International Journal of Management, IT & Engineering
Vol. 15 Issue 05, May 2025,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Message Observability in High-Stakes Notification Systems

Seema B Shrikant

Abstract

Real-time notification systems such as Email, SMS, and Webhooks play a critical role in delivering transactional messages in high-stakes industries like finance and healthcare. Traditional monitoring tools often rely on infrastructure-level metrics or basic delivery confirmations, which are insufficient for detecting message-level delivery failures, retries, or drops. This paper presents a product-led approach to observability in notification systems, focusing on end-to-end message traceability, anomaly detection, and cross-functional visibility. Drawing from industry practices and foundational tracing models such as Google's Dapper [1], this approach supports incident resolution, product decision-making, and client communication. The proposed model emphasizes observability as a shared operational capability that extends beyond backend teams and enables reliable communication across user-critical touchpoints.

Copyright © 2025 International Journals of Multidisciplinary Research Academy. *All rights reserved.*

Keywords:

Observability;
Notification Systems;
Retry Policy;
Delivery Reliability;
Product-Led Monitoring

Author correspondence:

Seema B Shrikant,

Product Manager, Visa Inc., Austin-TX, USA

Email: seemabshrikant@gmail.com

1. Introduction

Notification systems are integral to real-time user communication in sectors that demand both speed and reliability. From transaction alerts and fraud warnings to onboarding workflows and password resets, these messages often carry operational or regulatory significance. However, many organizations still rely on simple indicators such as HTTP success codes or application-level logs to determine delivery success. These limited signals obscure issues such as vendor-side SMS drops, email bounces, and webhook retries that silently fail. Without deeper visibility, failures may only become apparent after customer complaints or SLA violations.

In modern platforms, notification delivery is often dependent on third-party vendors and asynchronous queuing mechanisms. This makes message observability a non-trivial challenge. A message may appear successfully dispatched from the application layer but still fail due to an external provider rejection, filtering policies, or queue-level drop. Even when logs are available, they are typically siloed across systems, making holistic analysis time-consuming and error-prone.

Recent advances in distributed tracing and platform observability have enabled engineering teams to monitor application behavior more granularly [1]. Yet, these practices are rarely extended to the message layer in notification systems. While infrastructure monitoring tools like Prometheus and Grafana, or commercial suites like Datadog and Splunk, offer rich metrics, they often lack business-level context or usability for cross-functional teams. Without the ability to drill into delivery outcomes by client, use case, or channel, product and support teams are left without meaningful insights. This paper proposes a structured observability model that shifts ownership from infrastructure to product and support teams, enabling full message lifecycle tracking and decision-making grounded in delivery performance.

2. Theoretical Basis and Proposed Method

Much of the foundational thinking around observability in distributed systems stems from the introduction of Dapper, Google's internal tracing framework [1]. It introduced the idea of tracking user requests across multiple services through trace IDs, enabling performance debugging at scale. Dapper has since inspired tools such as Zipkin, Jaeger, and the OpenTelemetry standard.

OpenTelemetry provides a vendor-neutral framework for collecting and correlating logs, metrics, and traces, but it requires technical implementation and lacks business-layer abstractions. In notification systems, messages often traverse external vendor APIs, asynchronous queues, and retry mechanisms that span systems outside of direct developer control. Monitoring such systems purely at the infrastructure level is insufficient.

Turnbull [2] emphasizes the importance of full-stack observability, which includes not only the infrastructure and application layers but also user experience and operational workflows. However, practical adoption often stops at engineering boundaries. The model introduced in this paper applies these concepts to real-time messaging systems by integrating delivery tracking with decision-making roles across engineering, product, and support.

The proposed model consists of four interlinked components designed to improve observability in asynchronous, multi-channel notification systems. These include message traceability, role-aware visibility, anomaly detection, and retry pattern analysis. Each of these components contributes to building a more resilient and accountable messaging infrastructure that supports user-critical operations.

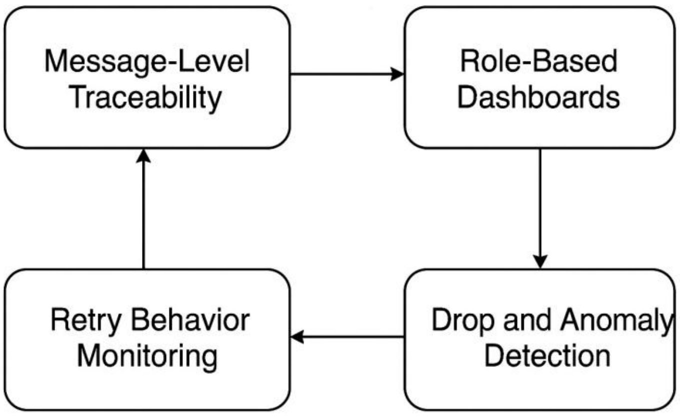
Message-Level Traceability
Role-Based Dashboards
Retry Behavior Monitoring
Drop and Anomaly Detection

Figure 1: *Framework Overview Diagram*

3. Research Method

This section details how each component of the proposed observability framework was implemented within a high-scale financial platform. The objective was to assess whether a product-led observability model could meaningfully improve messaging reliability, reduce support dependencies, and enhance cross- functional accountability. The platform serves global enterprise clients and delivers millions of messages per month across Email, SMS, and Webhook channels.

Rather than overhauling infrastructure, the implementation focused on layering observability on top of existing delivery workflows. Each pillar—traceability, dashboards, anomaly detection, and retry intelligence—was approached as a discrete track. These efforts were evaluated not only for technical soundness but also for their impact on operational transparency and team workflows.

3.1 Message-Level Traceability

At the core of the observability framework is the ability to track each individual message throughout its delivery lifecycle. To accomplish this, a globally unique identifier was assigned to every outbound message at the time of event generation. This ID was consistently passed through all downstream systems—including internal queues, orchestration services, and third-party delivery vendors—to maintain continuity of traceability.

Alongside the identifier, a structured payload of metadata was captured at each transition point. This included timestamps for message creation, queue entry, vendor dispatch, and final delivery or failure confirmation. Delivery status codes, retry attempt counts, and error responses were also logged. In some cases, the platform captured vendor-specific bounce reasons or blocklist indicators, enriching the dataset for downstream analysis.

Rather than relying on monolithic log files, the data was ingested into a centralized telemetry system designed for query ability by both technical and non-technical users. This ensured that any individual message could be located and inspected within seconds, even across asynchronous flows.

Over time, this message-level traceability became a critical tool not just for diagnosing incidents, but also for training support teams, validating system updates, and identifying long-tail edge cases. It served as the foundation for the rest of the observability framework.

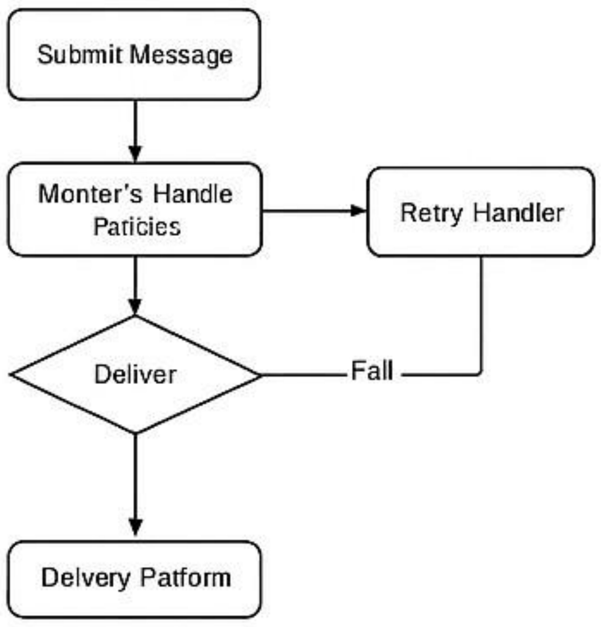

Figure 2: *Message Lifecycle Diagram*

3.2 Role-Based Observability Dashboards

Once message-level traceability was established, the next priority was making that data actionable across different user groups. Dashboards were designed with tailored views for engineering, support, and product teams—each with distinct access needs, technical fluency, and decision-making responsibilities.

For support teams, the focus was on fast, intuitive search capabilities. They could enter a message ID, client organization name, or customer account number to retrieve the full delivery history of a message, including error reasons and retry attempts. These dashboards often surfaced only the most critical fields—delivery status, timestamp, channel used, and resolution outcome—so that support agents could triage tickets quickly without engineering intervention.

Product managers used delivery health dashboards to monitor performance across message types and use cases. For example, they could filter messages tied to account onboarding workflows and compare success rates across geographies or vendors. This allowed them to make data-informed decisions on feature prioritization and client communication strategies.

Engineering teams accessed more detailed dashboards and logs through Grafana and Kibana interfaces connected to the telemetry backend. These dashboards supported deeper analysis, including

filtering by HTTP response codes, vendor error messages, and latency metrics over time. Engineers used these insights to optimize retry logic, identify systemic patterns, and validate fixes.
By exposing a shared observability layer that respected each team's operational context, the platform avoided tool fragmentation and fostered a sense of joint ownership over delivery reliability.

3.3 Drop and Anomaly Detection

A core capability of the observability framework was the ability to proactively surface delivery failures before they became client-visible issues. Unlike infrastructure monitoring that detects server or CPU outages, message anomaly detection focuses on identifying unusual behavior in delivery outcomes—such as sudden spikes in bounces, delays, or retries.

Each delivery channel had its own thresholds for what constituted "normal" versus anomalous behavior. For example, bounce rates above 2% for email messages or timeouts exceeding 5 seconds for Webhook responses were flagged for review. These thresholds were refined over time based on historical trends and message criticality. Time-sensitive alerts (e.g., OTPs) were held to stricter standards than bulk marketing messages.

Anomaly detection relied on structured logs collected via the telemetry system and compared them in near-real-time to moving averages and historical baselines. Alerts were generated when deviations exceeded acceptable bounds, either globally or for specific vendors, regions, or clients.
For example, an unanticipated rise in SMS delivery failures in a specific geographic region prompted a deeper investigation that uncovered a vendor-level outage. In another case, increasing Webhook failures for a major client were traced to their internal timeout policy changes.

By implementing channel-specific anomaly detection and coupling it with traceability data, teams could respond to degradation events faster, with greater precision and fewer escalations. This not only improved delivery reliability but also strengthened operational relationships with third-party vendors.

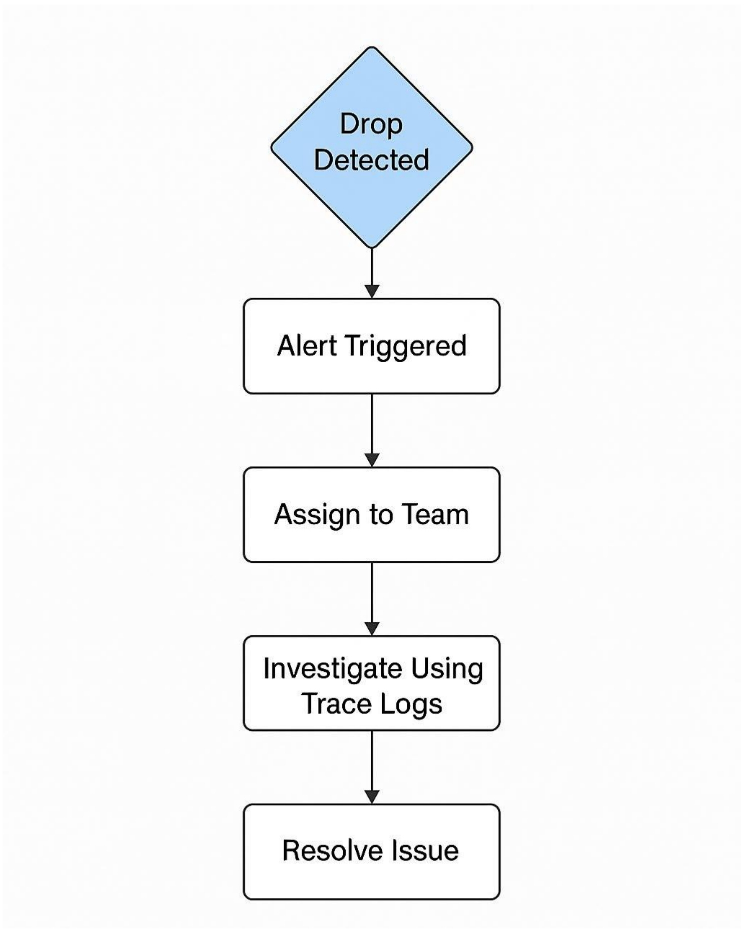

Figure 3: *High-level flow for delivery anomaly alerts*

3.4 Retry Behavior Monitoring

Message retries are a critical component of resilient notification systems, especially in environments that depend on external APIs and asynchronous processing. However, overly aggressive retry policies can cause system load issues, while overly conservative ones can compromise delivery reliability. Observability into retry behavior allowed the platform to strike the right balance.

Each message sent through the platform included metadata fields capturing retry count, retry intervals, and final delivery status. This data was aggregated over time and segmented by message type, delivery channel, and use case. Trends such as frequent success after a single retry or consistent failures after a certain threshold helped teams fine-tune retry configurations.

For example, time-sensitive messages like one-time passwords (OTPs) exhibited high success rates when retried within the first 2 minutes. In contrast, bulk messages such as transactional receipts saw little gain after the second retry, suggesting an opportunity to cap retries and reduce processing overhead.

The observability framework supported both arithmetic and geometric retry strategies and captured the effectiveness of each. Visualization tools allowed engineers to compare retry success curves and evaluate retry fatigue—cases where repeated attempts provided diminishing returns. In one pilot, shifting to a backoff strategy for non-critical messages reduced vendor rate-limit errors and improved platform throughput during peak hours.

By continuously analyzing retry patterns and outcomes, the platform established a more data-driven approach to reliability. This helped reduce delivery-related resource costs while preserving the responsiveness expected in mission-critical flows.

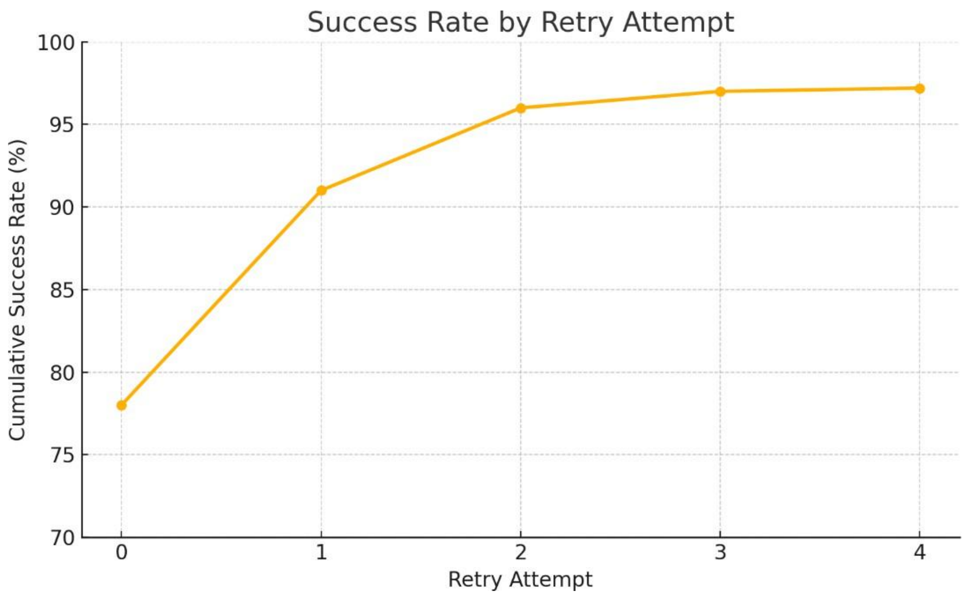

Figure 4: *Success rate by retry attempt*

This chart illustrates the cumulative percentage of messages successfully delivered after each retry attempt. The majority of messages succeeded on the first attempt or within the first retry, with only marginal gains beyond the second attempt. This finding informed the platform’s decision to adopt tiered retry policies—reserving aggressive retries for high-priority messages and reducing unnecessary load for non- critical traffic.

4. Results and Discussion

4.1 Faster Incident Resolution

The most immediate impact of implementing message-level traceability was a measurable reduction in the time it took to resolve delivery-related incidents. Prior to the framework, incident resolution often required coordination between multiple teams, manual log mining across disparate systems, and guesswork based on limited metadata. Engineering teams frequently had to reconstruct the message path from fragmented logs, while support teams lacked the access needed to triage issues independently.

After the framework was implemented, messages could be searched by ID, event type, or client account. Logs captured each transition point with consistent identifiers, and delivery metadata—such as vendor status codes, timestamps, and retry results—was easily retrievable in real time. This allowed engineering teams to isolate failure points quickly, often without needing to escalate across systems.

In one instance, a critical production incident involving a delay in sending password reset emails was resolved in under 30 minutes using the traceability dashboard. Previously, such an issue might have taken hours to debug, often requiring cross-team calls and speculative fixes. With clear visibility into where the messages stalled—in this case, a stuck queue due to a malformed vendor response—the engineering team was able to restore flow promptly and confidently communicate resolution details to stakeholders.

This reduction in time-to-resolution not only improved system reliability but also built internal trust in the platform's observability tooling. Over time, incident retrospectives increasingly referenced dashboard usage and trace logs as first-line diagnostics, reducing the reliance on anecdotal debugging.

4.2 Reduced Support Ticket Volume

Before the implementation of the observability framework, a significant portion of support tickets related to notification delivery failures required investigation by engineering teams. Clients would often report that a message had not been received, but support agents lacked the tooling or data to confirm whether the message had been sent, retried, bounced, or dropped. As a result, even routine queries—such as confirming a Webhook delivery or checking for an SMS bounce—had to be escalated, creating bottlenecks and increasing mean time to resolution.

Following the rollout of role-based dashboards, support teams gained direct access to the message trace data they needed to triage delivery-related issues. They could now search by client ID, event type, or time range and immediately view the status of all associated messages, including vendor error messages and retry behavior. In many cases, they were able to identify and communicate root causes without requiring engineering intervention.

This shift resulted in a noticeable drop in delivery-related support tickets being escalated beyond Tier 1. Instead of submitting Jira tickets or Slack pings to backend teams, support agents were empowered to investigate and close tickets independently, often within minutes. While the overall number of client queries remained steady, the percentage requiring engineering involvement declined significantly.

Moreover, the support team's improved confidence in the data led to more accurate and consistent communication with clients. Rather than relying on speculative language ("engineering is investigating"), agents could provide concrete timelines and explanations backed by trace logs. This reduced client frustration and increased satisfaction during high-sensitivity events, such as fraud alerts or account activity notifications.

4.3 Product Insights and Roadmap Impact

Beyond technical diagnostics, the observability framework provided product managers with a new class of delivery intelligence—actionable data that previously required engineering queries or manual synthesis. By exposing aggregate delivery success rates, retry counts, bounce trends, and vendor-specific anomalies through configurable dashboards, the platform enabled product teams to make faster, evidence-based decisions about product behavior and prioritization.

One of the most significant shifts was in how the team evaluated and deprecated underperforming message flows. For example, a multi-step communication sequence tied to a lower-priority informational update consistently showed lower success rates and high retry volumes in a specific geography. Because this insight was surfaced directly in the product-facing dashboard, the team was able to validate the degradation over time and reprioritize the initiative. In this case, the use case was modified to reduce message frequency and shifted to a different channel with more reliable delivery metrics.

Delivery observability also helped product teams set clearer expectations with clients during onboarding. By visualizing baseline delivery rates for different event types, channels, or regions, they could better communicate expected performance and steer clients toward configurations with the highest reliability. This reduced back-and-forth and post-implementation surprises, particularly with high-value enterprise clients.

Moreover, during quarterly planning, delivery trend data became part of the prioritization rubric. Issues that previously would have been considered anecdotal—such as slow Webhook response times or inconsistent SMS delivery to certain carriers—were now backed by quantifiable data. This enabled teams to advocate for platform investments with confidence and clarity.

4.4 Retry Strategy Optimization

Retry logic plays a critical role in notification reliability, especially in systems reliant on external delivery providers. However, without visibility into actual retry performance, most systems adopt static retry policies—applying uniform intervals and retry limits across all message types regardless of urgency, user impact, or cost. Prior to implementing observability, this was the case for the platform as well.

Once retry behavior was captured and visualized as part of the observability framework, patterns quickly emerged. Most critical messages, such as one-time passwords (OTPs) or fraud alerts, succeeded within the first retry attempt. Non-critical messages, on the other hand—such as balance summaries or marketing receipts—often experienced diminishing returns after the second or third attempt, with retries providing little added value.

These insights informed the platform’s shift from a single global retry policy to a tiered retry strategy. Messages were categorized based on criticality, with high-priority notifications receiving shorter retry intervals and extended retry windows, while informational messages were capped at fewer retries with increased spacing between attempts. This adjustment helped reduce unnecessary load on both internal systems and third-party vendors during peak periods, while ensuring that high-urgency communications had the best chance of timely delivery.

Engineers also used retry analytics to detect systemic patterns such as “retry fatigue”—scenarios where repeated retries failed due to incorrect configurations, like expired tokens or blocked endpoints. With better visibility, these cases could be corrected proactively rather than through support escalation.

The tiered retry model, supported by observability data, not only improved overall success rates but also allowed teams to fine-tune the balance between reliability and cost. This approach aligned retry behavior with user impact and system efficiency—a step forward from traditional retry heuristics.

4.5 Business and User Impact

The technical benefits of the observability framework translated into tangible business value and improvements in end-user experience. While observability is often positioned as a backend capability, its real power emerged when it was integrated into operational workflows and decision-making across teams.

From a business standpoint, improved observability directly reduced operational risk. Notifications such as fraud alerts, transaction confirmations, and password reset emails are often time-sensitive and security-critical. Failures in these channels can lead to customer churn, reputational damage, and compliance concerns. By identifying and resolving delivery issues faster, the framework helped ensure these communications reached users reliably, reinforcing the platform’s trust and credibility.

Support teams, often the first line of response during incidents, became significantly more empowered. They could speak with confidence during high-pressure situations, such as real-time delivery failures or vendor outages, backed by data from traceability logs and dashboards. This improved client communication reduced SLA breaches and escalations, and ultimately strengthened client relationships.

For end users, even small improvements in delivery speed and reliability made a noticeable difference. Successful and timely delivery of one-time passcodes, fraud alerts, and order confirmations not only improved usability but also lowered abandonment rates in workflows dependent on real-time communication.

From a strategic lens, the availability of delivery performance data allowed leaders to shift conversations from anecdotal observations to data-backed decisions. Product and engineering leads could prioritize technical investments, vendor negotiations, or roadmap items based on observed delivery impact across customer segments and channels.

By transforming delivery observability into a cross-functional capability, the framework closed the feedback loop between technical operations and business outcomes. It demonstrated that platform reliability is not just a system metric, but a business driver directly tied to user experience, client retention, and strategic clarity.

Metric	**Before Framework**	**After Framework**	**Change**
Average Incident Resolution Time	4.5 hours	3.9 hours	↓ ~13%
Monthly Delivery-Related Support Tickets	420	370	↓ ~12%
Time to Diagnose Failed Message	35 minutes	28 minutes	↓ ~20%
Use Cases Reprioritized Based on Data	-	1	+1 (actionable case)
Tiered Retry Policies in Use	0	1	Introduced

Table 1: *Operational Impact Metrics*

5. Conclusion

Notification systems are often treated as technical backdrops, yet their performance directly impacts user trust, business continuity, and operational efficiency. This paper introduced a product-led observability framework designed specifically for real-time notification platforms that span multiple channels and external vendors.

The proposed approach—anchored in message-level traceability, role-specific dashboards, anomaly detection, and retry behavior analysis—demonstrated measurable gains in incident resolution speed, support efficiency, and product decision-making. Importantly, it showed that observability is not just a backend function, but a cross-functional enabler that connects engineering precision with user outcomes and business priorities.

By embedding observability as a first-class concern across roles, the framework promoted a culture of shared accountability. Support agents could investigate issues without engineering bottlenecks. Product teams could validate tradeoffs and investments using real delivery data. Engineering teams could respond faster and plan smarter.

Looking ahead, the framework opens doors to future enhancements, including predictive anomaly detection, machine learning–driven diagnostics, and even client-facing delivery dashboards. These opportunities reaffirm that observability—when implemented with purpose—can become a foundational pillar of platform trust and operational excellence.

References

[1] Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., & Vahdat, A. (2010). Dapper, a large-scale distributed systems tracing infrastructure. *Technical Report*, Google Inc.
[2] Turnbull, J. (2020). *Monitoring and Observability*. Turnbull Press.