Submitted to
International Conference on Artificial Intelligence & Future Automations (ICAIFA 2025)
ESP Journal of Engineering & Technology Advancements (ESP-JETA)
ISSN: 2583-2646 / Volume 5 Issue 4 November 2025
Paper Id: JETA-V5I4P128/ Page No: 211--221

Original Article

Autonomous SRE: SLO-Aware Multi-Agent Root-Cause Analysis and Verified Self-Healing for Cloud-Native Systems

Rajesh Balaji
Independent Researcher, Seattle, WA, USA
Anna University, Chennai, India

Received Date: 20 December 2025 Revised Date: 25 December 2025 Accepted Date: 30 December 2025

Abstract: *Automated incident response in microservice systems fails in a predictable way. It blames the component that looks worst, which is often a victim, and then acts on it with confidence it has not earned. We present Autonomous SRE, a design that makes the service-level objective the judge at every step. Multi-window error-budget burn rates decide when to act. Four deterministic evidence agents (metrics, capacity, change and topology) and a coordinator separate culprits from victims and rank candidate remedies by likelihood and blast radius. Each remedy is then verified against the burn rate and reverted if it does not help, and a person is paged only when the hypotheses run out. We evaluate the design on a fault-injection simulator of a nine-component application, with eight fault types used during development, three held out until the design was frozen, and benign deployments and blips. On 240 test incidents of known types it spent 2.4% of a 28-day error budget per incident, against 12.4% for threshold runbooks, 12.6% for graph-based root-cause analysis with automatic remediation and 15.2% for manual response. It caused 0.05 harmful actions per incident, against 0.53 and 0.17. On never-seen faults, verification raised the share fixed without a person from 3% to 37%. Burn-rate triggering reacted more slowly than thresholds to faults that burn slowly. The simulator, source code, generated episodes, and archived results are supplied as a reproducibility package.*

Keywords: *Site Reliability Engineering, Service-Level Objectives, Error Budgets, Root-Cause Analysis, Self-Healing Systems, Microservices, Aiops*

I. INTRODUCTION

Most production incidents in a microservice system are diagnosed the same way. Someone is paged because a dashboard turned red. They open five more dashboards, notice that the service which paged them is slow because something it calls is slow, follow the chain down, find a deployment that went out twenty minutes earlier, roll it back, and watch the graphs until they are sure it worked. The fix is usually simple. The time goes into finding where to apply it, and into being confident enough to apply it. Both halves have been automated many times, and both automations have a well-known failure mode. Root-cause analysis tools rank components by how anomalous they look and how they sit in the dependency graph. But the component that looks worst is often a victim. A database overloaded because the cache in front of it failed is the classic case. Self-healing tools restart pods, roll back releases and scale out. But an automated action applied to the wrong component does not merely fail to help: a needless database failover or a restart of a healthy service turns one incident into two. Site reliability engineers are rightly wary of automation that acts with confidence it has not earned.

This paper describes Autonomous SRE, a design that tries to earn that confidence. Two ideas carry most of the weight. The first is to make the service-level objective (SLO), not a set of thresholds, the measure of everything. Detection uses error-budget burn rates [2], so the system wakes up when users are actually being hurt. The same burn rate decides how aggressive a remedy may be, and whether a remedy worked. The second is to never trust a remedy until it has been verified. Each action is applied, its effect on the burn rate is checked over a short verification window, and it is automatically reverted if the SLO does not recover. The next hypothesis is then tried, and a person is called only when the hypotheses run out. Root-cause analysis is done by several specialist agents rather than one model. A metrics agent scores how anomalous each component is. A topology agent ranks candidates by walking the dependency graph. A change agent links the onset of trouble to recent deployments and configuration changes. A capacity agent asks whether the problem is simply load. A coordinator fuses their evidence and is explicitly designed to tell culprits from victims. The agents here are deterministic software components, not language models, which makes their behaviour reproducible and their failures inspectable.

The contributions are as follows.

This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/2.0/)

- An SLO-aware detection and gating scheme that uses multi-window burn rates both to trigger investigation and to decide which remedies are allowed.
- A multi-agent root-cause analysis design that combines anomaly, topology, change and capacity evidence, with explicit handling of upstream-victim and cascading-dependency cases.
- A verified self-healing loop that applies remedies in order of hypothesis confidence and checks each against the SLO before keeping it. A remedy that fails verification is reverted automatically.
- A reproducible evaluation on a fault-injection simulator of a nine-component cloud-native application. The simulator covers eight fault types, including deliberate traps, plus benign deployments and transient blips, and three more fault types held out until the design was frozen. The evaluation compares threshold-based runbook automation, a single graph-based root-cause model with automatic remediation, and the full design. Ablations remove SLO awareness, verification, change awareness, agent coordination and the approval gate one at a time.

The experiment runs on a simulator rather than a live cluster, and we state this plainly throughout. The simulator, fault definitions, and parameters are supplied in the accompanying reproduction package, and Section VIII describes how to repeat the evaluation with chaos experiments on a real system.

Section II reviews related work and Section III sets out the system and fault model. Sections IV and V describe the design. Sections VI and VII cover the experiment and results, and Section VIII discusses limitations and concludes.

II. BACKGROUND AND RELATED WORK

A. SLOs and error budgets

Google's SRE books [1], [2] made the service-level objective the centre of operations. A target such as "99.9% of requests succeed within 500 ms over 30 days" defines an error budget: the 0.1% that may fail. The burn rate is how fast that budget is being spent relative to the rate that would exhaust it exactly at the end of the window. Multi-window, multi-burn-rate alerts [2] page only when a fast burn over a short window is confirmed by a longer window. This catches serious problems within minutes while ignoring brief blips that cost little budget. We take this mechanism further than alerting, and use the burn rate to gate remediation and to judge whether a remedy worked.

B. Root-cause analysis in microservices

Soldani and Brogi [18] survey anomaly detection and root-cause analysis for service-based applications. Graph-based methods dominate. MicroRCA [4] builds an attributed graph of services and hosts, weights edges by anomaly correlation, and ranks candidates with personalised PageRank [17]. AutoMAP [5] and CauseInfer [6] build causal graphs from metrics. Sage [7] learns a graphical model of performance dependencies. Ikram et al. [8] apply causal discovery to localise failures. These methods share a weakness in practice: the most anomalous component is often a victim of something upstream or downstream, and a single score struggles to separate the two. Real topologies make this harder. Meta's analysis of its microservice architecture [19] shows call graphs that are large, dynamic and far from tree-shaped.

C. Change as a cause

A large share of outages follow a change to the running system [1], and studies of production incidents [13] point the same way. Systems such as Gandalf [12] and SCWarn [20] detect bad changes by correlating deployments with anomalies across many machines. Progressive-delivery controllers such as Argo Rollouts and Flagger [16] promote or abort a release based on metric analysis. In other words, they verify a change before trusting it. We borrow that idea and apply it to remediations.

D. Self-healing

The MAPE-K loop of autonomic computing [3] (monitor, analyse, plan, execute, over shared knowledge) is the reference model for self-healing systems. Kubernetes implements its simplest form with liveness probes and controllers that restart failed containers and keep replica counts [15]. Chaos engineering [11] made deliberate fault injection a routine way to test whether such mechanisms work.

E. Language models for incidents

Ahmed et al. [9] showed that large language models can recommend root causes and mitigations for cloud incidents at Microsoft, and RCACopilot [10] combined automated diagnostic collection with LLM-based root-cause prediction. These systems help engineers reason faster but do not by themselves act on the system, and they raise the same question this paper asks of any automated responder: what happens when it is confidently wrong?

F. Position of this work

Prior work has produced good root-cause rankers and good self-healing primitives. Within the works reviewed, we did not identify a design that ties them to the SLO end to end, using the error budget to decide when to act, how boldly to act and whether the action worked, and that treats every automated remedy as a hypothesis to be verified and reverted if wrong. The present evaluation explicitly counts harm from incorrect remediation, while recognizing that the coverage of the related-work search is not exhaustive.

III. PROBLEM SETTING

A. The Application

The system under study is AegisCommerce, a simulated online retail application constructed for this evaluation. A gateway fronts five services (identity, catalog, cart, orders and payments). They share a PostgreSQL database and a Redis cache, and payments calls an external payment provider. That makes nine components. Traffic arrives as four user journeys: browsing the catalogue (6,000 requests a minute), logging in (600), using the cart (1,200) and checking out (300). Each journey follows a fixed path through the components. Checkout, for example, passes through the gateway, orders, cart, payments, the database, the cache and the provider.

Each component behaves as a queue with a fixed capacity per replica. Its own latency grows as base/(1 − ρ) with utilisation ρ, and above ρ = 0.9 it starts to reject work. A component's observed latency and error rate include those of everything it waits on. This is what makes victims: when the database slows down, every service that calls it looks slow too. Tracing, however, reports each component's self-time and its own errors separately, as span-based tracing does in practice.

B. The service-level objective

A request is good if it succeeds and finishes within its journey's latency target (300 ms for browse and cart, 500 ms for login, 2 s for checkout). The SLO is 99.9% good requests. In healthy operation about 0.016% of requests are bad, a burn rate of roughly 0.16. We measure the cost of every incident as the share of a 28-day error budget it consumes at normal traffic. One percent of the budget is about 3,300 bad requests.

C. Faults

Eight fault types were used while the design was being built. Three more were written only after it was frozen, and appear only in the final test (Table I). Several are deliberate traps. In F7 the cache fails and the database, now taking every read, becomes the most anomalous component, although the database is not the cause. In F8 a traffic surge saturates the catalog. That looks like a failing service, but restarting it only makes things worse. F2 and the other non-change faults often coincide with a healthy deployment that happens to land a few minutes before the fault, sometimes on the faulty service itself. This tempts any tool that blames the latest change. The held-out faults test what the design does when it meets something new. In U1, a bad cart release makes orders throw the errors. U2 is an expired certificate that no automated action can fix. U3 is a slow database disk with no rise in CPU.

Table 1: Fault Catalogue

ID	Fault	Root component	Remedy that fixes it
F1	Bad deploy, raises errors	one of five services	roll back that service
F2	Capacity loss on a node	one of five services	scale out that service
F3	Memory leak shipped 15–40 min earlier	one of five services	roll back or restart it
F4	Database degradation	database	fail over the database
F5	Bad configuration, adds latency	one of five services	roll back the configuration
F6	Slow payment provider	provider	switch payments to fallback
F7	Cache failure, load cascades to the database	cache	restart the cache
F8	Traffic surge on browsing	catalog	scale out catalog
U1	Bad cart release, errors appear in orders	cart	roll back cart
U2	Expired certificate on identity	identity	human only (rotate certificate)
U3	Slow database storage	database	fail over the database

Every episode also contains benign events: two to four healthy deployments, one or two configuration changes, one to three short latency blips (two to six times normal latency for one to four minutes), and a slow daily swing in load of about ±12%.

D. Remedies and their blast radius

The automation can take five actions. A rollback reverts the latest release or configuration of a service, and does nothing if that release was healthy. A restart takes a component down for one minute. Scaling out adds a replica to a stateless component. Switching payments to its fallback path removes the provider from checkout. A database failover takes the database down for two minutes. We rank these by blast radius, from scale-out (0) up to failover (1.0). A wrong restart or failover is an outage that the automation caused, and we count each one as a harmful action.

IV. ARCHITECTURE

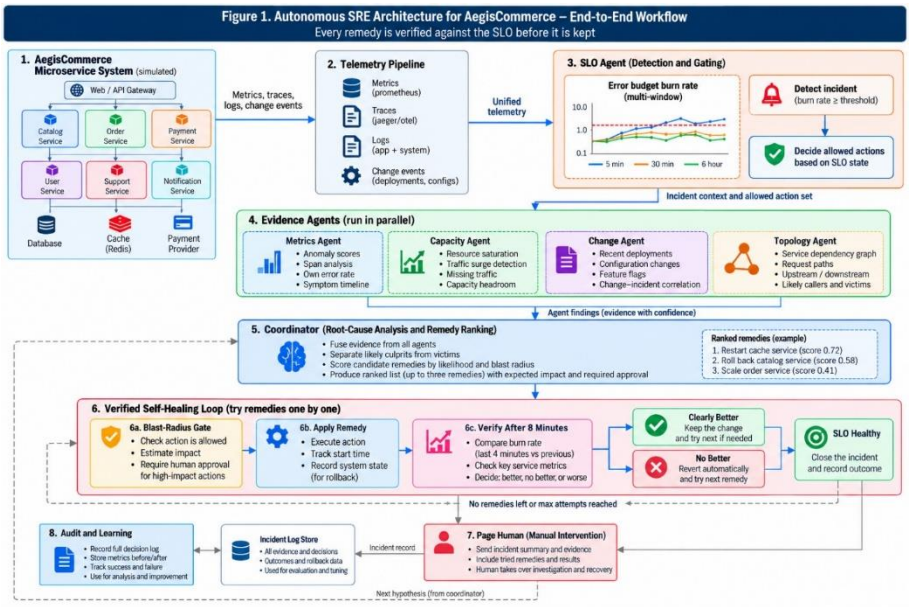


Figure 1: Proposed SLO-Aware Autonomous Incident-Response Architecture for Aegiscommerce

Four Deterministic Evidence Agents Analyze Metrics, Capacity, Recent Changes, And Service Dependencies. A Coordinator Ranks Candidate Remedies Using Evidence Scores and Estimated Blast Radius. Each Applied Remedy Is Evaluated Against the SLO After an Eight-Minute Verification Interval. Ineffective Reversible Actions Are Rolled Back, While Unresolved Incidents Are Escalated to An On-Call Engineer.

Fig. 1 shows the design. It follows the MAPE-K loop [3], with the SLO in charge of every stage. The SLO agent watches the burn rate and decides when to start. Four evidence agents each look at the incident from one angle. The coordinator turns their evidence into a short ranked list of remedies. The remediation loop then tries those remedies one at a time, and keeps a remedy only if the burn rate says it helped. A human is involved in two places only: approving an action with a large blast radius, and taking over when the list runs out.

The split into agents is mainly an engineering choice. Each agent is a few dozen lines with one job, can be tested on its own, and can be switched off to measure what it contributes (Section VII-D). None of them is a language model. An LLM could replace the coordinator or add a free-text agent that reads logs, but we wanted the first version to be fully deterministic, so that every decision in the evaluation can be traced and replayed.

V. AGENTS AND VERIFIED REMEDIATION

A. SLO agent

The burn rate over the last w minutes is B(w) = (share of bad requests in those minutes) / 0.001. A healthy system runs at about 0.16. Following the multi-window pattern [2], the agent raises an alert when B(15) ≥ 14.4 and B(2) ≥ 14.4 (a fast burn), or when B(60) ≥ 6 and B(5) ≥ 6 (a slower burn). The standard windows are one hour paired with five minutes, and six hours paired with thirty minutes. We shortened them because each simulated episode lasts four hours. After an alert the agent waits two minutes so the other agents have fresh evidence, and continues only if B(2) is still at least 2. It then reports which journeys

are burning: those whose bad share over five minutes exceeds both 0.3% and three times their own baseline. Only components on those journeys' paths are considered as suspects.

B. Evidence agents

Every agent compares the last two minutes with a baseline learned in a quiet hour at the start of each episode, as a z-score with a floor on the spread so that near-constant signals do not explode.

- Metrics agent. For each suspect c it uses self-time and the component's own errors, not end-to-end figures, so that a caller is not blamed for its callee: $m_c = \text{clip}((\max(z_{\text{self-latency}}, z_{\text{self-errors}}) - 3) / 7, 0, 1)$.
- Capacity agent. It reports saturation $s_c = \text{clip}((\text{CPU} - 0.80) / 0.15, 0, 1)$. It marks a component as busy when its CPU sits 0.25 above baseline, and reports a surge when gateway traffic exceeds 1.4 times baseline. It also flags any component whose traffic has vanished (below 30% of a baseline of at least 100 requests a minute). Vanished traffic is how a failed cache shows up: its clients stop calling it and go to the database instead.
- Change agent. It scores each suspect by its most recent deployment or configuration change in the last 90 minutes: $ch_c = \exp(-\Delta t / 45)$, where Δt is the age of the change in minutes.
- Topology agent. It supplies the journey paths and the call graph. It marks a component as a probable victim when its own traffic has risen by more than 40% while traffic to another component has vanished, which means load has shifted onto it.

C. Coordinator

The coordinator turns evidence into hypotheses, each a (remedy, target) pair with a score:

- Vanished traffic on c: restart c, score 0.95.
- A recent change on c: roll back c, score $0.4 + 0.6 \cdot ch_c \cdot \max(m_c, s_c)$. A change only scores highly if the changed component is itself misbehaving, which defuses decoy deployments.
- Saturation on c that is not a victim: scale out c, score $0.5 + 0.4\, s_c$, plus 0.1 during a surge. For the database the remedy is a failover, at $0.4 + 0.4\, s_c$.
- Self-anomaly on c with no recent change, not a victim and not busy: the component's own remedy. That is a fallback for the provider ($0.9\, m_c$), a cache restart ($0.8\, m_c$), a database failover ($0.6\, m_c$), or a service restart ($0.7\, m_c$). A component that is slow because it is busy is treated as a capacity problem, not restarted.

Each score is reduced by 0.15 times the remedy's blast radius, so when two remedies are about equally likely the gentler one goes first. The three highest distinct hypotheses are kept. The weights were set by hand from the reasoning above and checked only on development seeds (Section VI-D).

D. The verify-or-revert loop

Hypotheses are tried in order. A failover first needs a human to approve it. The approver is modelled as answering in 5 to 15 minutes, approving a correct failover 95% of the time and a wrong one 30% of the time. Each applied remedy is judged eight minutes later by comparing the burn rate over the last four minutes with its value when the remedy was applied:

- If $B(4) < 2$, the SLO is healthy again and the incident is closed.
- If $B(4)$ has at least halved, the remedy is kept and the next hypothesis is tried as well. This handles faults with two bottlenecks, such as a surge that saturates both the catalog and the gateway.
- Otherwise the remedy is reverted: an added replica is removed, the fallback is switched off, a rolled-back release is redeployed. Restarts and failovers cannot be undone, which is why they carry a higher blast radius. The next hypothesis is then tried.

When no hypothesis is left the on-call engineer is paged, with the list of what was tried and what each attempt did to the burn rate.

VI. EXPERIMENTAL SETUP

A. Simulator and Episodes

The simulator (Section III) advances in one-minute steps. Together with the designs and the evaluation it is about 1,000 lines of Python using NumPy and NetworkX. An episode lasts 240 minutes. Minutes 5 to 60 give each design its baseline, and the fault starts between minutes 90 and 130. Every design is run on exactly the same episodes, with the same traffic noise and the same benign events, so differences between designs come only from what the designs do.

B. Designs Compared

- Manual. The SLO alert pages a person, who fixes the fault correctly 20 to 45 minutes later. This is a deliberately kind model of a human: they never pick the wrong remedy.
- B0, threshold runbook. A component alerts when its latency exceeds 2.5 times baseline or its error rate exceeds the larger of 2% and five times baseline, for three minutes in a row. The runbook acts on the first component to alert. It fails over the database, restarts the cache, switches the provider to fallback and scales out the gateway. For a service it rolls back if there was a change in the last hour, scales out if CPU is above 0.85, and otherwise restarts. If the alert is still firing 15 minutes later, a person is paged. At most two automatic attempts are made per episode.
- B1, graph RCA with automatic remediation. This uses the same SLO alert as B2. Root cause is found with personalised PageRank in the style of MicroRCA [4], over end-to-end latency and error anomalies, with edges weighted by latency correlation over the last 15 minutes. The runbook is then applied to the top-ranked component, without verification. A person is paged if the burn rate is still above 2 after 15 minutes.
- B2, Autonomous SRE. The full design of Sections IV and V.

Five ablations each remove one part of B2. *No SLO* replaces burn-rate detection with B0's thresholds and drops the journey scoping. *No* verification applies the top hypothesis only, then waits 15 minutes and pages. *No change agent* switches the change agent off. *No coordination* merges each agent's own suggestions by raw score, without the victim and busy checks. *No gate* runs failovers without approval.

C. Measures

For each incident we record:

- RCA top-1: whether the first hypothesis would fix the fault.
- Time to detect: from impact onset, the first minute with more than 0.2% bad requests, to the alert.
- Time to recover: from onset to the moment the fault is removed. An incident never fixed within the episode counts as lasting to its end.
- Who resolved it: the automation or a person.
- Error budget burned: excess bad requests from onset to the end of the episode, as a percentage of a 28-day budget.
- Pages: escalations plus approval requests.
- Wrong and harmful actions: wrong actions are automatic remedies that did not fix the fault; harmful actions are wrong restarts and failovers, which cause outages.

In benign episodes we count false alerts and the actions taken per day.

D. Protocol and development history

All design work used development seeds 0 to 2. Each seed holds 8 fault types × 6 episodes plus 12 benign episodes, which is 180 episodes in total. The reported results come from test seeds 5 to 9: 240 incidents of the eight known types, 60 benign episodes (10 simulated *days*) and 90 incidents of the held-out types U1 to U3. The held-out faults were written after the design was frozen. Before the test run they were executed once on a development seed only to check that the code did not crash, and that output was discarded.

Four changes were made during development, and we report them because they shaped the results.

- The first B2 acted the moment the alert fired. With one minute of evidence it blamed the database in F7 and restarted a busy gateway in F8. We added the two-minute dwell and the busy check.
- The first verification step reverted any remedy that did not fully restore the SLO. In F8 it reverted the correct catalog scale-out because the gateway was still a second bottleneck, and it returned the system to a full outage. We added the keep-if-halved rule.
- In the simulator, latency blips were reduced from 3–10× to 2–6× so that short blips did not dominate every episode. Decoy changes on the faulty service itself were added after we noticed the scenarios had no such case.
- While first reading the test results we found a measurement bug: fault objects kept their state between designs run on the same episode. We fixed it and re-ran both the development and the test seeds, with no change to any design.

VII. RESULTS

All numbers in this section come from the test seeds and are produced by `evaluate.py` in the released package. Where a ± is given, it is the standard deviation of the per-seed means across the five test seeds.

A. Known Fault Types

Table 2: Main Results, 240 Incidents of Types F1–F8

Design	RCA top-1	Median detect (min)	Mean recover (min)	90th pct recover (min)	Fixed by automation	Budget per incident (%)	Pages per incident	Harmful actions per incident
Manual	–	6	46.3 ± 2.9	59	0%	15.2 ± 1.0	0.93	0
B0 threshold runbook	0.71	2	16.2 ± 2.2	52	72%	12.4 ± 1.8	0.35	0.53
B1 graph RCA + auto-fix	0.78	6	26.1 ± 3.1	59	73%	12.6 ± 1.9	0.24	0.17
B2 Autonomous SRE	0.97	6	18.6 ± 3.2	29	92%	2.4 ± 0.2	0.13	0.05

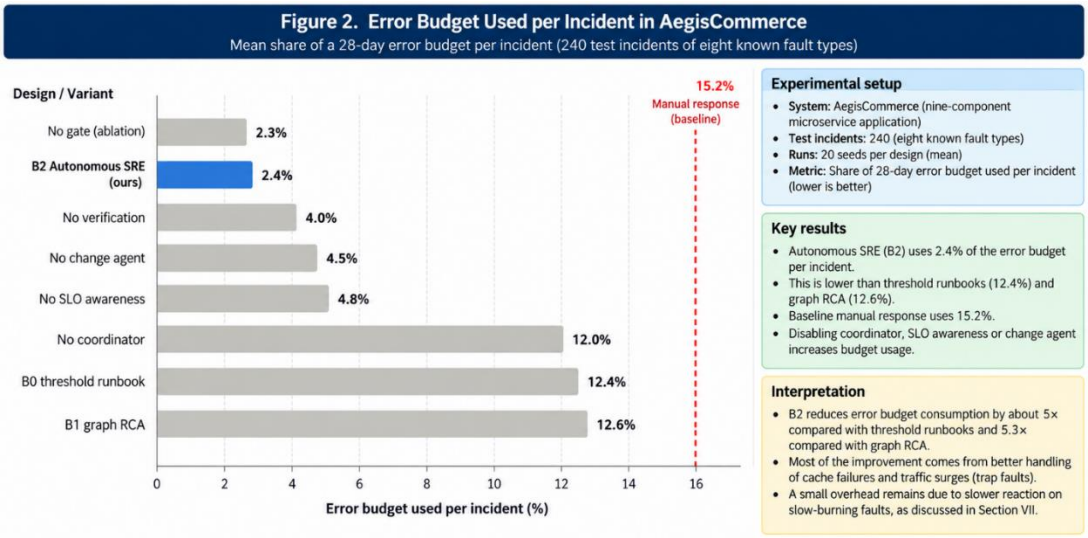


Figure 2: Mean Percentage of A 28-Day SLO Error Budget Consumed Per Incident Across 240 Simulated Incidents Representing Eight Known Fault Types.

Autonomous SRE (B2) Consumes 2.4% Of the Error Budget, Compared With 12.4% For Threshold-Based Runbooks (B0), 12.6% For Graph-Based Root-Cause Analysis (B1), And 15.2% For Manual Response. Ablation Results Illustrate the Contributions of SLO Awareness, Evidence Coordination, Change Analysis, Verification, And Approval Gating.

The clearest difference is in the error budget (Fig. 2). B2 spends 2.4% of a 28-day budget per incident, against 12.4% for B0, 12.6% for B1 and 15.2% for a person paged without automation. B2 also halves the long tail of recovery time: 90% of incidents are fixed within 29 minutes, against 52 and 59. And it causes about one harmful action in twenty incidents, against one in two for B0 and one in six for B1.

B2 is not the fastest on average, and Table II shows why. Threshold alerts fire within two minutes, while burn-rate alerts take a median of six. More importantly, half of the F2 incidents (lost capacity) burn the budget at between one and six times the sustainable rate. That is slow enough that neither burn-rate window confirms an alert within the four-hour episode. Those incidents count as unfixed until the end, which inflates mean recovery for every SLO-based design. They are also cheap: B2's F2 incidents cost 1.3% of the budget each. Ignoring them is what burn-rate alerting is designed to do, but it does mean an SLO-only system can leave a degraded service alone for hours. In practice a slower ticket-level alert would catch them [2].

Table 3: By Fault Type: RCA Top-1 / Mean Recovery (Min) / Budget Per Incident (%)

Fault	B0	B1	B2
F1 bad deploy	0.67 / 17.5 / 1.7	1.00 / 10.6 / 0.9	1.00 / 10.6 / 0.9
F2 capacity loss	0.71 / 16.4 / 2.8	0.80 / 72.0 / 3.1	0.93 / 67.8 / 1.3
F3 memory leak	1.00 / 21.0 / 1.1	1.00 / 16.2 / 0.8	1.00 / 16.2 / 0.8
F4 database degradation	1.00 / 2.0 / 1.7	1.00 / 17.2 / 2.0	0.96 / 29.5 / 3.0
F5 bad config	1.00 / 3.4 / 1.5	0.97 / 10.9 / 1.7	1.00 / 9.2 / 1.6
F6 slow provider	1.00 / 3.1 / 0.7	1.00 / 9.9 / 0.8	1.00 / 9.9 / 0.8
F7 cache failure	0.00 / 46.1 / 51.3	0.00 / 48.2 / 54.1	1.00 / 2.0 / 4.4
F8 traffic surge	0.53 / 23.9 / 38.3	0.53 / 24.1 / 37.5	0.87 / 3.5 / 6.7

Table III shows that the advantage comes almost entirely from the two traps. On F1, F3, F5 and F6, graph RCA is as accurate as B2 and recovers just as fast. Personalised PageRank finds a single misbehaving service very well. In F7 it blames the overloaded database every time, and the runbook fails the database over. That causes a two-minute outage, fixes nothing, and leaves a person to find the cache about 48 minutes after onset. B2 sees that cache traffic has vanished while database traffic has risen about fourfold, and restarts the cache within two minutes. In F8, B0 and B1 roll back the saturated catalog in roughly a third to two-fifths of incidents, misled by an unrelated deployment that happened to land there shortly before. B2 recognises a busy service in a surge and scales it out. F4 is the one case where B2 is slower, because it waits for a person to approve the failover.

B. Benign Episodes

Over 60 benign episodes (10 simulated days), B0's thresholds fired in 68% of episodes. It took 5.2 automatic actions a day, 4.3 of them restarts or failovers that caused needless outages. Both SLO-based designs alerted in 6.7% of episodes and took 0.2 actions a day. These were short blips that crossed the burn threshold and lasted past the two-minute dwell.

C. Fault Types Never Seen During Development

Table 4: Held-Out Faults U1–U3, 90 Incidents

Design	Fixed by automation	Mean recover (min)	Budget per incident (%)	Harmful actions per incident	Pages per incident
Manual	0%	56.3	2.7	0	0.72
B0	33%	44.3	3.5	0.62	0.70
B1	6%	65.0	3.6	0.49	0.67
B2	37%	49.4	2.8	0.47	0.43
B2 without verification	3%	66.4	3.6	0.43	0.71

The held-out faults are where verification earns its place. B2's first guess was almost never right (top-1 = 0.05): its rules were not written for these faults. In U1 the bad cart release shows up as errors in orders, so B2 first restarts or rolls back orders. Verification sees no improvement and moves on, and the rollback of cart, usually its second hypothesis, fixes the fault in 29 of 30 cases. Mean recovery is 27 minutes, against 65 for B1 and 64 for B2 without verification. Both of those page a person after their single guess fails.

U2, the expired certificate, has no automated fix. Every design that acts makes things slightly worse here. B2 spends about nine minutes and 0.43 harmful restarts per incident before paging, and recovers in 45 minutes against 36 for a person paged at once. A system like this needs a way to say "this is not mine" sooner, and Section VIII returns to this. U3, a slow disk, mostly burns less than the alert thresholds: SLO alerts fired in only 20% of cases, while B0's thresholds caught every one and failed the database over correctly.

D. Ablations

Each part does something different.

- Coordination matters most. Without the victim and busy checks, top-1 accuracy falls to 0.53 and budget use returns to B0's level. The decoy deployments and the overloaded database lead the raw agents astray.
- The change agent is the only source of evidence for bad deploys and bad configurations. Without it, F1 and F5 always end with a person.

- Verification barely changes the average on known faults, where the first guess is nearly always right. It shortens the tail (90th percentile 29 against 51 minutes) and cuts budget use from 4.0% to 2.4%. On unseen faults it is the difference between 37% and 3% fixed automatically. The price is slightly more wrong attempts on the way: 0.16 against 0.10 per incident.
- SLO awareness is a trade-off rather than a pure gain. Replacing burn-rate detection with thresholds gives the fastest recovery of any variant (6.7 minutes), because it acts early on slow faults such as F2 and U3. It also produces 5.1 automatic actions a day on a healthy system, 3.0 of them outages, and doubles budget use. Using thresholds to start the investigation but the burn rate to permit action is an obvious next step.
- The approval gate cost 1.5 minutes of mean recovery and caught nothing in this simulator. The coordinator almost never proposed a wrong failover, so harmful actions are the same with and without it. We keep it because the case it protects against, an automated failover of a healthy database, is rare but severe. That is a judgement, not a measured result.

Table V. Removing One Part of B2 At a Time (Known Faults, Benign Days)

Variant	RCA top-1	Mean recover (min)	90th pct (min)	Fixed by automation	Budget per incident (%)	Harmful per incident	Benign actions per day
B2 (full)	0.97	18.6	29	92%	2.4	0.05	0.2
No SLO awareness	0.92	6.7	14	95%	4.7	0.38	5.1
No verification	0.97	19.6	51	89%	4.0	0.03	0.2
No change agent	0.73	28.7	64	67%	4.5	0.33	0.2
No coordination	0.53	29.7	69	77%	12.0	0.49	0.2
No gate	0.97	17.1	25	93%	2.3	0.05	0.2

VIII. DISCUSSION, LIMITATIONS AND CONCLUSION

A. What the results do and do not show

The experiment supports three claims. First, most of the benefit of multi-agent analysis comes from a few cases where one anomaly score cannot tell a culprit from a victim. A failed cache behind an overloaded database and a surge behind a saturated service are the two we built in. Elsewhere a single PageRank ranking does as well, and we see no reason to pretend otherwise. Second, verification matters little when the first guess is right and a great deal when it is not. On faults the design had never seen, it raised the share fixed without a person from 3% to 37%. Third, tying action to the SLO rather than to thresholds removes most of the needless actions on a healthy system, at the price of slower reactions to faults that burn the budget slowly.

B. Limitations

- A simulator, not a cluster. The queueing model has no retries, timeouts, thread pools or garbage-collection pauses, and works in whole minutes. Real failures interact in ways it cannot show.
- The same people wrote the faults and the rules. The 0.97 top-1 accuracy on known faults is therefore optimistic. The held-out faults, where top-1 was 0.05, are a better guide to how the coordinator's rules would meet a new failure. They are still only three.
- A generous human. Our on-call engineer always picks the right fix within 20 to 45 minutes, and the approval behaviour is assumed rather than measured. Real responders are slower on unfamiliar faults and sometimes wrong.
- Shortened windows. The burn-rate windows were shrunk to fit four-hour episodes. Faults that burn slowly, like F2 and U3, were often never alerted on. U3 is a classic gray failure [14] and needs other signals.

- Unfixable faults. On U2 the automation tried, failed and only then paged, adding about nine minutes and some harmful restarts. An abstain rule is needed: if no hypothesis scores above a floor, page at once.
- Verification can be fooled. During development we saw a correct remedy reverted because an unrelated latency blip landed inside the verification window. Longer or repeated checks would reduce this but slow every recovery.
- Privilege. An automated responder that can roll back and fail over is a powerful identity in the cluster. Its own access control is outside this paper.

C. Validation on a live system

The design is meant to be checked on a real Kubernetes deployment. That means a microservice demo application, Prometheus recording rules for the burn rates, a chaos tool to inject F1–F8, and the agents running as a controller against the same measures. We have not done that run. The cells below are to be filled from it and must not be estimated.

Table 6: Prospective Live-Cluster Validation Protocol (Not Performed)

Measure	B1	B2
RCA top-1, F1–F8	Not measured	Not measured
Mean time to recover (min)	Not measured	Not measured
Error budget per incident (%)	Not measured	Not measured
Harmful actions per incident	Not measured	Not measured

D. Conclusion

Automating incident response is not mainly a ranking problem. Good root-cause rankers exist. The harder problem is acting safely when the ranking is wrong. Autonomous SRE treats every remedy as a hypothesis: it acts only when users are being hurt, tries the gentlest likely fix first, checks the error budget to see whether the fix worked, and undoes it if not. On a simulated nine-component application, it used a fifth of the error budget of threshold runbooks or graph-RCA automation, and caused a tenth as many harmful actions as the runbooks. It also fixed more than a third of never-seen faults without a person. The next steps are a live run (Table VI), an abstain rule for faults that automation cannot fix, and a hybrid trigger that starts investigating on early symptoms but acts only on the burn rate.

Code and Data Availability

The simulator, the compared designs, the ablations, and the evaluation scripts are included in the accompanying reproduction package; public repository publication has not been verified. The supplied evaluate.py script is intended to regenerate the reported results using test seeds 5–9. Archived results.json was cross-checked against the principal values in Table II; a complete independent rerun remains recommended.

Reproducibility and Evidence-Status Clarification

This study evaluates a discrete-time, synthetic nine-component system. It does not report a Kubernetes deployment, real-user traffic, real incident tickets, or LLM-based agent inference. The four evidence agents and coordinator are deterministic software modules.

The archive contains system.py, scenarios.py, ops.py, evaluate.py, episodes.json, and results.json. The archived results agree with the manuscript's principal known-fault comparisons (B0, B1, B2, and manual). An attempted full independent rerun exceeded the available execution window; therefore this revision does not claim independent end-to-end reproduction.

The held-out fault catalogue was developed after the main design was frozen according to the supplied development log, but the separation is supported by the authors' records rather than independently established provenance. Confidence intervals, alternative topology experiments, fault-intensity sensitivity, and live-cluster tests remain future validation work.

The architecture is not an unrestricted self-healing controller: restarts and database failovers cannot be reversed in the same sense as scaling or feature-switch actions. A failed verification can only trigger a compensating action when an inverse exists.

IX. REFERENCES

[1] B. Beyer, C. Jones, J. Petoff and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly, 2016.

[2] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara and S. Thorne, Eds., *The Site Reliability Workbook: Practical Ways to Implement SRE*. Sebastopol, CA, USA: O'Reilly, 2018.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
[4] L. Wu, J. Tordsson, E. Elmroth and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Proc. IEEE/IFIP NOMS*, 2020.
[5] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang and P. Wang, "AutoMAP: Diagnose your microservice-based web applications automatically," in *Proc. The Web Conference (WWW)*, 2020.
[6] P. Chen, Y. Qi, P. Zheng and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *Proc. IEEE INFOCOM*, 2014.
[7] Y. Gan, M. Liang, S. Dev, D. Lo and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proc. ASPLOS*, 2021.
[8] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi and M. Kocaoglu, "Root cause analysis of failures in microservices through causal discovery," in *Proc. NeurIPS*, 2022.
[9] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *Proc. ICSE*, 2023.
[10] Y. Chen et al., "Automatic root cause analysis via large language models for cloud incidents," in *Proc. EuroSys*, 2024.
[11] A. Basiri et al., "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
[12] Z. Li et al., "Gandalf: An intelligent, end-to-end analytics service for safe deployment in cloud-scale infrastructure," in *Proc. USENIX NSDI*, 2020.
[13] H. Liu, S. Lu, M. Musuvathi and S. Nath, "What bugs cause production cloud incidents?" in *Proc. HotOS*, 2019.
[14] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati and R. Yao, "Gray failure: The Achilles' heel of cloud-scale systems," in *Proc. HotOS*, 2017.
[15] The Kubernetes Authors, "Configure liveness, readiness and startup probes" and "Horizontal Pod Autoscaling," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/
[16] Argo Project, "Argo Rollouts – Kubernetes progressive delivery controller," and Flux Project, "Flagger." [Online]. Available: https://argoproj.github.io/rollouts/ and https://flagger.app/
[17] L. Page, S. Brin, R. Motwani and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.
[18] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro)service-based cloud applications: A survey," *ACM Computing Surveys*, vol. 55, no. 3, 2022.
[19] D. Huye, Y. Shkuro and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows," in *Proc. USENIX ATC*, 2023.
[20] N. Zhao et al., "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proc. ESEC/FSE*, 2021.