not use private user data; our evaluation uses controlled workloads and publicly documented vulnerabilities.

Stakeholders and potential harms. Stakeholders include cluster operators (SRE/security teams), platform owners, application developers, and end users impacted by service reliability. Potential harms include (i) availability incidents from incorrect or overly aggressive remediation actions, (ii) privilege misuse if a controller or its credentials are compromised, and (iii) supply-chain risk if automated patching pulls untrusted artifacts.

Mitigations and safeguards. PATCHWEAVER is designed to fail closed: actions that cannot be justified under the declared change-management policy are rejected rather than executed. Connectors are scoped to least-privilege credentials and can be configured to operate in a read-only or dry-run mode for auditing and staged rollout. For ambiguous or high-impact actions, PATCHWEAVER supports explicit escalation to a human approval gate, and all actions are logged in an append-only audit trail that links decisions to policy versions and supporting evidence.

Responsible release. Because remediation and attack scripts can be misused, the artifact package includes documentation on safe execution, clear default guardrails (rate limits and blast-radius bounds), and guidance for running experiments in isolated test clusters.

B Open Science

Artifact availability. All artifacts required to evaluate the paper are permanently archived on Zenodo with DOI https://doi.org/10.5281/zenodo.20477170. The corresponding Zenodo record is available at https://zenodo.org/records/20477170, and the same bundle is linked from the HotCRP artifact submission for review and re-verification.

What is included. The artifact bundle contains: (i) the CHANGESPEC specification and parser; (ii) the policy checker and control-plane implementation; (iii) connectors and workload definitions; (iv) attack scripts used in the security stress tests; (v) evaluation harnesses, plotting scripts, and figure regeneration commands; and (vi) raw logs and aggregated metrics with pinned random seeds and configuration files.

How to reproduce results. The repository provides step-by-step instructions to build and run the benchmarks, including the exact commit hash, pinned dependencies, and hardware/software prerequisites needed to reproduce the main tables and figures.

References

[1] AgentKube. SRE-Bench: Benchmarking site reliability engineering agents. https://github.com/agentkube/SRE-bench, 2026. Accessed 2026-05-26. URL: https://github.com/agentkube/SRE-bench.

[2] Amazon Web Services. Cedar: An open-source language for access control. https://www.cedarpolicy.

com/, 2026. Accessed 2026-05-26. URL: https://www.cedarpolicy.com/.

[3] Argo. Argo rollouts. https://argoproj.github.io/rollouts/, 2026. Accessed 2026-01-15. URL: https://argoproj.github.io/rollouts/.

[4] AXELOS. ITIL 4: It service management. https://www.axelos.com/best-practice-solutions/itil, 2019.

[5] Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–12. ACM, February 2024. URL: http://dx.doi.org/10.1145/3597503.3623303, doi:10.1145/3597503.3623303.

[6] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical intrusion detection and investigation using whole-system provenance. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3533–3551. IEEE, May 2024. URL: http://dx.doi.org/10.1109/sp54263.2024.00005, doi:10.1109/sp54263.2024.00005.

[7] CISA. CISA known exploited vulnerabilities catalog. https://www.cisa.gov/known-exploited-vulnerabilities-catalog, 2026. Accessed 2026-01-15. URL: https://www.cisa.gov/known-exploited-vulnerabilities-catalog.

[8] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings 2021 Network and Distributed System Security Symposium*, NDSS 2021. Internet Society, 2021. URL: http://dx.doi.org/10.14722/ndss.2021.23055, doi:10.14722/ndss.2021.23055.

[9] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kastner. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, May 2021. URL: http://dx.doi.org/10.1109/icse43902.2021.00121, doi:10.1109/icse43902.2021.00121.

[10] Flagger. Flagger: Progressive delivery kubernetes operator. https://flagger.app/, 2026. Accessed 2026-01-15. URL: https://flagger.app/.

[11] Akul Goyal, Xueyuan Han, Gang Wang, and Adam Bates. Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems. In *Proceedings 2023 Network and Distributed System Security Symposium*, NDSS 2023. Internet Society, 2023. URL: http://dx.doi.org/10.14722/ndss.2023.24207, doi:10.14722/ndss.2023.24207.

[12] Yue Gu, Xin Tan, Yuan Zhang, Siyan Gao, and Min Yang. Epscan: Automated detection of excessive rbac permissions in kubernetes applications. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3199–3217. IEEE, May 2025. URL: http://dx.doi.org/10.1109/sp61157.2025.00011, doi:10.1109/sp61157.2025.00011.

[13] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, September 2023. URL: http://dx.doi.org/10.1109/ase56229.2023.00135, doi:10.1109/ase56229.2023.00135.

[14] Ben Hindman, Andy Konwinski, Matei Zaharia, and Ion Stoica. Open policy agent: Policy-based control for cloud native environments. In *Proceedings of the USENIX Annual Technical Conference*, 2018.

[15] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. Spiderscan: Practical detection of malicious npm packages based on graph-based behavior modeling and matching. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1146–1158. ACM, October 2024. URL: http://dx.doi.org/10.1145/3691620.3695492, doi:10.1145/3691620.3695492.

[16] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2620–2638. IEEE, May 2023. URL: http://dx.doi.org/10.1109/sp46215.2023.10179405, doi:10.1109/sp46215.2023.10179405.

[17] ITBench. ITBench: Benchmarking ai agents for it automation. https://github.com/itbench-hub/ITBench, 2026. Accessed 2026-05-26. URL: https://github.com/itbench-hub/ITBench.

[18] Changguo Jia, Nianyu Li, Kai Yang, and Minghui Zhou. Sit: An accurate, compliant sbom generator with incremental construction. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 13–16. IEEE, April 2025. URL: http://dx.doi.org/10.1109/icse-companion66252.2025.00013, doi:10.1109/icse-companion66252.2025.00013.

[19] Kyverno. Kyverno: Kubernetes policy management. https://kyverno.io/, 2026. Accessed 2026-01-15. URL: https://kyverno.io/.

[20] LangChain. Agents and stateful workflows. https://www.langchain.com/use-case/agents, 2026. Accessed 2026-05-26. URL: https://www.langchain.com/use-case/agents.

[21] LangChain. Langgraph documentation. https://docs.langchain.com/oss/python/langgraph, 2026. Accessed 2026-05-26. URL: https://docs.langchain.com/oss/python/langgraph.

[22] Frank Li and Vern Paxson. Large-scale analysis of patch availability and timelines. In *Proceedings of the USENIX Security Symposium*, pages 1313–1330, 2017.

[23] LLM4AIOps. MicroRemed: Microservice remediation benchmark. https://github.com/LLM4AIOps/MicroRemed, 2026. Accessed 2026-05-26. URL: https://github.com/LLM4AIOps/MicroRemed.

[24] Francesco Minna, Agathe Blaise, Filippo Rebecchi, Balakrishnan Chandrasekaran, and Fabio Massacci. Understanding the security implications of kubernetes networking. *IEEE Security & Privacy*, 19(5):46–56, September 2021. URL: http://dx.doi.org/10.1109/msec.2021.3094726, doi:10.1109/msec.2021.3094726.

[25] Niranjan M M. Policy-constrained plan generation for autonomous ai agents. Technical Disclosure Commons, 2026. Posted March 16, 2026; accessed 2026-05-28. URL: https://www.tdcommons.org/dpubs_series/9536/.

[26] NIST. NIST special publication 800-40: Guide to enterprise patch management planning. https://csrc.nist.gov/publications/detail/sp/800-40/rev-3/final, 2013. Revision 3. URL: https://csrc.nist.gov/publications/detail/sp/800-40/rev-3/final.

[27] Steven Noel and Sushil Jajodia. Advances in topological vulnerability analysis. In *Proceedings of the IEEE Conference on Technologies for Homeland Security*, pages 124–129, 2009. doi:10.1109/catch.2009.19.

[28] Open Policy Agent. Gatekeeper: Policy controller for kubernetes. https://github.com/open-policy-agent/gatekeeper, 2026. Accessed 2026-01-15. URL: https://github.com/open-policy-agent/gatekeeper.

[29] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *Proceedings of the USENIX Security Symposium*, pages 113–128, 2005. URL: https://dblp.org/rec/conf/uss/OuGA05.html.

[30] OWASP. OWASP vulnerability management guide. https://owasp.org/www-project-vulnerability-management-guide/, 2023. Accessed 2026-01-15. URL: https://owasp.org/www-project-vulnerability-management-guide/.

[31] Maryam Rostamipoor, Aliakbar Sadeghi, and Michalis Polychronakis. Kubekeeper: Protecting kubernetes secrets against excessive permissions. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, pages 322–338. IEEE, June 2025. URL: http://dx.doi.org/10.1109/eurosp63326.2025.00027, doi:10.1109/eurosp63326.2025.00027.

[32] Nuno Saavedra, André Silva, and Martin Monperrus. Gitbug-actions: Building reproducible bug-fix benchmarks with github actions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 1–5. ACM, April 2024. URL: http://dx.doi.org/10.1145/3639478.3640023, doi:10.1145/3639478.3640023.

[33] Shazibul Islam Shamim. Mitigating security attacks in kubernetes manifests for security best practices violation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, pages 1689–1690. ACM, August 2021. URL: http://dx.doi.org/10.1145/3468264.3473495, doi:10.1145/3468264.3473495.

[34] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 273–284, 2002. doi:10.1109/secpri.2002.1004377.

[35] Sigstore. Cosign: Container signing, verification, and storage. https://github.com/sigstore/cosign, 2026. Accessed 2026-01-15. URL: https://github.com/sigstore/cosign.

[36] SLSA. SLSA: Supply-chain levels for software artifacts. https://slsa.dev/, 2022. Accessed 2026-01-15. URL: https://slsa.dev/.

[37] Azadeh Tabiban, Heyang Zhao, Yosr Jarraya, Makan Pourzandi, Mengyuan Zhang, and Lingyu Wang. Provtalk: Towards interpretable multi-level provenance analysis in networking functions virtualization (nfv). In *Proceedings 2022 Network and Distributed System Security Symposium*, NDSS 2022. Internet Society, 2022. URL: http://dx.doi.org/10.14722/ndss.2022.23103, doi:10.14722/ndss.2022.23103.

[38] Santiago Torres-Arias, Hammad Afzali, Trishank K. Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the USENIX Security Symposium*, pages 1393–1410, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[39] Eleanor Vance, Marcus Thorne, and Anya Petrova. Policy-aware autonomous remediation in kubernetes clusters using llms and machine learning. ResearchGate preprint, 2026. Accessed 2026-05-28. URL: https://www.researchgate.net/publication/399658652_Policy-Aware_Autonomous_Remediation_in_Kubernetes_Clusters_Using_LLMs_and_Machine_Learning.

[40] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, May 2023. URL: http://dx.doi.org/10.1109/icse48619.2023.00219, doi:10.1109/icse48619.2023.00219.

[41] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.

[42] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Leveraging large language models to detect npm malicious packages. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2625–2637. IEEE, April 2025. URL: http://dx.doi.org/10.1109/icse55347.2025.00146, doi:10.1109/icse55347.2025.00146.