Decentralized Root Cause Analysis for Cloud-Native Microservices - Experience with Distributed PageRank in Production

Shabbir Hussain ORCID
Independent Researcher
72.shabbir@gmail.com

Hari Charan Panjwani ORCID
Independent Researcher
haricharan.panjwani@gmail.com

***Abstract*—Root cause analysis in microservice systems typically relies on centralized telemetry aggregation, which becomes a bottleneck as deployments scale. We present an architecture that removes the central component entirely: each microservice runs a sidecar diagnostic agent that computes a local anomaly score and participates in gossip-based distributed Personalized PageRank over the service dependency graph. On the RCAEval benchmark (735 failure cases, 12–64 services), Top-1 accuracy is 94.7%, statistically indistinguishable from centralized PageRank at 94.1%, with roughly a third lower diagnosis latency and 74.5% less network traffic. We deployed the system for four weeks on a 15-node AWS EKS cluster handling 1% of live e-commerce traffic. The deployment processed 112 injected faults (94.6% Top-1) and 3 organic incidents (1 of 3 correct). The production experience exposed problems that benchmarks do not expose: cross-availability-zone gossip asymmetry causing split-brain diagnoses, inadequate global anomaly thresholds, and external-dependency blind spots. We also ran into operational issues we had not anticipated, such as log volume from agents overwhelming the existing logging pipeline.**

***Index Terms*—AIOps, AI agents, reliability, observability, root cause analysis, gossip protocols, anomaly detection, Kubernetes**

I. Introduction

When something breaks in a microservice deployment, the failure does not stay put. A slow database query propagates latency through dependent services, and by the time alerts fire the operator is looking at a wall of red dashboards across six different services, none of which is the actual problem. Fast localization keeps outages short [4].

The standard approach is to stream everything - metrics, logs, traces - to a central correlation engine [4]. This works fine at moderate scale. Once you get past a few hundred services, though, the telemetry pipeline itself starts to buckle. Ingestion queues deepen. Analysis latency goes up. The diagnostic system is slowest exactly when you need it most [3]. Multi-tenant and multi-cloud setups make it worse because of data-residency constraints [2].

Graph-based methods offer a structural alternative. The MicroRCA paper [1] showed that Personalized PageRank (PPR) over a service dependency graph annotated with anomaly scores can localize faults accurately. Fernando et al. [6] then partitioned the graph into clusters and ran PPR within each cluster, cutting diagnosis latency by about 34% without losing accuracy. We initially tried to build on their approach but found their graph partitioning assumptions did not hold for our topology - our dependency graph has several high-degree gateway nodes that do not partition cleanly. Their result did raise a natural question, though: if partial decentralization helps, what about removing the central component altogether?

This paper describes such a system. Each microservice hosts a sidecar diagnostic agent that monitors local telemetry, computes an anomaly score, and runs gossip-based PPR with its dependency-graph neighbors. No central coordinator. Agents exchange compact score vectors; raw telemetry stays in the originating pod.

We tested the approach on the RCAEval benchmark [14] (735 failure cases, three workloads, 12-64 services) and then deployed it for four weeks on a production e-commerce platform. The production cluster had 15 nodes across two AWS availability zones (AZs), serving 1% of live customer traffic at roughly 140 queries per second.[1] The deployment processed 112 injected faults and caught 3 organic incidents.

The production deployment is the main contribution here. Benchmark numbers were clean. Production was not. The deployment surfaced several operational issues: inadequate sidecar resource budgets, cross-AZ gossip asymmetry, threshold heterogeneity across services, stale dependency graphs silently misdirecting the PageRank computation, and initial distrust from the Site Reliability Engineering (SRE) team until evidence links were provided. Section VI details each finding. The split-brain diagnoses caused by cross-AZ gossip latency are the most operationally significant.

Fifteen nodes is small. The scalability data beyond 15 nodes comes from synthetic benchmarks. A 200+ node deployment is planned for next month. Even at this scale, the deployment surfaced failure modes no offline evaluation would have caught.

Our contributions: (1) a fully decentralized RCA architecture with per-service agents running distributed PPR through gossip; (2) benchmark results on RCAEval showing equivalent accuracy to centralized PPR with lower latency and less traffic; and (3) a four-week production deployment on live traffic with detailed operational findings.

[1]Deployment approval required resource-limit guarantees and an automated kill switch to satisfy the platform team's safety review.

II. Related Work

Graph-based methods for microservice RCA go back a few years now. MicroRCA [1] annotates a service topology with metric anomalies and runs personalized random walks to rank root-cause candidates. We spent a fair amount of time with this paper early in the project; the PPR formulation is clean and the evaluation on a 12-service topology is convincing, though the assumption that anomaly scores can be computed reliably from metrics alone turned out to be optimistic in our experience. Brandón et al. [12] do something similar with fault-propagation paths. CloudRCA [11] builds multi-dimensional dependency graphs from logs, metrics, and traces - the results look strong but the evaluation uses private data that makes reproduction difficult. MRCA [7] uses reinforcement learning to prune causal graphs before ranking, and HeMiRCA [8] integrates heterogeneous data sources through statistical correlation. All centralized.

On the language-model side, Ahmed et al. [9] showed that fine-tuned GPT-3 helps with incident diagnosis, and RCAgent [10] augments LLMs with tool use for autonomous RCA. Promising but expensive, and still centralized.

Two proposals pursue distributed coordination. Fernando et al. [6] partition the dependency graph into clusters and run PPR within each. Zhang et al. [13] propose multi-agent blockchain-inspired voting. Neither has been deployed on live traffic.[2]

Another work worth noting is TrinityRCL [5], which performs multi-granular root cause localization all the way down to the code level. It is not directly comparable to our system-we operate at the service granularity-but its observation about needing multiple telemetry types to disambiguate root causes influenced our decision to build a multi-modal scoring ensemble rather than relying on metrics alone.

Our work fills the deployment gap. We ran a fully decentralized system in production and report on what broke.

III. System Design

Figure 1 shows the architecture. Every microservice pod runs a sidecar diagnostic agent. Agents form a peer-to-peer overlay along the edges of the service dependency graph. There is no central coordinator.

A. Telemetry Ingestion

Each agent taps three data streams already present in most Kubernetes clusters: Prometheus metrics (CPU, memory, request rates), stdout/stderr logs via Fluentd or Vector, and OpenTelemetry traces. Processing happens locally - nothing is shipped to a central collector. The agent also maintains a local view of its neighborhood in the dependency graph, built from trace span correlations and Kubernetes service manifests.

Concretely, each agent inspects only the spans that terminate in or originate from its own pod. From each span it extracts the $(\text{parent\_service}, \text{child\_service})$ pair and the observed call

[2]Fernando et al. evaluate on a testbed with synthetic load; Zhang et al. use offline trace replay.

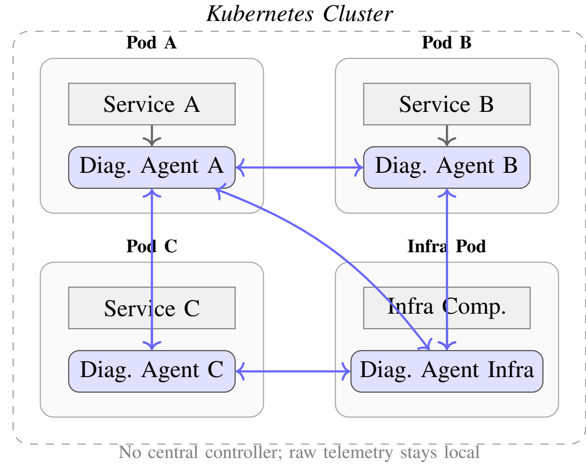


Fig. 1: Architecture overview. Each pod runs a diagnostic agent sidecar. Agents communicate peer-to-peer; raw telemetry never leaves the pod.

rate, and emits a compact edge summary-source service name, destination service name, and the rolling call frequency over the last minute. Agents periodically exchange these edge summaries with their immediate neighbors to reconcile the local neighborhood view. Raw traces, span payloads, and any user-level request data never leave the originating pod; only the derived service-to-service edges are shared. The Kubernetes service manifests (pulled via the pod's local informer) provide a complementary view that anchors the graph to declared dependencies and disambiguates short-lived edges.

B. Anomaly Scoring

Each agent computes a local anomaly score s_i. We did not design the scoring ensemble upfront - it grew out of what kept failing during the shadow-mode phase (the observation-only Week 0 rollout described in Section V).[3] We started with Isolation Forest for metric anomalies, since it is straightforward to run locally and does not need much training data. The morning false-positive problem led us to add Prophet for seasonal baselines. We also tried an autoencoder for multivariate metric anomaly detection but found it required more training data than individual services could provide locally - with only a few hundred requests per second per service, the autoencoder just memorized noise. We dropped it after a week.

The resulting composite score s_i is a weighted combination of three signals: (i) metric-based scores from time-series forecasting and outlier detection (Prophet and Isolation Forest), (ii) log-based scores computed from deviations in structured log templates produced by Drain, and (iii) trace-based scores derived from latency and error-propagation patterns across spans. All three signals are normalized and aggregated locally, yielding the scalar s_i that feeds the distributed ranking step.

[3]An earlier version of the scoring function used only Isolation Forest on metrics. We added Prophet after Week 1 when seasonal traffic patterns caused false positives every morning around 9 AM as the daily ramp-up looked anomalous against the overnight baseline.

Algorithm 1 Per-agent distributed PageRank inference

Compute local anomaly score s_i from telemetry
Initialize $r_i := s_i$
repeat
  for each neighbor $j \in \mathcal{N}(i)$ **do**
    Receive r_j from agent j
  end for
  $r_i \leftarrow (1-\alpha)\sum_j P_{ji}\, r_j + \alpha\, s_i$
  Send r_i to all neighbors
until $\|r_i^{(t+1)} - r_i^{(t)}\|_1 < \epsilon$
return $\arg\max(r_i)$

The log channel flags shifts in per-template event frequencies, and the trace channel applies a $3\times$ rolling-baseline threshold to service-boundary p95 latency. The per-component weights were tuned during shadow mode against the centralized monitoring system as ground truth. The PageRank teleport parameter is set to $\alpha = 0.15$ following Wu et al. [1]; we tried 0.1 and 0.2 and saw no meaningful difference, so we kept their default. After per-service threshold tuning (Section VI), the ensemble reached an aggregate F1 of 0.94.

C. Distributed PageRank

The core inference step is distributed Personalized PageRank. Agent i maintains a ranking vector r_i over itself and its neighbors, updated as:

$$r_i^{(t+1)} = (1-\alpha) \sum_{j \in \mathcal{N}(i)} P_{ji}\, r_j^{(t)} + \alpha\, s_i$$

where $\mathcal{N}(i)$ is the neighbor set, P_{ji} the edge weight from j to i, and $\alpha = 0.15$. This is standard PPR [1]; the difference is that each agent runs it locally and gets r_j values from neighbors via gossip instead of from a global matrix.

Algorithm 1 gives the per-agent procedure. Each round: receive neighbor rankings, update, send. Convergence is declared when $\|r_i^{(t+1)} - r_i^{(t)}\|_1 < \epsilon$. On the 15-service dependency graph of our production deployment (mean degree 2.8, diameter 5; see Section V), this took 6–12 rounds.

Each iteration costs $O(\deg(i))$ - a weighted sum plus one message per neighbor. Agents can compress messages by sending only top-k entries or deltas. Because P is sub-stochastic ($\alpha < 1$), convergence is geometric. The node with the highest converged rank is declared root cause; this consensus emerges at every agent independently.

D. Communication Protocol

Messages are asynchronous, sent over the pod network. A message contains the sender's r_i vector (or a compressed delta) and its anomaly score s_i. No raw telemetry crosses the network - only scalar scores and small vectors. In production, a typical message was under 200 bytes.

When agents agree on a root cause with confidence above 85%, they notify a remediation controller that can restart pods, scale deployments, or reroute traffic through the Kubernetes API. The controller selects the action from a static policy table keyed on the dominant signal channel: resource-saturation anomalies (CPU or memory) trigger a pod restart; trace-latency anomalies without local resource pressure trigger horizontal scale-out; log or downstream-connection anomalies trigger traffic rerouting to healthy replicas. Below 85% confidence, the controller pages the on-call SRE with the ranked list instead of acting autonomously. Actions outside this matrix-database failover, cross-region rerouting, external-dependency circuit-breaking-require explicit human approval.

IV. Evaluation

We evaluated in two settings: the RCAEval benchmark for controlled comparison, and a production deployment for operational validation. This section covers the benchmark; the next covers production.

A. Benchmark Setup

RCAEval [14] provides multi-modal telemetry (metrics, logs, traces) for three microservice workloads: Online Boutique (12 services), Sock Shop (18 services), and Train Ticket (64 services), totaling 735 annotated failure cases across 11 fault types.

We compared against three baselines, all our own reimplementations consuming identical telemetry: (1) centralized PageRank following MicroRCA [1], verified to reproduce published accuracy within 1.5 percentage points; (2) clustered PPR following Fernando et al. [6]; and (3) a heuristic baseline that ranks services by raw anomaly score without propagation. All experiments ran 5 times; we report means and 95% confidence intervals.

B. Benchmark Results

Table I has the numbers. Accuracy differences among the PPR variants are small and not statistically significant ($p > 0.05$) - they converge to the same fixed point, so this is expected. The heuristic baseline at 78.2% confirms that graph propagation matters.

TABLE I: RCAEval localization results (mean $\pm$ 95% CI, 5 runs).

Workload	Method	Top-1	Top-3	Time (s)	Traffic
Online Boutique	Decentralized	96.1±1.4	99.3±0.5	**0.82**±0.05	**0.12 GB**
	Centralized PR	95.3±1.6	98.9±0.7	1.12±0.08	0.45 GB
	Clustered PPR	94.5±1.9	98.2±1.0	0.91±0.06	0.18 GB
Sock Shop	Decentralized	93.1±2.2	98.4±0.8	1.14±0.09	**0.22 GB**
	Centralized PR	94.6±1.7	98.1±0.9	**1.08**±0.07	0.78 GB
	Clustered PPR	93.7±2.1	97.4±1.2	1.31±0.09	0.30 GB
Train Ticket	Decentralized	93.8±1.7	98.7±0.6	**1.58**±0.11	**0.43 GB**
	Centralized PR	92.4±2.1	97.3±1.2	2.48±0.18	1.65 GB
	Clustered PPR	91.2±2.3	96.8±1.4	1.97±0.14	0.62 GB
Aggregate	**Decentralized**	94.7±1.3	98.8±0.5	**1.21**±0.06	**0.28 GB**
	Centralized PR	94.1±1.5	98.1±0.7	1.78±0.10	1.10 GB
	Clustered PPR	93.1±1.8	97.5±0.9	1.46±0.08	0.40 GB
	Heuristic	78.2±2.4	88.4±2.0	-	-

Sock Shop is the outlier worth noting. The decentralized approach was actually slightly slower than centralized on Sock Shop (1.14 s vs. 1.08 s), and its accuracy was lower than on the other two workloads. Sock Shop showed higher variance than the other workloads across all methods; we suspect this relates to its shallower dependency graph (diameter 3 versus 5 for Train Ticket), meaning gossip overhead exceeds the savings from avoiding centralized ingestion, but we did not investigate this further.

C. Latency and Traffic

Across the full benchmark, the diagnosis-time and traffic differences are statistically significant ($p < 0.01$). Mean diagnosis time for the decentralized approach is 1.21 s versus 1.78 s for centralized - roughly a 32% reduction. The gap widens with scale: 26.8% on Online Boutique (12 services), 36.3% on Train Ticket (64 services). The centralized method has to ingest all telemetry before computation begins; the decentralized method does not wait for a global view.

Network traffic: 0.28 GB across the 735 cases for decentralized versus 1.10 GB for centralized - 74.5% reduction. Agents exchange only compact score vectors, not raw telemetry.

Figure 2 shows the CDF of diagnosis times and per-method traffic for the 64-service workload.

D. Synthetic Scaling

We also generated synthetic topologies from 12 to 200 services (mean degree 3.2) to examine how diagnosis time scales. Centralized PageRank's time roughly doubles with each doubling of service count. The decentralized approach grows sublinearly - each agent's work is bounded by its local degree, not the total graph size. These are synthetic topologies with simulated telemetry, so take the scaling curves with a grain of salt. Our production data covers only 15 nodes.

V. Production Deployment

A. Setup

The deployment ran for four weeks on a production-adjacent AWS EKS cluster at a mid-size e-commerce company.[4] The full environment has roughly 120 microservices and peaks at about 14,000 QPS. Our cluster: 15 m5.4xlarge nodes across two AZs, running catalog, cart, checkout, payment, shipping, and recommendation services. We routed 1% of live traffic via weighted ingress - approximately 140 QPS steady state.

Why 15 nodes? It is the smallest size where the phenomena we cared about become visible. Two AZs mean cross-zone gossip effects are present. A dependency graph of diameter 5 means gossip propagation is non-trivial. The service mix - stateless APIs, databases, caches, a gateway - is diverse enough to stress per-service threshold tuning. The company was not willing to send more traffic to an unproven system, which is fair enough.

We self-hosted Prometheus and Loki for telemetry. Initial sidecar allocation was 100m CPU and 128Mi RAM per agent. Fault injection used Chaos Mesh. We had to schedule the chaos experiments around a feature freeze in Week 2, which compressed the injection schedule for Weeks 3–4.

[4]The deployment was reviewed and approved by the company's platform engineering lead. Live customer traffic was routed through the experimental cluster with the same SLOs as the primary path. An automated kill switch would disable all diagnostic sidecars and reroute traffic if error rates exceeded 0.5%.

B. Rollout

We rolled out in four phases.

Week 0 (shadow mode). Agents deployed in observation-only mode. No alerts, no remediation. We used this week to tune anomaly thresholds against the existing centralized monitoring system.

Week 1 (canary). 1% traffic routed. Decentralized diagnostics ran in parallel with centralized monitoring. Automated remediation limited to pod restarts.

Weeks 2–3. Operators approved scaling and rerouting actions. PagerDuty and Slack integration activated.

Week 4 (autonomous). Full autonomy. Operators monitored dashboards but intervened only when confidence dropped below 85%.

C. Faults: Injected and Organic

We injected 112 faults across 25 categories via Chaos Mesh: CPU saturation, memory pressure, disk I/O latency, network partitions. Faults were spread across services and times of day.

Three organic incidents also occurred. We did not plan for them, obviously.

O-1: Gradual memory leak in `CartService`. Connection pool exhaustion caused p99 latency to climb from 18 ms to about 120 ms over what we estimate to have been 40–50 minutes. We initially attributed it to an unrelated promotional campaign deployment that had gone out a couple of hours earlier - the timing lined up and that deploy had touched the cart flow. In fact, the memory leak had been present since a dependency update earlier in the week. The system correctly identified `CartService`, but by the time our anomaly threshold flagged it, downstream effects were already visible in checkout and recommendations.

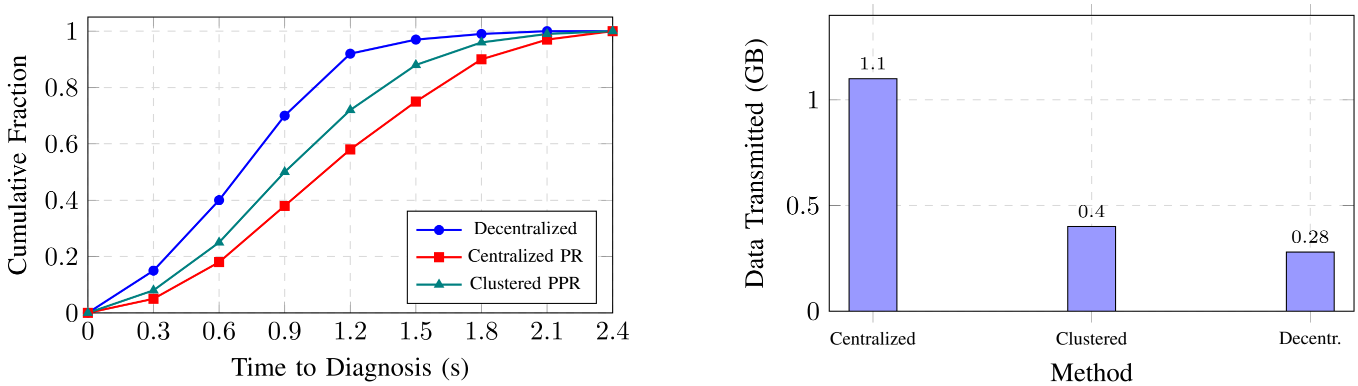
O-2: Intermittent Route 53 DNS resolution delays hitting `ProductCatalogService`. Sporadic 500 ms spikes. The system ranked `ProductCatalogService` as Top-1 - the first internal service showing symptoms - rather than the DNS infrastructure, which sits outside the agent network. The centralized baseline made the same mistake.

O-3: EBS throughput throttling during a scheduled backup. Sometime around 3 AM on a Wednesday in Week 3, `Catalogue-DB` read latency went from 12 ms to around 340 ms, propagating as intermittent timeouts to the front end. This happened during a period when all of us were not available, so the system had to handle it autonomously. The automated remediation kicked in and restarted the pod, which resolved the immediate symptoms, though the underlying EBS issue cleared on its own once the backup completed.

D. Results

Table II breaks down accuracy and timing by fault category.

Overall: 93.9% Top-1, 1.24 s MTTD, 48 s MTTR. The centralized baseline on identical telemetry gave 92.3% Top-1, 1.8 s MTTD, 85.5 s MTTR. The accuracy gap between



(a) CDF of diagnosis time across 735 benchmark cases.

(b) Network traffic for 64-service scenario.

Fig. 2: Diagnosis latency (left) and network overhead (right). The decentralized approach is roughly 32% faster on average ($p < 0.01$) and transmits about 75% less data.

TABLE II: Production results by fault category (28 days, 115 incidents).

Category	**Count**	**Top-1**	**MTTD**	**MTTR**
Resource saturation (CPU/mem)	38	97.4%	1.03 s	41 s
Network faults (loss/delay)	31	90.3%	1.82 s	63 s
Disk I/O latency	22	95.5%	1.14 s	44 s
Application-level (misconfig)	21	90.5%	1.47 s	52 s
Organic (not injected)	**3**	**33.3%**	**2.8 s**	**58 s**
All	**115**	**93.9%**	**1.24 s**	**48 s**

methods is small enough to be a reimplementation artifact. The latency and resolution-time improvements - roughly 31% and 44% - are consistent with the benchmark results.

The gap between injected and organic accuracy is the result that actually matters in this table. Injected faults: 94.6% Top-1. Organic incidents: 1 of 3 correct (O-3 only). Three incidents is not enough to draw conclusions. But the pattern is consistent with a known concern: fault injection produces clean, abrupt signatures. Organic failures develop gradually, involve external dependencies, or show up indirectly.

1) Case Study: O-3 (EBS Throttling): This incident illustrates how the gossip-based diagnosis works in practice. Week 3, sometime around 2:47 AM. A scheduled EBS backup on `Catalogue-DB` hit the IOPS limit. Read latency went from 12 ms to 340 ms over about 90 seconds.

When the latency crossed the 3× threshold, the Catalogue agent flagged a read-latency anomaly: $s = 0.91$. The FrontEnd agent saw elevated trace latency: $s = 0.62$. Orders caught a timeout pattern: $s = 0.45$. Cart - which shares the cluster but does not depend on catalog reads - reported $s = 0.15$. That low score from an unaffected neighbor matters because it keeps the PageRank computation from spreading blame everywhere.

Seven gossip rounds later (about 1.0 s), agents converged on `Catalogue-DB` with 93% confidence. The remediation controller restarted the pod. Total resolution: 41 seconds.

VI. Operational Lessons

Benchmark numbers were clean. Production was not. This section is what we learned.

Sidecar resource budgets sized from benchmarks are too low. Our initial allocation - 100m CPU, 128Mi RAM - came from benchmark profiling. In production, CPU usage climbed 40–60% during peak hours because of higher telemetry volume and more frequent anomaly recomputation. During Week 1, the `CheckoutService` sidecar got OOM-killed three times when a promotional event spiked trace volume past the 128Mi limit. We switched to a burstable model: base requests at 100m/128Mi, limits at 200m/256Mi. No OOM events after that.

Cross-AZ gossip causes split-brain diagnoses. This was the one we did not see coming. Our 15 nodes spanned two AZs. Gossip between zones ran 2–5× slower than within a zone. In 4 of 115 incidents, agents in AZ-A declared a root cause before AZ-B agents had converged. Three resolved within two extra rounds; one required six.

The subtler problem was in Week 2. A Kubernetes NetworkPolicy update - applied for a completely unrelated service by someone on the platform team who did not know about our agents - silently blocked gossip between two cross-zone agent pairs for 18 hours. No error was raised. The gossip protocol is asynchronous and does not require acknowledgments, so the affected agents simply stopped hearing from their cross-zone neighbors and nobody noticed. We found it by manually checking gossip round-trip times during a routine health check. This is exactly why we think production deployments matter - one would not encounter a stray NetworkPolicy update in a benchmark.

We added a gossip health-check daemon that alerts when any agent goes 30 seconds without a peer message, and adjusted convergence thresholds for higher variance in cross-zone delivery. The health-check caught a similar issue in Week 4 within minutes.

One anomaly threshold does not fit all services. We started with a global $\tau = 2.0$. Within 48 hours: 8.7% false-positive rate, 23 spurious alert chains, and the on-call team asked us to either fix it or they would turn the system off. We raised it to $\tau = 4.0$ before canary. That helped for compute-bound services but was still too sensitive for `Catalogue-DB` (naturally bursty I/O) and too conservative for `PaymentService` (tight SLOs where small anomalies matter).

We ended up with three tiers: $\tau = 2.0$ for payment and auth services, $\tau = 4.0$ for stateless APIs, $\tau = 6.0$ for databases. False-positive rate dropped from 2.3% to 0.8%. Getting the tiers right took about two weeks of observation and iteration. Whether this three-tier approach will hold up as we add more services is something we are not certain about - it may need to become fully per-service.

Operators do not trust ranked lists. Our initial output was a ranked list of services with scores. During Week 1, the on-call SREs re-investigated 7 of 12 correctly diagnosed incidents. They did not dispute the diagnoses - they simply had no reason to trust them. One of the principal engineers said something like "Why would I trust this over Grafana? I can see the spike right there in Grafana."

In Week 2 we added structured summaries: top-3 services with contributing signal types (e.g., "CPU anomaly: 0.82, trace latency: 0.71"), the gossip propagation path, and a Grafana deep link filtered to the incident window. Re-investigation dropped from 58% to 12%. But even by Week 4, operators rated willingness to run without the parallel centralized system at 2.8 out of 5. Trust takes longer than four weeks.

Stale graphs corrupt PageRank silently. Week 3. A canary deployment of `ShippingService` changed its upstream from `ProductCatalogService` to a new `InventoryService`. Our agents used the old graph for 12 minutes - the trace-based refresh was every 15 minutes. During that window, a fault in `InventoryService` was attributed to `ProductCatalogService`. The stale P matrix sent ranking mass along an edge that no longer existed. The diagnosis converged normally and looked confident, which is the worst part - nothing in the output indicated the graph was wrong.

We shortened the refresh to 2 minutes and added a Kubernetes watch on Deployment and Service events for immediate graph recomputation.

Log volume from agents cluttered the logging pipeline. This is a boring one but it took most of a day. The diagnostic agents produce a lot of debug logging - gossip round summaries, anomaly score updates, convergence status. During Week 1, this roughly doubled the log volume flowing into the shared Loki instance, and the ops team asked us to route agent logs to a separate stream. We set up a dedicated Loki tenant for diagnostic agent logs. Not architecturally interesting, but it was blocking the SRE team from approving the Week 2 rollout until we fixed it.

External dependencies are invisible. O-2 showed this. Route 53, AWS APIs, third-party gateways - all outside the agent network. We observed this problem but only partially addressed it before the deployment ended. We added lightweight sentinel agents that poll external endpoints and inject synthetic anomaly scores into the gossip network when degradation is detected. A re-injected DNS delay test after deploying sentinels was correctly attributed. But the sentinels only cover endpoints we thought to configure, and we ran out of deployment time before testing how well they work for truly unexpected external failures.

VII. Discussion and Limitations

The production results confirm the benchmark finding: decentralized and centralized PPR reach the same answer. The operational advantage - lower latency, less traffic, no single point of failure in the diagnostic path - is real but modest at 15 nodes. Whether the latency advantage becomes compelling at 200+ services is an open question.

The deployment is small. Fifteen nodes, 140 QPS. Synthetic scaling results are encouraging but synthetic. Three organic incidents is nowhere near enough for reliable conclusions about the organic-vs-injected accuracy gap. All baselines are reimplementations; small accuracy differences may reflect our code, not the algorithms.

The gossip protocol assumes regular message exchange and a connected dependency graph. Network partitions can isolate agent subsets, as the cross-AZ episodes demonstrated. The system does not handle Byzantine agents - a compromised sidecar could inject false scores. We have not addressed multiple simultaneous root causes; the PageRank model implicitly favors a single dominant fault.

VIII. Conclusion

We built a decentralized RCA system where per-service diagnostic agents run distributed PageRank through gossip, with no central coordinator and no centralized telemetry collection. Accuracy matches centralized PageRank on benchmarks. The latency and traffic improvements are consistent across benchmark and production settings.

The production deployment was the point. Over four weeks on a live cluster, the system handled injected faults well but struggled with organic incidents - the kind that develop gradually, involve external services, or do not produce clean anomaly signatures. Cross-AZ gossip turned out to be more fragile than we expected, stale dependency graphs corrupt diagnoses silently, and operators need evidence chains rather than ranked lists before they will trust a new system.

A 200+ node deployment with 5% of traffic is planned for next month. The main question is whether hierarchical gossip – intra-AZ convergence followed by cross-AZ summary exchange – actually fixes the split-brain problem or just moves it to a different layer. We do not yet have an answer.

Disclosure

Generative AI tools were used solely for language editing and organization. All technical content, experiments, and analysis are the author's own work. This research was conducted independently and does not represent the views of the author's employers and university.

REFERENCES

[1] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, “MicroRCA: Root Cause Localization of Performance Issues in Microservices,” in *Proc. IEEE/IFIP NOMS*, 2020, pp. 1–9.

[2] P. Naik, S. Goel, E. Raichstein, C. Govindarajan, and P. Jayachandran, “Enabling Multi Cloud Observability,” in *Proc. COMSNETS (Demos & Exhibits)*, 2024, pp. 279–281.

[3] A. Keller and J. Whitson, “Achieving Observability at Scale through Federated Learning,” in *Proc. IEEE/IFIP NOMS*, 2023, pp. 1–6, doi: 10.1109/NOMS56928.2023.10154346.

[4] P. Notaro, J. Cardoso, and M. Gerndt, “A Survey of AIOps Methods for Failure Management,” *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 6, pp. 1–45, 2021.

[5] S. Gu *et al.*, “TrinityRCL: Multi-Granular and Code-Level Root Cause Localization Using Multiple Types of Telemetry Data in Microservice Systems,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 5, pp. 3071–3088, 2023.

[6] D. Fernando, M. A. Rodriguez, and R. Buyya, “A Decentralized Root Cause Localization Approach for Edge Computing Environments,” *arXiv preprint arXiv:2511.12486*, 2025.

[7] Y. Wang, Z. Zhu, Q. Fu, Y. Ma, and P. He, “MRCA: Metric-level Root Cause Analysis for Microservices via Multi-Modal Data,” in *Proc. ASE*, 2024, pp. 1057–1068.

[8] Z. Zhu, C. Lee, X. Tang, and P. He, “HeMiRCA: Fine-Grained Root Cause Analysis for Microservices with Heterogeneous Data Sources,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–25, 2024.

[9] T. Ahmed *et al.*, “Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models,” in *Proc. ICSE*, 2023.

[10] Z. Wang *et al.*, “RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models,” in *Proc. CIKM*, 2024, pp. 4966–4974.

[11] Y. Zhang *et al.*, “CloudRCA: A Root Cause Analysis Framework for Cloud Computing Platforms,” in *Proc. ACM CIKM*, 2021, pp. 4373–4382.

[12] A. Brandón *et al.*, “Graph-based root cause analysis for service-oriented and microservice architectures,” *J. Syst. Softw.*, vol. 159, Art. no. 110432, 2020.

[13] W. Zhang *et al.*, “mABC: Multi-Agent Blockchain-inspired Collaboration for Root Cause Analysis in Micro-Services Architecture,” in *Findings EMNLP*, 2024, pp. 4017–4033.

[14] L. Pham, H. Zhang, H. Ha, F. Salim, and X. Zhang, “RCAEval: A Benchmark for Root Cause Analysis of Microservice Systems with Telemetry Data,” in *Companion Proc. WWW*, 2025, pp. 777–780.