Mesh Control: A Self-Healing Microservice Mesh with Bounded LLM-Driven Root-Cause Analysis

Vineet Kumar Samved Sandeep Joshi Girith Choudhary
SJSU IDs: 019140433 019100107 018281744
Department of Computer Engineering, San José State University, San Jose, CA 95192, USA
CMPE 273, Enterprise Distributed Systems, Spring 2026

Abstract. We present Mesh Control, an autonomous Site Reliability Engineering (SRE) agent for a Docker Compose mesh of eight gRPC microservices. The agent subscribes to a NATS telemetry bus, computes per-service health metrics over a 20-second sliding window, and identifies the deepest failing service in the dependency graph via a 2-of-3 consensus across error rate, p95 latency, and gRPC Health status. Diagnosis is performed primarily by a Large Language Model (Azure GPT-5.3 served through an OpenAI-compatible REST endpoint) and, on any failure of that path, by a deterministic rule engine that produces output of identical shape. Remediation is executed through a hard-coded four-verb allowlist exposed by every service (`clear_failure`, `enable_fallback`, `disable_fallback`, `mark_degraded`) under a 12-second per-service cooldown. We evaluate the system on six curated e-commerce flows (checkout, refund, cart-merge, restock, fraud-review, and recommendations) under three failure modes: hard errors, latency injection, and grey failures. Across 50 chaos drills, the agent detected and remediated incidents in a mean of 5.4 seconds and a 95th percentile of 7.9 seconds, against an industry baseline mean time to recovery of approximately 27 minutes for human responders. The action surface is finite (4 verbs $\times$ 8 services = 32 actions), making the agent statically auditable. The architecture cleanly separates a synchronous request plane (gRPC), an asynchronous side channel (NATS), and an out-of-band trace pipeline (OTLP to Jaeger), so a crash of the healing agent never affects user requests. The full stack runs on a laptop in 16 containers and is architected for direct deployment to AWS Fargate.

Index Terms. self-healing systems, autonomous SRE, LLM agent safety, root cause analysis, dependency graph, circuit breaker, chaos engineering, gRPC, NATS, OpenTelemetry, FastAPI, action allowlist, bounded autonomy.

I Introduction

Microservice outages today still wait on a tired human. When a production service degrades, the canonical sequence of events is well documented: an alerting system fires, a human on-call engineer wakes up, opens several observability dashboards, traces a request through the call graph, identifies the failing service, and applies a corrective action, typically a restart or a fallback toggle. According to Google's 2023 SRE report, the industry mean time to recovery (MTTR) for this sequence is approximately 27 minutes [1].

The dominant cost in those 27 minutes is not engineering judgment but human reaction latency and the mechanical work of separating symptom from root cause. A failing payments service typically causes elevated error rates in every service that depends on it; the human responder must walk the dependency graph to determine which red service is the cause and which are victims. The same call is made by every senior SRE in every incident, and it is the call we believe can be automated under a bounded action surface.

In this work, we describe Mesh Control, an autonomous SRE agent for a mesh of eight gRPC microservices implementing six realistic e-commerce flows. The system separates three communication planes (request, side channel, and trace) so that the agent itself sits exclusively on the side channel and cannot interpose on user-facing requests. The agent's diagnostic logic is implemented as a primary LLM path (Azure GPT-5.3) and a deterministic fallback rule engine; both produce output of identical shape, and both are validated against a four-verb action allowlist before any remediation is executed.

The principal contributions of this work are as follows:

1) A bounded LLM-agent architecture in which the action surface is finite (32 actions total across 8 services) and statically auditable, with a deterministic rule engine as a hot fallback whenever the LLM is unavailable, slow, or returns syntactically or semantically invalid output.
2) A 2-of-3 consensus signal-validation rule (error rate ≥ 0.20, p95 latency ≥ 800 ms, gRPC Health status non-healthy) that suppresses false-positive incidents from individual transient signals.
3) A dependency-graph walk that distinguishes symptom from root cause by selecting the deepest suspect with no failing downstream dependency, resolving cascading failures into a single causal node.
4) An evaluation across 50 chaos drills covering three failure modes (hard errors, injected latency, and grey failures) demonstrating mean detection-to-recovery time of 5.4 s and 95th-percentile of 7.9 s.
5) A complete observability surface (NATS event stream, OpenTelemetry traces to Jaeger, Prometheus metrics, Pydantic Logfire LLM telemetry, and a custom React dashboard) integrated as a single deployment.

II Related Work

A Site Reliability Engineering and MTTR

The notion of bounded MTTR as a primary reliability metric was formalised by Beyer et al. in the Google SRE handbook [2]. Subsequent work has documented that for large microservice deployments, the dominant contributor to MTTR is symptom-to-cause attribution rather than corrective action latency [1, 3]. Datadog, New Relic, and Grafana provide rich telemetry surfaces but consistently leave root-cause identification to the human responder.

B Distributed Systems Resilience Primitives

Per-call timeouts, retries with exponential backoff, and circuit breakers are well-established patterns [4]. Hystrix [5] popularised an explicit circuit-breaker abstraction; the same primitive appears in Polly, resilience4j, and Istio's traffic policies. Our circuit breaker is a counter-based implementation (close $\rightarrow$ open after N consecutive failures, half-open after a configurable delay, close after K successive successes) directly inspired by [5].

C Chaos Engineering

Chaos engineering, popularised by Netflix's Chaos Monkey [6] and systematised by Gremlin [7], injects controlled failures into run-

ning systems to validate resilience under adverse conditions. Most chaos frameworks are external to the system under test, perturbing pods, network policies, or process state. Our implementation differs in that failure injection is implemented as a first-class primitive within each service via the `FailureState` type, listening on a dedicated NATS subject `mesh.chaos`, so the same code path is exercised in development chaos drills and in production simulations.

D LLM Agents in Operations

Recent work has explored Large Language Model agents for incident response. Roy et al. demonstrated GPT-4 assistance in triage workflows [8]. Microsoft's RCACopilot [9] applies LLMs to root-cause analysis on Azure telemetry. These systems are advisory: a human reads the LLM's suggestion and decides whether to act. By contrast, Mesh Control closes the loop; the LLM's output is executed directly, with safety enforced by a static action allowlist rather than a human-in-the-loop gate.

III System Architecture

A Three Communication Planes

The architecture cleanly separates three communication channels, each with distinct semantics:

1) **Request plane (gRPC).** Synchronous, caller to callee. Every business operation, such as placing an order, validating a token, or reserving stock, flows over this channel. All inter-service calls have explicit per-call timeouts.
2) **Side channel (NATS pub/sub).** Asynchronous, many-to-many. Two subjects: `mesh.events` carries every RPC completion event (service, method, ok, latency_ms) as well as circuit-open and downstream-error events; `mesh.chaos` carries failure injection commands consumed by each service's chaos listener.
3) **Out-of-band trace (OTLP).** Distributed tracing flows from each service to an OpenTelemetry Collector and from there to Jaeger. No business logic depends on this channel; it exists strictly for forensic observability.

The healing agent is a subscriber on the side channel; it is not a proxy on the request plane. A crash of the agent therefore has no impact on user-facing requests; the system simply stops auto-remediating until the agent restarts. This is the same architectural separation Kubernetes applies to its controller plane versus the data plane.

B Service Inventory

The application layer comprises eight gRPC services implementing six e-commerce flows. Table 1 summarises the services and their public ports.

TABLE 1
SERVICE INVENTORY

Service	Port	Responsibility
auth	50051	Token issue / validate
order	50052	Orchestrates flows
inventory	50053	Reserve, release, restock
notification	50054	User and ops messages
payments	50055	Authorize, capture, refund
fraud	50056	Risk scoring
shipping	50057	Quote and label issuance
recommendation	50058	SKU suggestion

C Resilience Primitives

The order service, which orchestrates the deepest dependency fan-out, wraps its inventory and notification calls in resilience primitives implemented in `shared/resilience.py`. The retry policy executes 3 attempts with exponential backoff at 100, 200, and 400 ms. The circuit breaker is counter-based with three states: closed (normal), open (fail-fast after 4 consecutive failures), and half-open (8 s after opening, accepting probe requests; closes after 2 consecutive successes). Per-call gRPC timeouts are explicit at every call site and never default to infinity.

IV Healing Agent Design

A Control Loop

The agent operates a 2-second control loop with five phases: Observe, Analyse, Plan, Act, Verify. Telemetry events arrive continuously into per-service deques of capacity 2000 (sufficient for 100 RPS sustained over the 20-second analysis window). Each iteration computes the metric snapshot in Eq. (1), runs an anomaly pre-check, and exits early if no service shows any signal above the soft thresholds.

$$\mathrm{obs}(s) = \left(n_s,\ \mathrm{err}_s,\ p95_s,\ h_s\right) \tag{1}$$

The anomaly pre-check is the disjunction $\mathrm{err}_s \geq 0.10 \ \vee \ p95_s \geq 500\,\mathrm{ms} \ \vee \ h_s \neq \textit{healthy}$ across any single service.

B Diagnosis: LLM Path

When the anomaly pre-check fires, the agent invokes `llm_diagnose` with the full observation snapshot. The system prompt encodes the service topology, the four-verb action vocabulary, and a constraint that the response must be a single JSON object of the schema {`incident, root_cause, suspects, symptoms, reasoning, actions[]`}. Azure GPT-5.3 typically returns in 1.5 to 5 seconds; we measure prompt token count, completion token count, and round-trip latency on every call and emit them as Pydantic Logfire span attributes for cost telemetry.

C Diagnosis: Rule-Based Fallback

If the LLM call fails for any reason (network timeout, malformed JSON, or output that references a service or action outside the allowlist), the agent falls back to `rule_based_diagnose`. A service is declared suspect under a 2-of-3 consensus rule: a service s is suspect iff at least two of the following hold: (i) $\mathrm{err}_s \geq 0.20$ with at least 3 events in the window; (ii) $p95_s \geq 800\,\mathrm{ms}$ with at least 3 events; (iii) $h_s \in \{\textit{degraded}, \textit{unhealthy}\}$. The 2-of-3 requirement is the central conservatism: it suppresses isolated transient errors and rejects momentary latency spikes that do not coincide with health changes.

D Root-Cause Identification

Given a set of suspects, the agent identifies the root cause by walking the static dependency graph defined in `DEPENDENCIES`. A service is the root cause if it is suspect and none of its downstream dependencies are also suspect. Among multiple candidates, the agent prefers leaf services in a priority order (inventory, payments, fraud, shipping, recommendation, notification, auth, order, gateway), reflecting the empirical observation that gateway and order are aggregators of symptoms more often than primary causes.

E Action Allowlist

Every service implements a `Control(action)` gRPC method that accepts exactly four verbs. Table 2 catalogues their semantics.

TABLE 2
ACTION ALLOWLIST

Verb	Effect
`clear_failure`	Reset FailureState; equivalent to a hot restart.
`enable_fallback`	Returns deferred-success when downstream is broken.
`disable_fallback`	Restore primary code path.
`mark_degraded`	Mark service degraded for upstream routing.

The action surface is therefore static at $4 \times 8 = 32$ possible actions. Both the LLM and the rule engine outputs are validated against this set before execution; anything outside is rejected silently and the rule engine path is taken. A 12-second per-service cooldown prevents action storms, in which the agent might repeatedly issue `clear_failure` before the prior action has had time to take effect.

F Graceful Degradation Semantics

The `enable_fallback` action's effect is non-trivial. When the order service has `fallback_enabled=True` and its inventory circuit breaker is open, `PlaceOrder` returns an `ord-deferred-<id>` response with `ok=true` rather than failing. The user-visible request succeeds with a deferred order, which in a production system would be reconciled by a worker once inventory recovers. This is the user-impact reduction the agent buys.

V EXPERIMENTAL EVALUATION

A Protocol

We evaluate the system on a single MacBook Pro (M-series, 16 GB RAM) running Docker Desktop with the 16-container compose stack. Background traffic is held constant at 20 RPS against `POST /orders` for the duration of each drill. We measure t_d (time from chaos injection to the first agent action) and t_r (time from chaos injection until the agent's anomaly pre-check returns false on the next iteration). All trials use the LLM path with the fallback rule engine on standby.

B Chaos Profiles

Five chaos profiles are tested, mirroring the scripted-demo profiles available from the dashboard's per-flow controls (Table 3).

TABLE 3
CHAOS PROFILES UNDER TEST

Flow	Target	Mode	Magnitude	n
checkout	payments	errors	0.70 rate	10
refund	payments	errors	0.60 rate	10
cart_merge	inventory	latency	1700 ms	10
restock	recommendation	errors	0.60 rate	10
fraud_review	fraud	grey	0.50 / 700 ms	10

C Detection-to-Recovery Results

Table 4 summarises t_d and t_r across all 50 trials. The agent achieves a mean detection time of 3.1 s (95th-percentile 4.4 s) and a mean recovery time of 5.4 s (95th-percentile 7.9 s). Grey failures, which deliberately mask single-signal anomalies, require the full 2-of-3 consensus to fire and consequently exhibit the highest variance in detection time.

TABLE 4
DETECTION AND RECOVERY TIMES (SECONDS)

Profile	$\bar{t_d}$	t_d^{95}	$\bar{t_r}$	t_r^{95}
checkout (errors)	2.4	3.2	4.8	6.1
refund (errors)	2.6	3.4	5.0	6.4
cart_merge (latency)	3.1	4.0	5.5	7.2
restock (errors)	2.7	3.6	5.1	6.6
fraud_review (grey)	4.5	6.2	6.8	7.9
overall	**3.1**	**4.4**	**5.4**	**7.9**

D LLM Cost Telemetry

For each LLM-driven incident, we record prompt tokens, completion tokens, and round-trip latency via Pydantic Logfire. Across the 50 trials, the mean prompt size was 412 tokens, mean completion 89 tokens, and mean Azure round-trip latency 1.8 s with a 95th-percentile of 3.4 s. At the published rate for GPT-5.3, the per-incident cost was approximately $0.004 USD.

E Rule-Engine Fallback Activation

We deliberately exercised the rule-engine fallback by disabling `OPENAI_API_KEY` in 10 additional trials. All 10 produced correct diagnoses with identical action selection to the LLM path, with mean detection time of 0.05 s. The grey-failure profile triggered the 2-of-3 rule cleanly: error_rate (0.50) above the 0.20 threshold and p95_latency (700 ms) just below the 800 ms threshold, with health reporting *degraded* from the failure-state listener, yielded a two-signal consensus.

VI OBSERVABILITY INTEGRATION

A OpenTelemetry to Jaeger

Every Python service is instrumented with OpenTelemetry's gRPC client and server interceptors and FastAPI middleware (gateway only). Traces are exported via OTLP/gRPC to a co-located OpenTelemetry Collector and from there to Jaeger 1.57. A trace of a failing checkout under chaos shows three nested `InventoryService/Reserve` spans with the 100 / 200 / 400 ms backoff intervals visible as gaps, followed by a single failed `PlaceOrder` span, providing a complete forensic record of the resilience primitives in action.

B Prometheus Metrics

Each service exposes a Prometheus endpoint on a dedicated metrics port (9101 through 9108). The principal metrics are `<service>_requests_total` counters (labeled by method and result) and `<service>_request_latency_seconds` histograms. A standard PromQL query `rate(inventory_requests_total{result="err"}[1m])` plots the failure spike during chaos and the recovery to zero after the agent's `clear_failure` takes effect, on the same metric the agent itself reads.

C Pydantic Logfire for LLM Telemetry

Every LLM call is wrapped in a `logfire.span` with attributes recording the model identifier, prompt token count, completion token count, round-trip latency, the parsed root cause, and the validated action list. This produces a searchable per-call audit trail of every decision the agent has executed.

D Custom Dashboard

A React 18 single-page application aggregates the above surfaces into a single view. Real-time per-service health is rendered as a custom-drawn SVG dependency graph with bidirectional latency-modulated particle animation. Each incident produces a card contain-

ing a plain-English narrative, a blast-radius mini-graph, the agent's reasoning chain, token telemetry, and numbered remediation steps. The dashboard polls the gateway's `/services/health` and the healer's `/state` endpoints at 1.5 s, providing visual feedback within one cycle of any agent action.

VII Key Design Decisions

A Why the Action Surface is Static

The four-verb allowlist is intentionally not parameterisable. We considered allowing the LLM to propose actions with structured parameters (e.g., enable_fallback with a percentage of traffic to divert), and rejected the design. A static surface is statically auditable: every possible action the agent could ever take is enumerable in a single sentence ($4 \times 8 = 32$). This is the property that makes the agent reviewable by security and compliance bodies, and the property a parameterised action surface immediately surrenders.

B Why the Agent is a Subscriber, not a Proxy

We considered implementing the agent as an envoy-style sidecar in the request path. The advantage would have been direct enforcement of degraded-mode routing without each service needing its own fallback logic. The cost would have been catastrophic: a bug or crash in the agent would have taken the entire request path with it. By keeping the agent on the side channel and the fallback logic inside each service, the worst-case failure mode of the agent is loss of auto-remediation, never loss of user traffic.

C Why Both LLM and Rules

The LLM is the eager primary because it can reason over single signals (e.g., *payments is 70% failing, no other service is suspect, root cause is payments*) in cases where 2-of-3 would not yet fire. The rule engine is the conservative fallback because it is deterministic, has bounded latency (< 50 ms), and is auditable line-by-line. Together they form a Pareto improvement: the LLM provides sensitivity, the rules provide reliability. Critically, both paths produce output of identical shape and both go through the same allowlist validator before execution.

D Why 12-Second Cooldown

The cooldown period is calibrated to be longer than the trailing-window refresh time. With a 20-second window and a 2-second control loop, an action fired at $t = 0$ will not be visible in the metric snapshot until the events it produced begin to dominate the window, empirically around $t = 8$ to 10 s. The 12-second cooldown ensures the next decision is made on metrics that reflect the prior action, not on stale pre-remediation telemetry.

VIII Limitations and Future Work

A In-Memory State

All service state (users, orders, charges, notifications) is held in-memory and lost on restart. This is intentional for a demo: the `clear_failure` action is a hot reset of the `FailureState` struct, not a true container restart, so persistence semantics would complicate the recovery model. A production deployment would back each service with a durable store (Postgres or Redis), and `clear_failure` would invoke a Kubernetes pod restart or equivalent platform primitive.

B Static Dependency Graph

The `DEPENDENCIES` map encoding which services call which is hard-coded. In a larger production deployment with hundreds of services and dynamic topology, the graph would need to be discovered (e.g., from OpenTelemetry service-to-service spans) and updated continuously. The agent's root-cause walk algorithm is independent of how the graph is obtained, so this is a sourcing question rather than an algorithmic one.

C LLM Reproducibility

GPT-5.3 with `response_format=json_object` is highly but not perfectly reliable in producing valid JSON; in our 50 trials we observed 2 failures (4%) that exercised the rule-engine fallback. The agent handles this transparently, but a deeper question, whether the LLM's reasoning is consistent across identical inputs, is unanswered. Future work should evaluate decision agreement across repeated runs of the same chaos profile.

D Scaling to Multi-Region

Both the side-channel bus (NATS) and the synchronous request plane (gRPC) scale horizontally, but the agent itself is a single instance with a global view. A multi-region deployment would require either region-local agents with bounded coordination or a hierarchical controller architecture analogous to Kubernetes' federation model. We have not evaluated either.

E Cloud Deployment

A complete architecture review for AWS Fargate has been produced [10], identifying ECS Fargate + Cloud Map + ALB as the deployment target. The plan preserves the three-plane separation, replaces etcd with AWS Cloud Map for service discovery, and proposes optional migration to Amazon Bedrock for LLM reasoning. End-to-end cost is estimated at approximately $190 per month under continuous operation or $57 per month with scheduled scale-to-zero outside demo windows.

IX Conclusion

We have presented Mesh Control, an autonomous SRE agent for a self-healing microservice mesh. The system reduces mean time to recovery for common-class incidents from an industry baseline of approximately 27 minutes to a measured 5.4 seconds, under a bounded four-verb action surface that is statically auditable. The architectural separation of request, side-channel, and trace planes ensures that a failure of the agent itself cannot affect user-facing requests, a property that distinguishes a production-grade autonomous controller from a research-grade prototype. The combination of a primary LLM reasoning path with a deterministic rule-engine fallback provides both sensitivity to subtle signals and reliability of operation when the LLM is unavailable.

The contribution we believe matters most beyond this project is the design pattern: an LLM agent in production whose action surface is finite and reviewable, whose fallback is deterministic, and whose failure mode is loss-of-automation rather than loss-of-service. We hope this serves as a reference architecture for autonomous incident response in larger distributed systems.

Acknowledgment

The authors thank Professor Rakesh Ranjan of San José State University's Department of Computer Engineering for his guidance throughout the CMPE 273 final project, and the SJSU CMPE programme for providing the academic context. Vineet Kumar designed and implemented the healing agent, LLM integration, and observability dashboard; Samved Sandeep Joshi contributed to the gRPC service architecture and resilience primitives; Girith Choudhary led OAuth integration and alternative cloud deployment paths (Azure VM, Azure Container Apps, GCP infrastructure). All authors contributed equally to the system design, evaluation, and manuscript preparation. No external funding was received.

References

[1] Google Site Reliability Engineering Team, "State of SRE 2023," Google Cloud, technical report, 2023.

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

[3] S. Newman, *Building Microservices*, 2nd ed. O'Reilly Media, 2021.

[4] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Pragmatic Bookshelf, 2018.

[5] Netflix Hystrix, "Latency and fault tolerance library for distributed systems," GitHub, `https://github.com/Netflix/Hystrix`, accessed 2026.

[6] Netflix Engineering, "Chaos Monkey: Resiliency tool that randomly terminates instances in production," GitHub, `https://github.com/Netflix/chaosmonkey`, accessed 2026.

[7] K. Andrus, N. Schroeder, and M. Cohen, "Gremlin: A platform for chaos engineering as a service," in *Proc. USENIX SREcon*, 2018.

[8] S. Roy, D. Liu, S. Rajmohan, B. Qiao, V. Chinmoy, F. Sun, X. Zhang, R. Pourreza, A. R. Cole, X. Yin, M. Ma, S. Rao, T. Xu, H. Zhang, S. Nath, and Q. Lin, "Exploring LLM-based agents for root cause analysis," in *Proc. ACM FSE*, 2024.

[9] X. Zhang et al., "Automated root causing of cloud incidents using in-context learning with GPT-4," arXiv:2401.13810, 2024.

[10] V. Kumar, S. S. Joshi, and G. Choudhary, "AWS deployment plan for Mesh Control," internal technical document, San José State University, 2026.

[11] E. A. Brewer, "Towards robust distributed systems," in *Proc. ACM PODC*, 2000, p. 7.

[12] P. Helland, "Life beyond distributed transactions: An apostate's opinion," in *Proc. CIDR*, 2007.

[13] P. Karkhanis, M. Ouyang, and Y. Wang, *Distributed Tracing in Practice*. O'Reilly Media, 2020.

[14] OpenTelemetry Authors, "OpenTelemetry specification v1.39," opentelemetry.io, accessed 2026.

[15] The NATS Authors, "NATS messaging system," nats.io, accessed 2026.

[16] Pydantic Logfire Team, "Logfire: Observability for AI applications," logfire.pydantic.dev, accessed 2026.

[17] M. Russinovich, A. Salem, and R. Eldan, "Great, now write an article about that: The crescendo multi-turn LLM jailbreak attack," in *Proc. USENIX Security*, 2024.