International Journal of Engineering Inventions
e-ISSN: 2278-7461, p-ISSN: 2319-6491
Volume 15, Issue 11 [November 2026] PP: 102-113

Towards Accountable Autonomic Computing: Gated Multi-Agent Program Repair for Cloud-Native Incidents

Likith Lochan U, Kruthik N, Avinash M V, Lohit Yadav R, and S Vaibhava Sindhu

Dept. of Computer Science and Engineering, A.P.S. College of Engineering, Bengaluru, INDIA
Corresponding Author: Lohit Yadav R

ABSTRACT: *Monitoring tools indicate when an issue has occurred within a service. Engineers are responsible for diagnosing, fixing and deploying the service. A model can be used to generate patches; however, these are not guaranteed to be correct, can be influenced by a text injection in a log line or can pass tests by removing them. We present GhostOps, our proof-of-concept AIOps platform that uses the MAPE-K loop. The incident is diagnosed by three agents and a patch is produced, which must pass several independent checks. These include a prompt injection protection, Open Policy Agent allows list, blast radius check and network isolated Docker sandbox with regression test that must fail and then pass. A risk score between 0-100 is used to decide which path the fix will go down. An audit log with chained hashes and signed evidence bundles are used to make sure all decisions can be verified. In an offline evaluation using a fake model with no cluster there were three scenarios with fixes requiring between 2-4 lines of code and took between 2.6-9.0s. Of the 530 tests run, 529 passed. One test failed which meant one regression proof could not be verified.*

Date of Submission: 02-11-2026 Date of acceptance: 12-11-2026

I. INTRODUCTION

The system consists of many small components which run on clusters and are frequently redeployed. If a component fails, the engineer receives an alert. They will then analyse logs and metrics to form a hypothesis and locate the source file. The engineer will make the change, test it, and wait for approval. The first step is well catered for by monitoring systems such as Prometheus [17]. This system will run alert rules on real metrics and alert the correct person quickly. All subsequent steps are manual. Site reliability engineering aims to reduce toil by automating this process. Tools such as blameless post-mortems are used to learn from failures [1]. Autonomic computing has provided a control loop to achieve this goal, where the system monitors itself, analyse information, create a plan and execute it using a shared knowledge base. This is known as MAPE-K [2].

Large language models provide an easy way to create a closed loop as they can read the stack trace and source file mentioned. However, there are a few risks. The model may be incorrect, it may be guided by text inserted into the log lines, and the patch may pass the test suite as it removed the failing tests. In terms of engineering, the problem becomes how we can make the model output a patch and what evidence is required.

Most research into AIOps has focussed on the initial parts of the incident cycle. Whilst applying AI to real systems can be challenging [3], it provides value by detecting, diagnosing, and eventually repairing incidents automatically. Notaro et al. [4] performed a literature review into failure management techniques and found most research focuses on prediction and detection. Diagnosis and remediation are largely performed manually. Ahmed et al. [5] have performed an evaluation of models for recommending root causes and mitigation steps for cloud incidents. Chen et al. [6] describe a root cause assistant which can look up incident details before calling the model. Both studies show that models can extract useful diagnostic information from incidents if they are grounded.

Automated program repair can provide the second part of the incident. Le Goues et al. [7] provide a summary of the field and show how patches are generated and tested. A common issue is overfitting, where the patch will pass the tests, but the underlying bug will not be fixed. Jimenez et al. [8] created the SWE-

bench benchmark to test if models can fix real GitHub issues. Yang et al. [9] showed that the interface provided to the model driven software agent has a large impact on the success rate. This shows that patches can be verified and overfitting is when the patch is not designed.

Security research explains why telemetry should not be passed into the model without inspection. Perez and Ribeiro [10] describe attacks that can cause language models to ignore their instructions. Greshake et al. [11] describe indirect injection attacks where the malicious instruction is in external data. This includes alert messages and log lines as the external party can influence what is written by the service. Techniques can be applied from mature software delivery and security engineering practices. Open Policy Agent (OPA) [12] allows policies to be written in code using the Rego language. Canary releases [13] are used to apply a change to a small percentage of traffic. Chaos engineering [14] injects faults into a system to test for resilience. Audit logs [15] are used, where each log entry is cryptographically bound to the previous one. This will allow any tampering to be detected. Ed25519 is defined in the RFC 8032 [16].

The research shows that models can be used for diagnostics and patching. Techniques can be applied to ensure changes are gated. However, we could not find any system which would take an active alert, treat the model as untrusted, ensure the patch proves itself with more than just its own tests and provide evidence for a third party to verify. This is based on the sixteen works in the references. Section 3.4 shows a comparison of capabilities between alert-only monitoring, runbook automation, LLM coding assistants and LLM site reliability agents. The latter two are general patterns and do not represent a specific product.

We present GhostOps, a proof-of-concept incident remediation platform. We have created an architecture to convert an alert from Prometheus into a remediation run with no human interaction. We use a FastAPI backend to receive the webhook from Alertmanager. This will combine duplicate and related alerts into an incident and begin a three-agent pipeline, using CrewAI [18] and Claude [20]. The Diagnostic Reasoner will classify the root cause, the Patch Generator will create the smallest diff and the Validation Officer will make their decision using deterministic checks and the exit status from the sandbox.

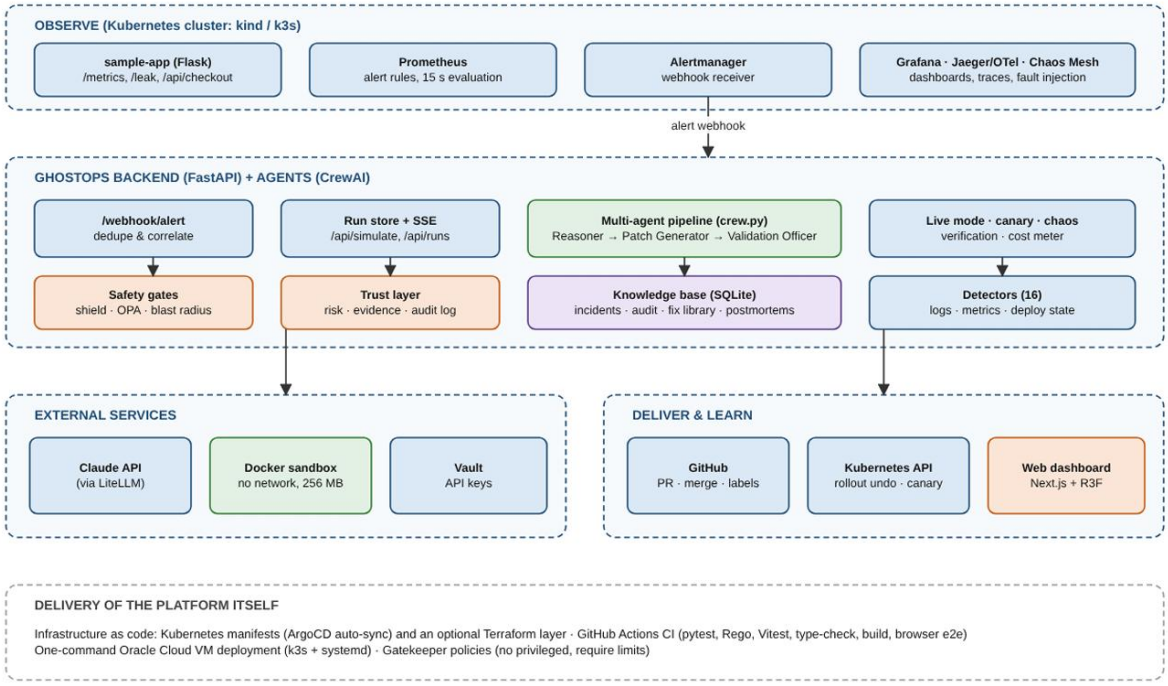
We have created a series of gates for the patch to pass. The shield will ensure any injected instructions or hidden characters in the incoming telemetry are blocked. OPA is used to ensure any patch only affects one application directory and manifest. The blast radius gate will ensure any patch does not affect more than one file, more than 40 lines of code, weakens any tests, adds any dependencies or calls any risky functions. The project tests are run in a Docker sandbox with network isolation. The generated regression test must fail before the fix and pass after. This is followed by a counterfactual replay and canary pod. The risk score between 0-100 will decide whether it is automatically merged to staging, created as a pull request or pull request with escalation to a human. The kill switch can override any decisions. Finally, we have added accountability by recording each decision in an audit log with SHA-256 chaining. Any validated fixes will include an evidence bundle signed with Ed25519. A blameless post-mortem is generated for each run.

We have performed an offline evaluation of the implementation. The platform has sixteen deterministic detectors and seventeen use cases. Four are fully remediated and thirteen are detected and diagnosed. Three of the four were evaluated using the bundled sample application. We ran 530 tests in Python, with 529 passing. We did not use a Kubernetes cluster, Prometheus, Alertmanager or GitHub write access, as we mocked the model call. This shows that the pipeline and gates operate as intended for these scenarios. It does not show the accuracy of Claude in diagnostics, how long it takes to actually repair the incident, or how the canary, chaos and post deployment stages operate in a live cluster. The logic for these

stages is covered in the stubbed backend tests. Section 2 describes the architecture, safety gates, and accountability mechanisms. Section 3 describes the results, threats to validity and comparison of capabilities. Section 4 concludes.

II. EXPERIMENTAL SETUP

An adaptation of the MAPE-K loop is used [2]. Prometheus [17] collects metrics from the sample application and kubeletcAdvisor endpoint, and the alert rules are evaluated every 15s. The two key rules used are working_set memory > 80% of limit and 5xx status codes on more than 20% of requests from one release. These rules need to be true for 30s. If they are, then the alertmanager creates a webhook that triggers a fastapi backend, this responds instantly and runs the pipeline as a background process. This collects the latest logs from the pods, memory metrics and source files to create a failure context for the agents. The agents then perform analysis and planning, whilst the gates and delivery determine if and how to deliver. The SQLite database acts as a knowledge base for the system. The code base is approximately 6700 lines of Python, 3100 lines of TypeScript and has Kubernetes, ArgoCD, Terraform and Gatekeeper configurations along with a 4-job CI pipeline.

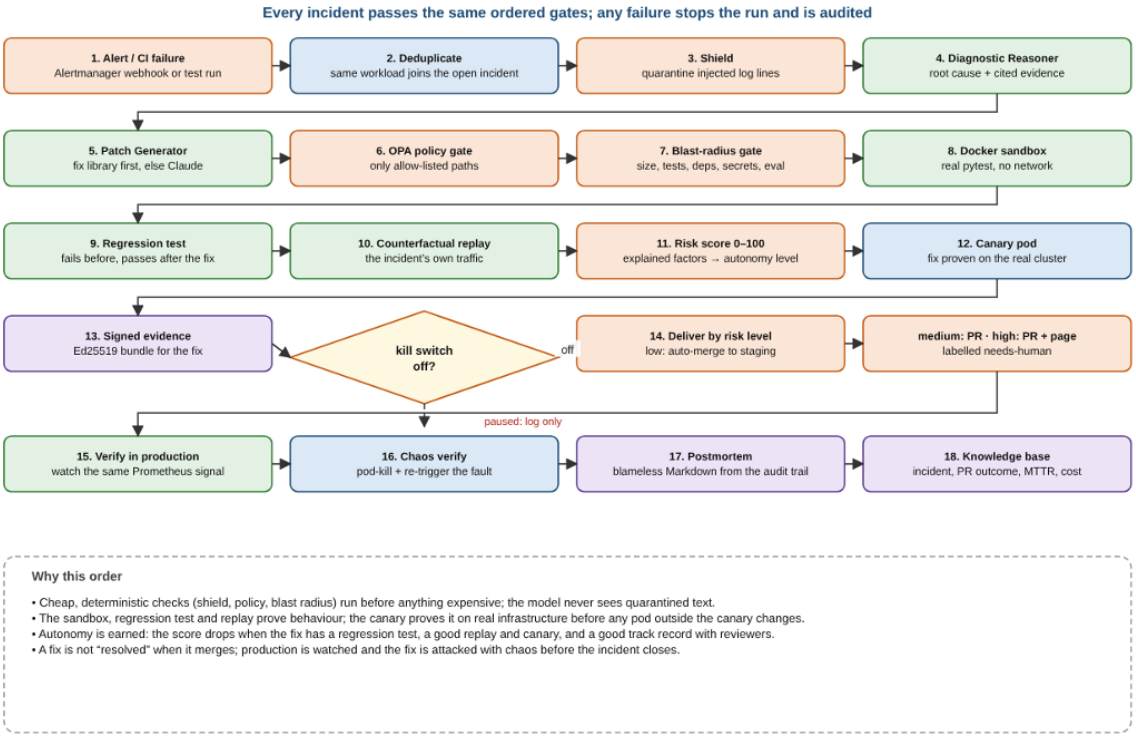


.Fig. 1 System architecture for GhostOps

The backend hashes the alert name and workload. If there is an open incident with the same alert it will be duplicate. If there is a different alert type for the same workload within 10 minutes, it will be added to the incident. If there is no activity for 2 hours it will be closed as it is stale. Only new incidents will trigger the pipeline, as there is a lock to ensure one incident happens at a time as git branches are switched by the deployer. Sixteen deterministic detectors are used to analyse logs, metrics and deployment information. Each detector returns typed findings with severity, confidence and evidence. There are 17 cases in the system (A1-A5 logs, B1-B6 metrics, C1-C6 deployments). As C4 is simply a failed test run, no detectors are needed.

The pipeline is a sequential process using crewai [18]. The Diagnostic Reasoner identifies the root cause, this can be memory_leak, ci_test_failure, null_reference or unknown. The reasoner will also quote the logs and metrics it has used. The Patch Generator will then return a minimal unified diff to address the issue; it may not touch the tests. The Validation Officer does not use the model, instead, it performs deterministic checks and determines the outcome based on the exit code from the sandbox. The first two agents are powered by Claude Sonnet 5.5 [20] via LiteLLM. Sometimes the model will generate correct changes but with incorrect line counts in a hunk header, making them invalid as git apply and GNU patch will not accept them. The pipeline will therefore recalculate this. If the system is run in offline mode, by setting GHOSTOPS_MOCK_LLM=1, then only the model will be mocked. Diagnosis will be done via the detectors,

and the patches will be generated from a template. Policy checks, sandbox, scoring, audit and signing are performed normally. If the system is run in a mock run, no pull request will be opened unless a second flag is set. The delivery stage will be reported as blocked. The stages are, shield, diagnosis, patch, validation, regression testing, replay, risk scoring, canary, evidence, delivery, post-mortem and knowledge base [Fig. 2].



.Fig. 2 GhostOps remediation pipeline with gates

Each patch will go through up to 6 checks. None of these will ask for an opinion from the model. The shield will protect from prompt injections. Every log line will be checked for override instructions, role takeover, fake conversations, mentioning the agent, tampered verdict, secrets extraction and code execution. Lines will also be checked for hidden content such as zero width characters, bidi overrides, Ansi escapes and large encoded blobs. Any matching lines will be replaced with a marker, and any hidden characters will be removed from the remaining lines. The source files will be scanned but will not be modified as the patch must apply exactly to the files. The shield will determine a rating (none, low, elevated or high) based on the checks performed and this will feed into the risk score. The output will then go through two further checks. First, an Open Policy Agent [12] Rego policy will be applied using conftest. The patch can only touch files under sample-app/ and the manifest infra/k8s/sample-app.yaml. This will block any path traversal, similar prefixes such as sample-app-evil/ and empty paths. The next check is the blast radius gate. This will determine what the diff is doing. Patches that alter more than 40 lines or more than one file will be blocked. Patches that remove any assertions, tests or add skip/xfail annotations will be blocked. Any changes to dependencies manifests or importing new third party packages will be blocked. Patches touching authentication, secrets or credentials will be blocked. Patches adding eval, exec, os.system, subprocess, pickle, unsafe yaml.load or network calls will be blocked. Patches that elevate privileges in the manifest will be blocked. No model or network is used.

The 4 th check is the sandbox. A disposable container without network access, 256MB memory and limited capabilities will be used to apply the diff and run the projects tests. The outcome will be determined by the exit code. The regression test created will prove the patch by ensuring it fails when unpatched and passes when patched. If this is not the case, then it will be ignored. The 5 th check is counterfactual replay. The HTTP requests from the access log in the pod will be multiplied by 10 and run in the sandbox. The incident must occur with the unpatched code (errors, exceptions or increased memory usage over 256KiB per request) and not occur with the patched code. The 6 th check is the canary. This is a pod with the label track=canary and has the same service selector as the stable pods [13]. This pod will receive the traffic from

the incident. The pod will be assessed using Prometheus metrics, looking at memory increase per request and error rate compared to the stable release. It will only be promoted if it is clearly better, low risk and if the kill switch allows. If the release is bad (C1) then no patch will be applied, but it will be rolled back and verified using real requests. The latter stages will monitor how long it takes to recover and repair. Optionally, the fix can be attacked using Chaos Mesh [19]. However, these are implemented but not performed against a cluster.

The risk score determines how autonomous the fix will be. Factors will be added or subtracted, capped between 0-100.

$$R = \min(100, \max(0, \Sigma f_k)) \quad (1)$$

Where f_k is the positive or negative contribution of factor k. The diff size factor will increase depending on how many lines are changed (c).

$$f_{size} = \min(25, \text{round}(25c / 40)) \quad (2)$$

If there was a proven regression test, canary passed, and incident was reproduced and fixed during replay, -10 will be subtracted from the score. If the model had low confidence, little evidence in tests, findings from the shield, and incident occurs again then the score will increase. If the reviewer has a good track record, then the score will decrease. If the score is less than 30 then it will be classed as low risk. The patch will automatically be merged to the staging branch. If the score is between 30-59 then it will be classed as medium risk. A pull request will be opened. If the score is 60 or higher, it will be classed as high risk. A pull request will be opened and labelled as human needs. A page will also be sent. Before any automated actions are taken (pull request, auto merge, rollback, canary promotion), the kill switch will be checked. If the kill switch is paused, the action will not be taken, and the rejection will be logged. Figure 3 shows a summary.

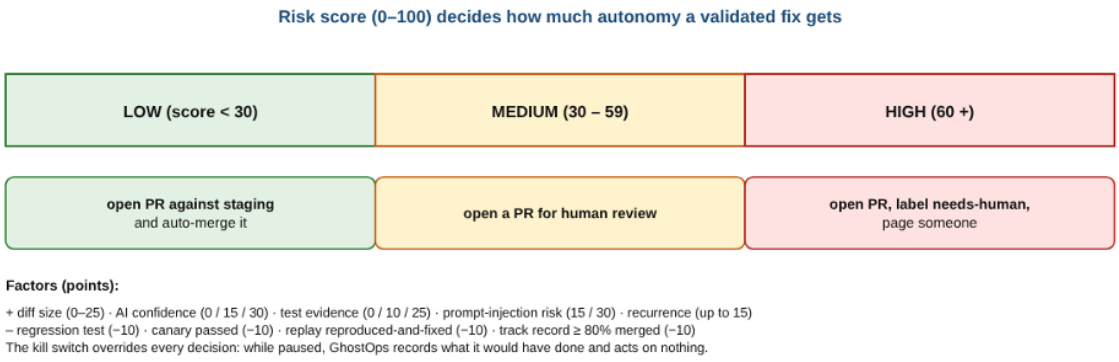


Fig. 3 Risk score and autonomy.

There are three accountability mechanisms. The first is an audit log. Each decision will be logged, and a hash of the previous entry will be stored. The secure audit logs from Schneier and Kelsey [15] are used.

$$h_i = \text{SHA-256}(h_{(i-1)} \| J(e_i)) \text{ where } h_0 = 64 \text{ zeros} \quad (3)$$

Where e_i is the ith entry, J(·) is the canonical JSON representation with sorted keys and compact separators. Altering or deleting any rows will change all subsequent hashes. The first broken row can be identified using verify_chain(). The second mechanism is evidence. This is a bundle containing the incident, SHA-256 hash of the diff, sandbox output, all gate decisions, regression, replay and canary decisions, risk score, sandbox image and commit. This is signed using Ed25519 [16]. The reviewer can calculate the hash of the diff in the pull request and verify the signature. This proves the bundle is genuine and the diff is the one that was validated. The third accountability mechanism is a blameless postmortem [1]. This is written for every run, either passed or failed, using the audit trail. There is a feedback loop to get the outcome of the pull request from GitHub. If the fix has been merged by humans twice then it will become a skill.

Each mechanism was provided input to ensure it would be rejected. The shield was given log lines with injected instructions and one line with zero width characters. The blast radius gate was given a patch that

added skip, removed an assertion, imported requests, called subprocess and read a password. The policy was given paths backend/main.py, sample-app/../backend/main.py, sample-app-evil/app.py. It was also given two paths that should pass. The patch was altered and checked against a signed bundle. One altered row in the audit. Scenario C4 was ran with the kill switch paused. The suites in the repository were also run. 350 pytest tests for the agent's package. 180 for the backend (cluster, Prometheus and GitHub were mocked). Rego unit tests, Vitest dashboard tests, TypeScript type check, and production build. The browser end to end test was not performed.

Seven metrics were recorded. Whether the fix was correct by checking if the project tests pass when the code is patched in the sandbox. Whether the proof was strong by checking if the regression test failed pre-fix and passed post-fix. How big the patch was by counting the number of lines and files changed. Risk and autonomy were determined by the score, level and delivery decision. Whether each gate was rejected and accepted as expected. The latency was determined by the clock time taken from the start of the scenario until a validated and audited fix. The container would take time to start up on a shared machine. The 30s alert delay, groupings from Alertmanager and model delay were not included. This provides information about how long the pipeline took and not a real incident. Accountability was determined by how many audit entries there were, the hash chain and signatures. None of these metrics determine how accurate the diagnosis was from the model.

III. RESULTS AND DISCUSSION

The below table gives the outcome of the three code-fix scenarios. In each run the diagnosis matched the planted bug with high confidence, although in the offline configuration that label comes from the detectors and says nothing about the model. Each patch was a one-file change of two to four lines: C4 turned a + b into a * b, A1 added a guard for an empty payload, and B1 replaced an unbounded list with a deque of maxlen 2. The sandbox passed the project's tests in all three runs, with 2, 3 and 7 tests. The generated regression test failed before the fix and passed after it for C4 and A1. For B1 the proof could not be verified, which we discuss with the test results. Counterfactual replay was not applicable in any run: C4 has no traffic, A1 is not a web application, and B1 recorded four requests from an application that cannot be replayed. The risk scores were 6, 6 and 7, all low, and every run reached the decision to auto-merge to staging. Delivery itself was held, because a mock run opens no pull request.

Results of the three code-fix scenarios.

Measure	**C4: CI test failure**	**A1: unhandled exception**	**B1: memory leak**
Root cause	ci_test_failure (high)	null_reference (high)	memory_leak (high)
File patched	sample-app/math_utils.py	sample-app/user_service.py	sample-app/app.py
Change	a + b changed to a * b	guard for empty payload (2 lines added)	bounded deque(maxlen=2) (3 added, 1 removed)
Changed lines	2 (1 added, 1 removed)	2 added	4
Sandbox verdict	passed, 2 tests	passed, 3 tests	passed, 7 tests
Regression test	failed before, passed after	failed before, passed after	not verified
Counterfactual replay	not applicable (no traffic)	not applicable (not a web app)	not applicable (4 requests, app not replayable)
Risk score / level	6 / low	6 / low	7 / low
Delivery decision	auto-merge to staging	auto-merge to staging	auto-merge to staging
Actual delivery	held: mock mode	held: mock mode	held: mock mode
Wall time	9.0 s	2.6 s	5.9 s

Figure 4 shows the wall-clock time from the start of the scenario to a validated and audited fix. A1 required 2.6 seconds since its sandbox carries out only three tests. B1 took 5.9 seconds because its sandbox includes seven tests, such as the 40-request leak checks. C4, which was the first run of the session, took 9.0 seconds. These times take into account the container start-up on the shared machine but do not include the 30-second alert hold, Alertmanager grouping, and model latency, so they reflect the pipeline time rather than the time taken to repair the issue. After updating the repository, we repeated the scenarios. The verdicts, the regression proofs and the risk scores were the same as before, but the times were different from the original measurements (8.0 seconds, 2.7 seconds and 6.5 seconds). Since we have two measurements for each scenario, we cannot make any statement regarding their spread.

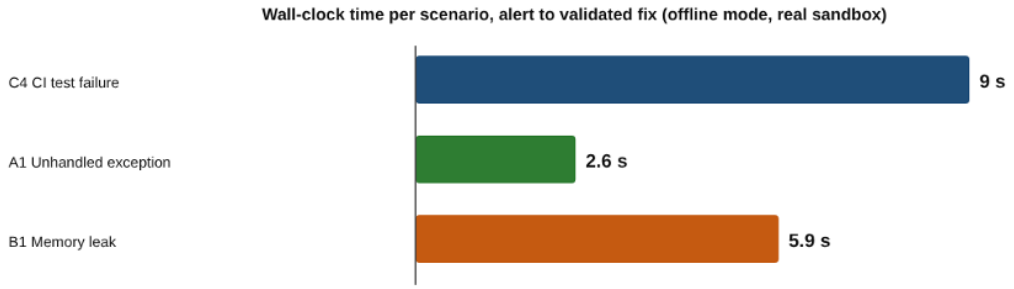


Fig. 4 Time from scenario start to validated fix

With respect to C4, the failing test_multiply was found to be due to the multiply() function returning a + b. For A1, the traceback resulting from a call to display_name() when there was an empty upstream payload was deemed a null reference, and the guard returns "anonymous" since this is what the function's own docstring states. In the B1 run the Reasoner referred to memory samples having increased from 27,000,000 to 118,000,000 bytes, that is 88 % of the 134,217,728-byte limit, together with the access-log entries for the /leak endpoint. Figure 5 illustrates the completed run in the dashboard.



Fig. 5 The completed memory-leak (B1) run as shown in the dashboard

Outside the three scenarios, the detector lab used deterministic detectors with a canary-regression preset and stated that C1 was critical with a confidence level of 98 per cent, the canary showing an error rate of 15.0 per cent as compared to 0.1 per cent for the stable release. Following the dashboard and API runs, the knowledge base contained eight incidents, all of which were validated and none of which were blocked. Each run produced about nine audit entries: diagnose, patch, regression_test, replay, validate, risk, evidence, open_pr and postmortem.

Out of the 530 pytest tests (350 belonging to the agents' package and 180 to the backend), 529 passed while one failed, with the tests taking 108 seconds to run against the actual Docker sandbox. All 10 Rego tests passed, and all 26 Vitest tests also passed, this taking 2.3 seconds. No errors were found during the TypeScript type-check and the Next.js production build was successful. The browser end-to-end test was not carried out in this environment since it usually runs in continuous integration using Chrome. Figure 6 displays the numbers by suite.

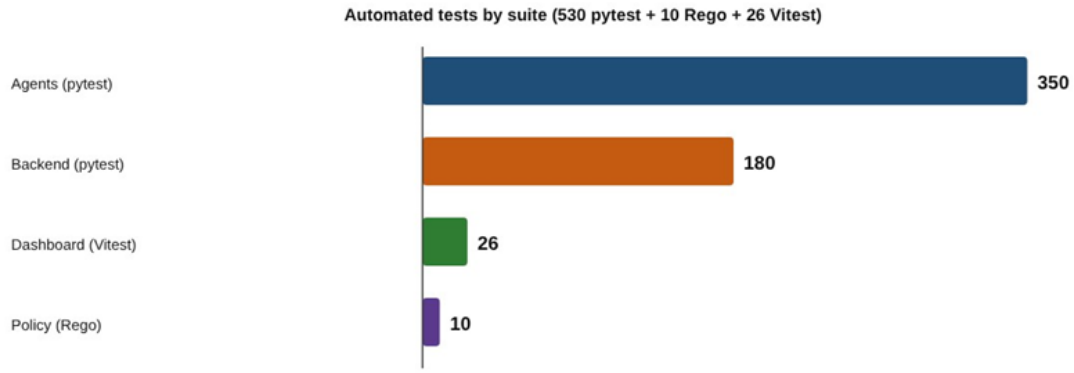


Fig. 6 Automated tests by suite

The only failure concerns the B1 end-to-end test, which claims that the regression test should fail when running on the unpatched version of the code. In our environment the unpatched code allocates 40 times 10 MiB within a sandbox that has a limit of 256 MB; the container is terminated by the OOM killer with an exit status of 137 and the harness does not regard this as a test failure, so the proof is shown as not having been verified even though the patch has passed all seven of the existing tests. We confirmed this directly by using a container restricted to 256 MB that allocated 400 MiB and existed with status 137. The reason for this is the method in which the host enforces the memory limit, not the fix, and since C4 and A1 do not rely on the memory cap, they were able to verify their regression tests as normal. As a result, we could not obtain a verified B1 proof in this environment.

We also provided each safety mechanism with an input designed to cause it to reject the request. Three lines involving injections and one line containing zero-width characters raised the shield's risk level to high, three of the lines being quarantined and the fourth being cleaned before it was allowed to pass. The blast-radius gate rejected a patch that included a skip, a removed assertion, an import of requests, a call to a subprocess, and a reading of a password, and it recorded six violations. It accepted the fix for multiply () which involves altering one file by adding one line and removing one line. Figure 7 shows both demonstrations.

```
$ shield.sanitize(failure_context)
risk = high | quarantined = 3
  model sees: 10.244.0.1 - - "GET /leak HTTP/1.1" 200 -
  model sees: [QUARANTINED by GhostOps shield: instruction_override, verdict_tampering]
  model sees: [QUARANTINED by GhostOps shield: role_hijack, secret_exfiltration]
  model sees: [QUARANTINED by GhostOps shield: code_execution]
  model sees: INFO ok normal line with hidden zero-width characters
  finding: logs[1] ['instruction_override', 'verdict_tampering'] -> quarantined
  finding: logs[2] ['role_hijack', 'secret_exfiltration'] -> quarantined
  finding: logs[3] ['code_execution'] -> quarantined
  finding: logs[4] ['zero_width'] -> cleaned

$ blast_radius.analyze(<patch that weakens a test and adds a network call>)
passed = False {'files': 1, 'added': 5, 'removed': 1}
  VIOLATION weakens_tests - sample-app/test_app.py: removes assertions or tests
  VIOLATION weakens_tests - sample-app/test_app.py: skips or neutralises tests
  VIOLATION new_dependency - new third-party import: requests
  VIOLATION network_call - import requests
  VIOLATION dangerous_call - subprocess.run(["curl", "http://evil.example"])
  VIOLATION sensitive_code - touches secrets/credentials: password = os.environ["DB_PASSWORD"]

$ blast_radius.analyze(<the validated multiply() fix>)
passed = True {'files': 1, 'added': 1, 'removed': 1}
```

Fig. 7 prompt-injection shield and blast-radius gate

The policy rejected backend/main.py, the path traversal path sample-app/../backend/main.py and the similar path sample-app-evil/app.py, while permitting sample-app/app.py and infra/k8s/sample-app.yaml. The audit log, which consisted of 74 entries verified as intact, showed the breach only when we altered one of the rows, at which point verify_chain() indicated the failure at row 20—the row that had been modified. Figure 8 displays both the results.

```
$ conftest test --policy allowlist.rego --namespace allowlist  (one input per patch target)
  sample-app/app.py  =>  1 test, 1 passed, 0 warnings, 0 failures, 0 exceptions
  backend/main.py  =>  FAIL - /in.json - allowlist - Patch targets unauthorized file path: backend/main.py 1 test, 0 passed,
0 warnings, 1 failure, 0 exceptions
  sample-app/../backend/main.py  =>  FAIL - /in.json - allowlist - Patch targets unauthorized file path: sample-app/../backen
d/main.py 1 test, 0 passed, 0 warnings, 1 failure, 0 exception
  sample-app-evil/app.py  =>  FAIL - /in.json - allowlist - Patch targets unauthorized file path: sample-app-evil/app.py 1 te
st, 0 passed, 0 warnings, 1 failure, 0 exceptions
  infra/k8s/sample-app.yaml  =>  1 test, 1 passed, 0 warnings, 0 failures, 0 exceptions
$ governance.verify_chain()                    # untouched database
{'valid': True, 'entries': 74, 'broken_at': None}

$ UPDATE audit_log SET decision='passed' WHERE id=20   # someone edits history
$ governance.verify_chain()
{'valid': False, 'entries': 74, 'broken_at': 20}
```

Fig. 8 OPA policy and audit-chain tamper detection

The evidence bundles, which had been verified based on the diff it was constructed from, still produced a valid signature when checked against an edited diff since the bundle itself had not been altered; however, the digest of the diff no longer matched. To be certain that the diff in a pull request is the one that has been validated, a reviewer must carry out both checks (see Fig. 9).

```
$ python -m agents.evidence verify bundle_c4.json --diff patch_c4.diff
incident   ci_test_failure (ci_failure)
patch      sample-app/math_utils.py  sha256 856f01f17f36c931…
gates      sandbox=pass  blast-radius=True
regression failed before=True  passed after=True
signature  VALID  (key c1770546511b2e36)
patch      matches the validated diff

$ python -m agents.evidence verify bundle_c4.json --diff patch_tampered.diff
incident   ci_test_failure (ci_failure)
patch      sample-app/math_utils.py  sha256 856f01f17f36c931…
gates      sandbox=pass  blast-radius=True
regression failed before=True  passed after=True
signature  VALID  (key c1770546511b2e36)
patch      DOES NOT MATCH the validated diff
```

Fig. 9 Checking a signed evidence bundle

Even though the kill switch was paused, scenario C4 still carried out the diagnosis of the incident, verified the fix and signed the bundle, and prepared its postmortem; the only action that was blocked was open_pr, with the reason given as autonomy_paused, and the refusal was recorded in the audit log (Fig. 10).

```
$ model diff with wrong hunk-header counts:
@@ -6,9 +6,3 @@
$ recount_hunks(diff):
@@ -6,4 +6,4 @@
```

Fig. 10 Kill switch – an action that has been blocked and audited

Putting the results together, it shows that the loop from trigger to a validated, risk-scored, signed and audited fix was completed without any human intervention in a wall-clock time of 2.6 to 9.0 s when the system was not running, and that each mechanism rejected the input it was designed to reject. The gate cases are manually written, with only a few per mechanism, so they demonstrate the correct refusals and acceptances for those cases and do not give any false rejection rate for inputs we did not anticipate. The B1 run illustrates what happens to the system when proof is missing. Since the regression test could not be shown to fail on the unpatched code, the dashboard treated it as rejected and the score of 7 includes no credit for the proof. The purpose of the regression test is to detect patches which pass the current tests without eliminating the bug [7]. However, according to Equation (1), the absence of proof only resulted in a loss of credit. The fix remained in the low band and was given the auto-merge decision, and it was only the mock-mode hold that prevented its delivery. Whether the absence of proof should block auto-merge completely is a design issue that these results raise but do not resolve. Replay was not applicable in any of the runs, so we have not been

able to show it reproducing and then fixing an incident end to end. Only its unit tests cover that path.

Several threats mean that the results do not provide full support. The model call has been replaced by a mock which obtains its diagnoses from the same detectors that are used in the pipeline and whose patches are taken from templates for the known bugs, so the evaluation is testing the pipeline and its gates, not Claude's diagnostic accuracy. In the absence of a cluster, the canary, chaos, post-deployment verification, cost-meter and live-mode stages were tested only by 60 stubbed backend tests, and we therefore give no actual repair time. The three scenarios are carried out using a bundled sample application, each containing one planted bug, and the fixes involve changes of one to four lines, so what happens on larger services is unknown. All the measurements are taken on a single host, with at most two-timing values per scenario. The B1 proof relies on how that host enforces the memory cap. The live API is unauthenticated, and the public site provides a secret-free mock-mode version. Vault operates in development mode, and the allow-list is manually edited. The pipeline is heavily dependent on tokens and can reach API rate limits, and a lock ensures that only one pipeline runs at a time. Thirteen of the seventeen use cases are detected and diagnosed but not remediated.

We carried out a comparison between GhostOps and four other categories of existing methods: alert-only monitoring, runbook automation, LLM coding assistants, and typical LLM site-reliability agents. The last two groups represent general trends that are found in the literature and in current tooling, rather than referring to specific products. The comparison is qualitative and none of the aspects were measured during our evaluation. Alert-only monitoring can detect incidents and send out notifications, but it neither diagnoses them nor carries out any fixes, and it does not keep a tamper-evident record; as a result, verifying that an incident has been recovered has to be done manually. Runbook automation too detects incidents and carries out pre-scripted actions, although its diagnosis is based on fixed rules, it cannot modify application code, and it provides no risk-graded autonomy. The best it can do is to keep normal logs and it does not check whether recovery has taken place. An LLM coding assistant does not detect incidents from live metrics; it only diagnoses when asked and suggests a patch, with no defence against injection, no independent gate on the patch, no risk-graded autonomy, no audit record and no recovery check. A typical LLM site-reliability agent is capable of detecting incidents, diagnosing them and proposing a patch, but it rarely protects against prompt injection, usually relies solely on the project's tests to gate the patch, and rarely provides risk-graded autonomy, a tamper-evident audit, signed proof or recovery verification. GhostOps includes all of these features when it comes to the patch. Some of the capabilities mentioned were not put into practice during our evaluation: detection from live metrics, the canary gate and recovery verification were only addressed through detector-lab presets and stubbed tests, and the assertion that code fixes are validated applies only to the sample application that came with the bundle.

IV. CONCLUSION

We developed GhostOps, a proof-of-concept platform designed for incident remediation that regards the language model as an untrusted participant, and we carried out an offline evaluation by substituting the model call with a mock one. The main conclusions can be summarised as follows:

1. A Prometheus alert can go through the entire process of diagnosis, applying a minimal patch, validation, risk scoring, signing and auditing without the need for any human intervention. In the case of three scenarios — a memory leak, a failing CI test and an unhandled exception — each was resolved by making changes to a single file consisting of two to four lines, with validation carried out by 2, 3 and 7 sandbox tests respectively, and the entire process taking between 2.6 and 9.0 seconds of offline wall-clock time.
2. The three fixes obtained scores of 6, 6 and 7 and were included in the low-risk category; the C4 and A1 regression tests had failed before the fix and had passed afterwards, while the B1 proof was not verified.
3. Out of the 530 pytest tests, 529 passed; the only one that failed was the B1 end-to-end test, since the unpatched leak was killed by the OOM condition when the 256 MB limit was reached, and the harness does not regard this as a test failure, so the regression proof is not considered verified. All 10 Rego tests and 26 Vitest tests passed.
4. The results indicate that the pipeline and its gates function as intended when used in a bundled sample application; however, they do not provide a measure of Claude's diagnostic accuracy, nor do they give information about actual repair times or the behaviour of the canary, chaos, and post-deployment phases on a live cluster, and in thirteen of the seventeen use cases detection takes place but remediation does not.

REFRENCES

[1]. Beyer, B., Jones, C., Petoff, J. and Murphy, N. R. (Eds.) [2016] Site Reliability Engineering: How Google Runs Production Systems.

[2]. Kephart, J. O. and Chess, D. M. [2003] "The Vision of Autonomic Computing" IEEE Computer, Vol. 36, No. 1: pp.41–50.

[3]. Dang, Y., Lin, Q. and Huang, P. [2019] "AIOps: Real-World Challenges and Research Innovations" In Proc. IEEE/ACM 41st International Conference on Software Engineering: Companion (ICSE-Companion).
[4]. Notaro, P., Cardoso, J. and Gerndt, M. [2021] "A Survey of AIOps Methods for Failure Management" ACM Transactions on Intelligent Systems and Technology.
[5]. Ahmed, T., Ghosh, S., Bansal, C., Zimmermann, T., Zhang, X. and Rajmohan, S. [2023] "Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models" In Proc. ICSE.
[6]. Chen, Y., et al. [2024] "Automatic Root Cause Analysis via Large Language Models for Cloud Incidents" In Proc. EuroSys.
[7]. Le Goues, C., Pradel, M. and Roychoudhury, A. [2019] "Automated Program Repair" Communications of the ACM, Vol. 62, No. 12: pp.56–65.
[8]. Jimenez, C. E., et al. [2024] "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In Proc. ICLR.
[9]. Yang, J., et al. [2024] "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering" In Advances in Neural Information Processing Systems (NeurIPS).
[10]. Perez, F. and Ribeiro, I. [2022] "Ignore Previous Prompt: Attack Techniques For Language Models".
[11]. Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T. and Fritz, M. [2023] "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection" In Proc. ACM Workshop on Artificial Intelligence and Security (AISec).
[12]. Open Policy Agent Project. "Open Policy Agent Documentation and the Rego Policy Language"
[13]. Humble, J. and Farley, D. [2010] Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley.
[14]. Basiri, A., et al. [2016] "Chaos Engineering" IEEE Software, Vol. 33, No. 3: pp.35–41.
[15]. Schneier, B. and Kelsey, J. [1999] "Secure Audit Logs to Support Computer Forensics" ACM Transactions on Information and System Security, Vol. 2, No. 2: pp.159–176.
[16]. Josefsson, S. and Liusvaara, I. [2017] "Edwards-Curve Digital Signature Algorithm (EdDSA)" RFC 8032, IETF.
[17]. Prometheus Authors. "Prometheus: Monitoring System and Time Series Database".
[18]. CrewAI. "CrewAI Documentation: Multi-Agent Orchestration"
[19]. Chaos Mesh Authors. "Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes"
[20]. Anthropic. "Claude API Documentation"