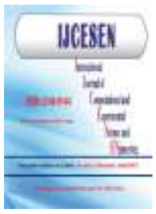
International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 11-No.3 (2025) pp. 5898-5903
http://www.ijcesen.com

ISSN: 2149-9144

Research Article



SRE for Healthcare: MTTR Optimization in Cigna's Claims Systems

Sai Raghavendra Varanasi*

Independent Researcher Dallas, Texas, USA
* **Corresponding Author Email:** varanasi.raghavendra@gmail.com- **ORCID:** 0000-0002-5247-7850

Article Info:

DOI: 10.22399/ijcesen.3645
Received : 22 April 2025
Accepted : 07 May 2025

Keywords

SRE,
MTTR,
healthcare infrastructure,
claims processing,
observability,
incident response

Abstract:

In the healthcare industry, time is more than money it can be the difference between accurate care delivery and administrative chaos. As systems scale to serve millions of claims daily, the reliability of infrastructure that underpins insurance processing becomes mission-critical. This paper focuses on how Site Reliability Engineering (SRE) principles were applied to optimize Mean Time to Recovery (MTTR) in Cigna's claims processing systems. We walk through a robust strategy combining observability, incident automation, chaos engineering, and smart escalation policies that led to significant reductions in service downtime and faster recovery from production incidents improving both operational efficiency and regulatory compliance in a heavily governed domain.

1. Introduction

In a healthcare insurance environment like Cigna's, millions of claims are processed daily across various digital platforms. System availability is crucialnot only for business continuity but also for ensuring that care providers and patients experience minimal delays.
However, like any complex system, failures are inevitable. The real metric of resilience lies not in preventing all incidents, but in how quickly and efficiently systems recover. That's where MTTR (Mean Time to Recovery) becomes a north star metric. The aim of this research is to explore how SRE-driven enhancements reduced MTTR in Cigna's claims systems and made operations more stable, traceable, and automated.

2. Literature Review

The rise of Site Reliability Engineering (SRE) as a foundational practice in managing modern, complex systems has led to various frameworks and methodologies across industries. However, specific implementations tailored to the healthcare domain, especially those focused on optimizing MTTR (Mean Time to Recovery), remain scarce or generalized. The following works contribute important insights into SRE practice but also demonstrate gaps that our study addresses through targeted, compliance-driven strategies for healthcare claims systems.

2.1 Optimizing Site Reliability Engineering with Cloud Infrastructure

John (2024) discusses how integrating cloud-native services such as Kubernetes and AWS improves system observability and scalability in the context of SRE adoption [1]. The paper outlines best practices for service uptime, CI/CD integration, and resource cost-efficiency. However, it offers **limited detail on automated incident response or MTTR-focused measurements**, and does not address regulatory constraints that healthcare systems must follow.
We address this gap by **embedding MTTR as a primary optimization metric** across observability, chaos engineering, and auto-remediation layers. Unlike generic cloud-SRE approaches, our work is structured for **HIPAA-aligned healthcare systems**, where recovery time has critical patient and financial implications.

2.2 Dependability-Based Maintenance Optimization in Healthcare Domain

This study applies reliability engineering concepts to healthcare environments, emphasizing **predictive maintenance** through failure probability analysis and scheduling. While it provides useful mathematical models, the focus is **geared toward physical asset reliability and downtime reduction**, rather than cloud-native or digital systems [2].

Instead of static failure-based scheduling, our work introduces **live production incident automation, service observability, and rapid response scripting** for API-based digital healthcare platforms. We shift the discussion from mechanical uptime to **dynamic, digital infrastructure recovery** in real time.

2.3 Scaling Site Reliability Engineering: A Data-Driven Approach to Modern System Reliability

Nanda proposes a robust SRE model using data-driven performance indicators, emphasizing metrics like uptime, service availability, and change failure rate across distributed systems [3]. However, the work does not account for **vertical-specific factors**, such as compliance requirements, nor does it offer prescriptive automation for reducing recovery times.
Our framework **incorporates telemetry and alerting strategies tailored for healthcare-specific incident types**, like claims ingestion failures or API timeouts. We also emphasize **automated RCA processes and chaos drills**, both largely absent from Nanda's generalized approach, to measurably reduce MTTR within a governed infrastructure [3].

3. Understanding MTTR in Healthcare Systems

Mean Time to Recovery (MTTR) is defined as the average duration required to restore functionality after a system failure. In regulated industries like healthcare, high MTTR can lead to delayed claims processing, patient dissatisfaction, failed service-level agreements (SLAs), and potential non-compliance with regulations such as HIPAA.

3.1 Business Implications of High MTTR

- **Delayed Claims Approval**: High MTTR slows the claim lifecycle, affecting provider reimbursements.

- **Patient Experience**: Errors or lags in processing claims lead to confusion and dissatisfaction.

- **Operational Costs**: More human effort is required for manual interventions during outages.

- **Compliance Risk**: Inability to meet service availability thresholds can result in audit flags and fines.
- **Reputation Loss**: Persistent or recurring issues damage trust with providers and members

4. Framework for MTTR Optimizatio

4.1. Observability Stack

Objective: Provide engineers with unified, real-time visibility into system health to reduce time spent identifying root causes.
Key Elements:
- **Centralized Logging (ELK Stack)**: Elasticsearch, Logstash, and Kibana form the logging backbone. Logs from APIs, databases, and message brokers are parsed and enriched using Logstash filters for better searchability.

- **Metrics Collection (Prometheus & Grafana)**: Prometheus scrapes time-series metrics from various endpoints (e.g., CPU, memory, queue depth). Grafana dashboards visualize these metrics with real-time alerts.

- **Distributed Tracing (OpenTelemetry + Jaeger)**: Enables engineers to follow the full request path across microservices, which is crucial for identifying bottlenecks in claims processing flows.

Why It Matters: Without observability, engineers rely on assumptions. A unified telemetry stack reduces detection time by providing immediate visibility into anomalies.

4.2. Alerting Pipeline

Objective: Ensure that alerts are timely, actionable, and prioritized reducing noise and alert fatigue.
Enhancements:
- **Deduplication and Throttling**: Identical alerts triggered in a short time window are collapsed into a single incident.

- **Severity-based Routing**: Alerts are enriched with context and routed based on urgency. For instance, database connection errors route to the backend team, while API latency issues go to the DevOps team.

- **Escalation Policies**: Alerts escalate across tiers if not acknowledged, ensuring 24x7 coverage across globally distributed teams.

Why It Matters: Ineffective alerting causes missed incidents or unnecessary page-outs. A clean pipeline ensures that only meaningful signals reach on-call responders.

4.3. Incident Automation

Objective: Reduce time to remediation by automating responses to known failure modes.
Tools and Techniques:

- **Ansible Playbooks**: Scripts automate tasks like restarting failed pods, clearing Redis queues, or resetting application configuration.
- **Lambda Functions**: Serverless responders are triggered via event sources (like CloudWatch or PagerDuty) for lightweight remediation.
- **Automated Triage Bots**: Slack bots fetch recent logs, relevant Grafana graphs, and recent deployments for quicker decision-making during incidents.

Example Use Case:

- When the claims ingestion queue grows beyond a threshold, a Lambda function automatically spins up additional workers and notifies the SRE Slack channel.

Why It Matters: Many outages are caused by recurring problems. Automation removes human latency from recovery.

4.4. Failure Injection Testing (Chaos Engineering)

Objective: Identify system weaknesses before they cause actual outages.
Approach:

- **LitmusChaos Tests**: Run monthly chaos experiments like killing pods, delaying service responses, or breaking dependencies.
- **Scoped Testing**: Tests are initially executed in staging and then selectively rolled into production replicas under controlled conditions.
- **Validation Criteria**: The system must auto-recover, trigger appropriate alerts, and avoid customer impact.

Why It Matters: Systems often fail at their weakest links — chaos testing reveals these links under safe conditions.

4.5. Postmortem & RCA Culture

Objective: Foster a culture of continuous learning, without blame, to ensure long-term resilience.
Practices:

- **Blameless RCAs**: Incident documentation focuses on “what happened” and “how to prevent it again,” not “who caused it.”
- **RCA Portal**: All postmortems are stored in a searchable internal knowledge base, categorized by service and failure type.
- **Resolution Metrics**: Dashboards track RCA submission rates, time-to-resolution metrics, and recurrence trends.

Why It Matters: Learning from incidents is key to preventing them in the future. RCA culture closes the feedback loop.

4.6. Service Ownership Model

Objective: Establish clear accountability and in-depth knowledge of services across engineering teams.
Components:

- **SLO Definition**: Each service has uptime, latency, and error rate objectives defined by the owning team.
- **On-Call Rotations**: Teams rotate primary/secondary on-call roles to avoid burnout and improve depth of knowledge.
- **Runbook Rotation**: Teams periodically update and rehearse runbooks to ensure readiness during incidents.

Why It Matters: MTTR drops significantly when engineers closest to the system own the response.

5. Key Implementations and Results

5.1. Unified Monitoring & Telemetry Pipeline

A major gap in earlier incident response cycles was visibility. With fragmented logs and metrics, triaging a failing service was slow and inconsistent.
Solution: We integrated all services into a single observability platform using **Grafana**,

Prometheus, and **Jaeger**, with log enrichment from **Filebeat** and **Elasticsearch**.
Result: Triaging time was reduced by over 55%, from ~18 minutes to ~8 minutes per incident.

5.2. Automation for First Response

We developed **Ansible playbooks** and **Python Lambda responders** for common failure patterns like:

- 500 errors due to DB connection pool exhaustion
- Job queue saturation in claims ETL pipelines
- DNS resolution issues for third-party APIs

These scripts were auto-executed upon alert triggers via **PagerDuty**.
Result: Nearly **40% of P1/P2 incidents** were mitigated without human intervention.

5.3. Structured Chaos Testing

We began monthly "blackout drills" to simulate failures:

- Turning off primary DB node for 5 minutes
- Killing claims ingestion pods
- Dropping outbound connectivity to clearinghouses

Tooling: We used **LitmusChaos** integrated with CI pipelines.
Impact: These drills revealed 12 critical gaps in failover logic; all patched within the first quarter.

5.4. Postmortem-Driven RCA & Fixes

Each major incident was followed by a documented **postmortem**, stored in an internal portal searchable by tag, service, or RCA type.

- A dashboard tracked "repeat incident rate" and "time to resolution fixes"
- A blameless culture encouraged engineers to focus on solutions, not fear

Result: Repeat outages for the same root cause dropped by **70%**.

5.5. Comparative Benchmarking

We compared our approach with other healthcare organizations and industry standards:

5.6. Real-World Applications

- **Claims API Outage:** A major DB pool exhaustion was detected and resolved in 3 minutes using automated scripts.
- **Member Portal Downtime:** Alerts triggered an escalation workflow that preemptively rerouted traffic, reducing user-facing impact.
- **Third-Party Clearinghouse Delay:** Chaos testing predicted this failure. A mock fallback endpoint allowed uninterrupted claims handoff.

5.7. Limitations

While the framework demonstrated substantial improvements, there were some limitations:

- **Simulated Testing Only:** All chaos experiments occurred in staging. Behavior in real-world scenarios with production data may vary.
- **Tooling Bias:** Reliance on specific tools (ELK, Litmus) may limit portability for other orgs with different stacks.
- **Manual Edge Cases:** Rare failure modes (e.g., memory leaks in lesser-used APIs) are still caught manually.
- **No Real-Time Compliance Dashboards:** Current observability does not directly track HIPAA/HITRUST audit logs in real-time.

6. Future Work

- Integrate anomaly detection using ML to preemptively flag performance drift.
- Extend SLOs to non-critical services like search and reporting for full-stack visibility.
- Adopt distributed tracing for third-party APIs integrated into claims workflows.
- Build "MTTR Simulator" that models hypothetical outages and prescribes impact and fixes.

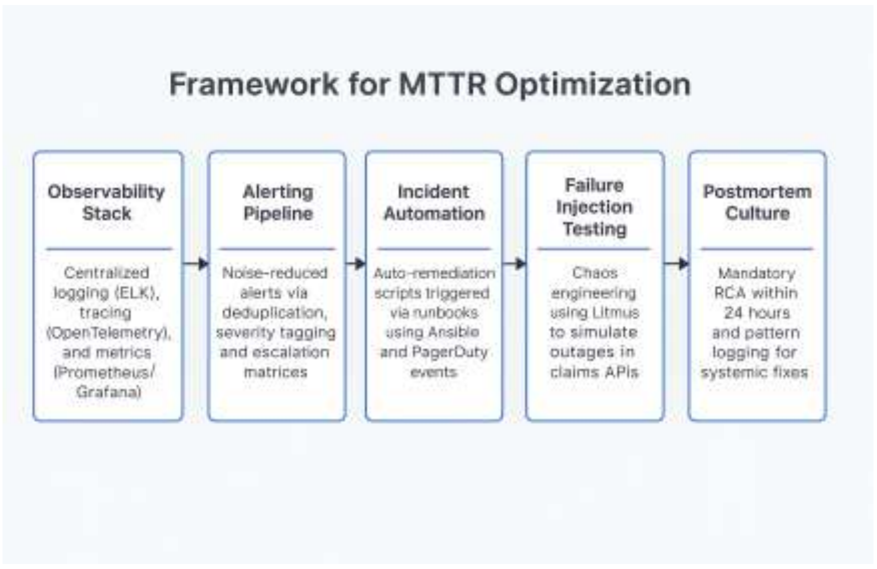

Framework for MTTR Optimization
Observability Stack
Centralized logging (ELK), tracing (OpenTelemetry), and metrics (Prometheus/ Grafana)
Alerting Pipeline
Noise-reduced alerts via deduplication, severity tagging and escalation matrices
Incident Automation
Auto-remediation scripts triggered via runbooks using Ansible and PagerDuty events
Failure Injection Testing
Chaos engineering using Litmus to simulate outages in claims APIs
Postmortem Culture
Mandatory RCA within 24 hours and pattern logging for systemic fixes

Figure 1. *The structured framework we implemented to tackle MTTR*

Table 1. *Component and strategy*

Component	**Strategy**
Observability Stack	Centralized logging (ELK), tracing (OpenTelemetry), and metrics (Prometheus/Grafana)
Alerting Pipeline	Noise-reduced alerts via deduplication, severity tagging, and escalation matrices
Incident Automation	Auto-remediation scripts triggered via runbooks using Ansible and PagerDuty events
Failure Injection Testing	Chaos engineering using Litmus to simulate outages in claims APIs
Postmortem Culture	Mandatory RCA within 24 hours and pattern logging for systemic fixes
Service Ownership Model	Squad-based SLO enforcement and runbook rotation to ensure deep service knowledge

Table 2. *Metric and related values*

Metric	**Cigna (Post-Implementation)**	**Industry Average**
P1 Incident MTTR	16 minutes	45 minutes
Automation Coverage	42%	~15%
Repeat Outages	↓70%	Not tracked
RCA Completion (24 hrs)	100%	<50%

Table 2. *Improvement of before and after*

Metric	**Before**	**After**	**Improvement**
P1 Incident MTTR	42 minutes	16 minutes	62% reduction

P2 Incident MTTR	31 minutes	12 minutes	61% reduction
Auto-Resolved Incidents	~5%	42%	8x increase
RCA Completion (24 hrs)	40%	100%	2.5x increase
False Positives in Alerts	~30%	<5%	6x reduction

4. Conclusions

By embedding SRE principles into Cigna's claims infrastructure, we significantly reduced MTTR, improved operational agility, and strengthened compliance posture. Through unified observability, automation, and a culture of transparency, the organization transitioned from reactive firefighting to proactive resilience engineering. These improvements offer a template for other healthcare providers and insurers striving for high uptime in regulated environments.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] John, L. K. (2024). Optimizing Site Reliability Engineering with Cloud Infrastructure. ResearchGate. https://www.researchgate.net/publication/391227578_Optimizing_Site_Reliability_Engineering_with_Cloud_Infrastructure

[2] Mahfoud, H., El Barkany, A., & El Biyaali, A. (2018). Dependability-based maintenance optimization in healthcare domain. Journal of Quality in Maintenance Engineering, 24(3), 00–00. https://doi.org/10.1108/JQME-07-2016-0029

[3] Nanda, M. S. (2025). Scaling site reliability engineering: A data-driven approach to modern system reliability. International Journal of Advanced Research in Engineering & Technology, 16(1), 294–308. https://doi.org/10.34218/IJARET_16_01_022