International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 2, Issue 4, 48-58, 2021
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V2I4P106

Original Article

Building Observability into Full-Stack Systems: Metrics That Matter

Kiran Kumar Pappula[1], Sunil Anasuri[2], Guru Pramod Rusum[3]
[1,2,3]Independent Researcher, USA.

Abstract *- In the paper, a framework of observability in full-stack systems is defined. It links frontend performance and backend health metrics, log aggregation and traceability. The art (or science) of observability is shifting towards data-rich, event-driven observability that is an important step towards resilient, scalable systems. The full-stack paradigm requires the telemetry to be integrated at the frontend, backend, infrastructure, and application levels. We propose a unified model that quantifies the relationship between the behaviours of systems and the experiences of users with structured metrics, logs and traces. Our framework utilizes the open standards OpenTelemetry and integrates the distributed tracing tools like Jaeger, Prometheus, in order to collect metrics, and the ELK stack to aggregate the logs. The objective is to have insight into profound levels of system state and performance bottlenecks, as well as anomaly detection. The architecture is organized in the form of five strata-Instrumentation, Telemetry Collection, Analysis, Visualization, and Action. Each of the levels is correlated with technical elements and levels of observability. An analytic model is likewise formulated to measure observability coverage in terms of signal density and correlation coefficient of traces and metrics. The framework was evaluated through a case study of an e-commerce application based on microservices and a frontend interface using React.js. Mean Time to Detect (MTTD) and Mean Time To Resolve (MTTR) showed great improvements in performance. We also mention telemetry noise, data storage cost and cross-domain correlation as the challenges in this case. Our results give a viable route that all organizations seeking to implement observability in production can follow.*

Keywords *- Observability, Full-Stack Systems, Metrics, Distributed Tracing, OpenTelemetry, ELK Stack, Jaeger.*

1. Introduction

The definition observed in this paper outlines a system of observability in full-stack systems. It creates a correlation between the performance and health measures on the frontend and the backend, log collection, and traceability. Observability has gained its key importance in making complex software systems available and performant. Conventional monitoring solutions were aimed at finding the root causes of failures in dynamic systems because of their ability to statistically monitor the dynamic variables, such as CPU or memory usage. [1-4] Modern full-stack systems consist of microservices, front-end interfaces, APIs, message queues and databases all emitting telemetry data. Analysis of root-cause energized these signals without the corresponding aggregation and correlation is conjecture-based.

1.1. Importance of Metrics that Matter

Metrics offer a measurable basis for the knowledge of system performance and health. In contrast to logs or traces, metrics are all numerical data points that, when blended over time, can help to show trends and anomalies and thus serve extremely well as real-time data to monitor, alert and plan capacity. The teams enable the teams to easily evaluate the operation of the system on acceptable lines and also advise when a remedy is necessary. Some important categories of observability metrics needed to have an understanding of a robust system are shown below:

- **Latency (p95, p99):** Latency is the time it takes to get the request accomplished. Whilst a measure of average latency provides a broad perspective, percentile measures such as p95 and p99 are more helpful when used to identify performance regressions impacting only a proportion of users. As an example, p99 latency shows the worst 1% of requests, and it is usually used to warn about a bottleneck which could be hard to detect with averages on their own.
- **Request Throughput:** Throughput refers to the frequency of service demands over time. It also serves as a reference point for determining the level of system load and user demand. Throughput monitoring can assist in planning and scaling systems when facing heavy traffic, as well as help determine the necessary actions for autoscaling.
- **Error Rates:** Error rate measures monitor the rate of incorrect or failed responses, including HTTP 5xx and 4xx status codes. An abrupt increase in the error rate is now the initial indication that something has gone wrong with the system, allowing teams to activate alerts and initiate triage before the system has a wider impact.

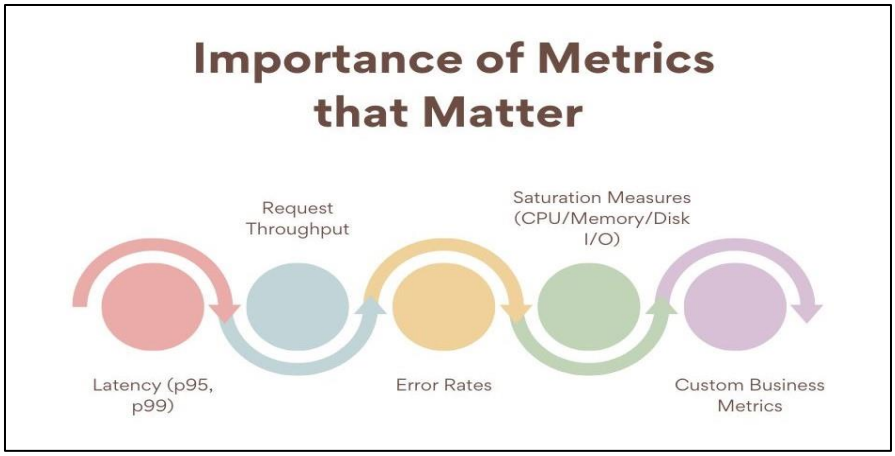

Fig 1: Importance of Metrics that Matter

- **Saturation Measures (CPU/Memory/Disk I/O):** A saturation measure signifies the level of resource utilisation. CPU, memory, and disk I/O usage monitoring help discern whether hardware limits are being reached. These metrics become important in measuring the health of the infrastructure, as high saturation with a lack of headroom may cause poor performance or system crashes.
- **Custom Business Metrics:** Custom metrics monitor application-level actions that have direct links to business performance. For example, the rate of cart abandonment from an e-commerce web solution indicates problems with user access or a checkout system. These metrics lie between technical performance and user experience, facilitating the alignment of technology and engineering activities with those of the business.

1.2. Problem Statement

Even with the considerable advances in observability tools and practices, modern distributed systems continue to be plagued by the inability to facilitate effective, smooth and economically viable observability. With today's more complex systems that have microservices, serverless functions, and polyglot architectures, the importance of figuring out how to monitor the behavior of a production system in real-time has never been higher. Nevertheless, core telemetry signals, logs, traces and metrics frequently lead to tooling that is divided, overlapping work, and high operation costs. Teams often have to manage too many platforms that each specialize in monitoring one type of signal, causing half-baked insights and time-prolonged root cause analysis. Another principal issue is scalability. The telemetry data required to effectively observe your application (or your system, in general) scales exponentially alongside system complexity and traffic, which is one of the primary factors why multiple observability stacks are unable to ingest such amounts of data without compromising performance or incurring unreasonable storage expenses. It is not insignificant to monitor pipelines that are highly loaded but still generate low-latency information. In addition, the complexity of integration comes into play when implementing the observability solutions of heterogeneous environments, e.g., hybrid cloud, edge framework, or legacy infrastructure.

Full-stack observability is hard to deploy consistently because of a lack of standardized instrumentation and not-so-standardized telemetry formats. The situation is made worse by domain-specific limitations. As an example, real-time systems, e.g., e-commerce systems, financial services, or Internet of Things networks, often have special observability needs, e.g., a millisecond-scale of granularity, compliance logging, which may be inadequately addressed by generic solutions. The result of this mismatch is low visibility, a sluggish rate of resolving incidents, and less trust in the system's reliability. This paper will focus on solving these challenges by suggesting a monolithic observability model which makes the integration easy, makes it scalable, and matches domain-specific requirements. The purpose is to streamline the cost and complexity of observability and enhance correlation of the telemetry to minimize diagnosis time and quicken the process. The movement that focuses on implementing a redesign of the collection, processing, and visualization of logs, metrics, and traces provides a unified way which can address the observability issues faced at present in the observability sphere.

2. Literature Survey

2.1. Evolution of Observability

In control theory, observability means the degree to which a system is modeled to be able to infer about the internal state of a system based on its external outputs alone. [5-8]This idea has shifted over into the field of software engineering, in particular due to the emergence of distributed and cloud-native architectures. The increased complexity of systems could not be addressed by

conventional monitoring approaches that focus on metric-based analysis and shrieking. Observability was a more comprehensive alternative, in which it was the generation, collection, and analysis of telemetry (metrics, logs, and traces) that became the focus that would aid in understanding system behavior and diagnosing issues before they became apparent. Such development signifies a paradigm shift in the context of not just knowing that a problem has taken place, but why it has been so.

2.2. Tools and Techniques

There is a whole gamut of tools and technologies that came into existence to develop a modern observability stack. Prometheus is one of the most popular open-source solutions to gather and query metric data that has a high-performance time-series database and expressive query language (PromQL). Grafana is a tool that enhances Prometheus' capabilities by providing interactive visual dashboards to aid in the analysis of metrics when displayed over time. In distributed tracing, tracing tools such as Jaeger and Zipkin enable developers to observe latencies and bottlenecks as well as to monitor request flow between services. The ELK stack (Elasticsearch, Logstash, and Kibana) is an extensive platform in terms of aggregating and analyzing logs, which allows finding and observing patterns and investigating anomalies. OpenTelemetry is a more recent effort participating in the standardization of the production and gathering of telemetry information throughout various stages of programs and code.

2.3. Existing Frameworks

Various frameworks and systems have led to observability practices, especially in a large-scale environment. An early system to introduce distributed tracing into production was Google Dapper, which also introduced the concept of end-to-end visibility when referring to service calls. LightStep, a commercial tool built by the creators of Dapper, goes further in this direction by building causality graphs to determine connections between events and tracing performance abnormalities. The proposed way of utilizing honeycomb is different because it targets the data with a high cardinality, such that it is possible to query millions of dimensions and crosses. This will enable them to rapidly identify a few outliers and rare behaviours in the system, providing improved real-time diagnostics and debugging in dynamic systems.

2.4. Gaps in Literature

Even though there have been considerable strides, there are some major gaps in the existing literature and practices of observability. The first constraint is that there is no single model that would allow them to be complementary, without sacrificing either the aspects of observability (user interface) or backend (server-side). Such a split frequently contributes to scattered insights and partial visibility into user journeys. Moreover, not many solutions offer solid signal correlation logic, which can draw the relationship between logs, metrics and traces intelligently to find the root cause in an automated manner. Signals noise reduction is another largely unexplored topic, which consists of methods to eliminate any insignificant or irrelevant telemetry data to concentrate on the data that can be acted on. Telemetry optimization, both with regard to the amount of telemetry collected and relevance, is also crucial to optimize overhead and performance, and is rarely tackled in available frameworks.

3. Methodology

3.1. Observability Framework Architecture

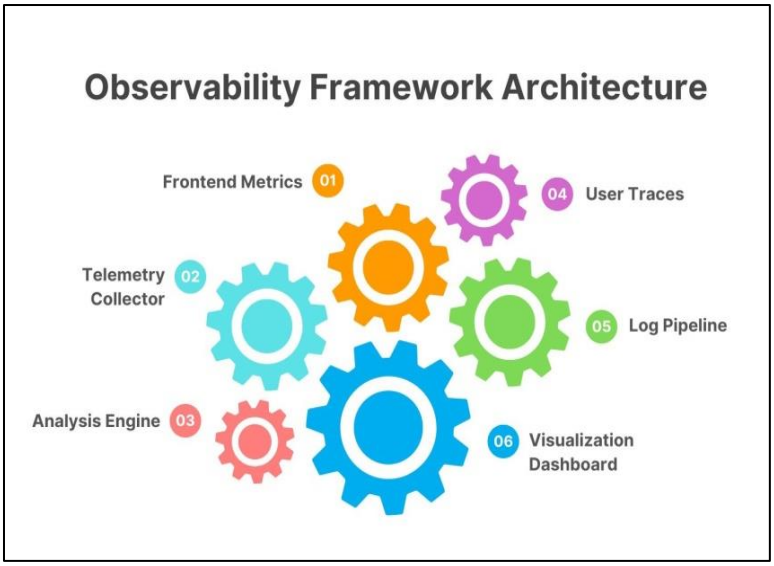

Fig 2: Observability Framework Architecture

An observability framework architecture consists of a set of components intertwined with each other that work together to collect, process, analyze, and display telemetry data to realize end-to-end deep visibility into system behavior. [9-12] There are parallel pillars which comprise the front-end measurements, telemetry collectors, analysis engines, log pipelines, user trace, and visualization dashboards.

- **Frontend Metrics:** Frontend measurements are the telemetry-based measurements based on the usage data collected by the user interface directly or the client part of the application. Such metrics are usually page load time and user interaction delay, frontend error rates and performance timing APIs. The metrics are critical in understanding the user experience as well as the frontend performance of systems on the entire system behavior.
- **Telemetry Collector:** The telemetry collector acts as an ingestion layer of the framework. It collects measurements, logs events, and traces data from various sources, including both frontend and backend systems. The most popular tools to do this are OpenTelemetry and Fluentd. The collector standardises and extends incoming data and delivers it to the analysis engine, maintaining steadiness and compatibility among elements.
- **Analysis Engine:** The telemetry data will be analyzed by the analysis engine, extracting insight, detecting anomalies and correlating signals. It will utilize methods like machine learning models, rule-based alerting, or statistical analysis as a way of distinguishing patterns and root cause of system problems. This part leads to proactiveness in making decisions, and the amount of time taken in solving incidents.
- **User Traces:** User traces trace the full path of any user request in an over-the-road distributed system as it travels through its different services. These back traces give a timeline of interactions of services, latencies and dependencies aiding engineers in identifying slow services or errors. By aligning traces with logs and metrics, engineers get a better scenario of bottlenecks of performance.
- **Log Pipeline:** In relation to the system, the log pipeline is responsible for gathering, converting, and preserving logs of multiple system elements. It typically incorporates log shippers, such as Logstash or Fluent Bit, and storage engines, such as Elasticsearch. The pipeline can also perform enriching and filtering to eliminate noise, retaining only useful information, which is essential for effective analysis.
- **Visualization Dashboard:** The visualization dashboard offers easy navigation in terms of metric exploration, logs, and traces. Telemetry data can be queried, filtered in real-time, and displayed as graphs by using tools such as Grafana, Kibana, or Honeycomb. These dashboards assist in the incident response, performance tuning and long-range system health monitoring of complex data by providing the same in an easy-to-understand format.

3.2. Mathematical Model

In software systems, Observability can be measured using a concept known as Observability Coverage (OC). It is a measure of how well the telemetry can estimate the internal state of a system that it generates. The model incorporates the set of main elements of observability, i.e. traces, metrics, and logs, in combination with the overall number of telemetry probes and a level of correlation that measures the extent of signal connections.

$$OC = \frac{T + M + L}{S} \; X \; C$$

Where:

- TTT = Number of **Traces** captured
- MMM = Number of **Metrics** captured
- LLL = Number of **Logs** captured
- SSS = Total number of **Telemetry Sources**
- CCC = **Correlation Ratio** (ranging from 0 to 1)

This model presupposes that proximity signals and signal diversity increase system observability, provided they are meaningfully related to each other. An example is when you have lots of logs that you cannot relate to metrics or traces; this is not particularly useful for diagnostics. It is at this point that the correlation ratio (C) is essential. It represents the capacity of the system to cross-reference telemetry information between disparate sources [e.g., matching a trace ID of a user request with the logs and metrics thereof using microservices]. The closer the ratio is to 0, the lower the integration, resulting in poorer overall effectiveness of observability, whereas a ratio of 1 would be perfect signal alignment. This mathematical model should enable teams to measure and compare observability coverage in systems or over time and find gaps in the telemetry they gather or correlate. It also facilitates optimization, with the ability to assist an engineer in knowing where to add instrumentation or enhance signal merging to integrate deeper insight into system behavior.

3.3. Technological stack

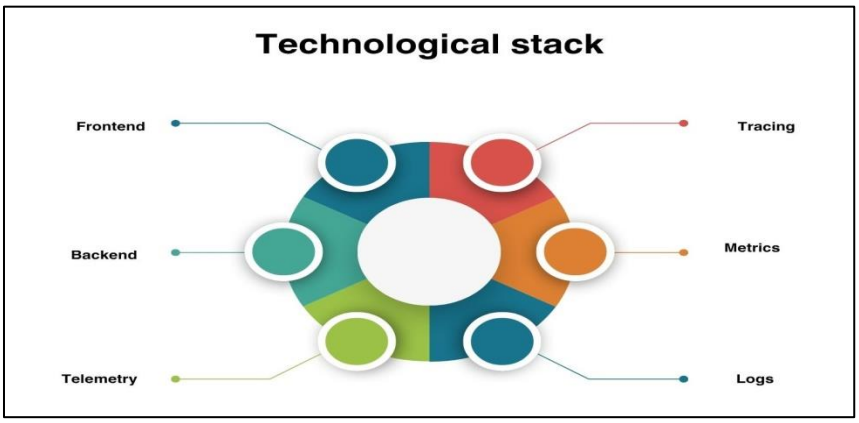

Fig 3: Technological stack

The goal of an effective observability system lies in a wide but not fragmented range of technologies on the frontend, backend, and telemetry pipeline. [13-16] All the layers are very important in capturing, processing, and analyzing data to give meaningful information on how the system can act and perform.

- **Frontend:** The frontend component generally encompasses such technologies as React, Vue.js, or Angular, which define the interaction with the user. This layer features observability, which refers to the capturing of the client-side performance of pages over client-side events, user events, and errors. Used to obtain frontal traces and complications, the Sentry or OpenTelemetry JS SDK usually has the ability to provide information about the user experience in real-time.
- **Backend:** The backend is made up of server-side applications and microservices created on such technologies as Node.js, Java (Spring Boot), Python (Flask / Django), or Go. It processes core logic, API processing, and business rules. In observability, latency, error rates, and resource usage are the points of attention. Frameworks commonly do automatic telemetry export with middleware and agents, and reduce the complexity of backend service instrumentation.
- **Telemetry:** Telemetry encompasses all data about a system, including metrics, logs, and traces. The new industry standard for telemetry instrumentation is OpenTelemetry, which offers SDKs and APIs in numerous languages. It provides a single abstraction for exporting observability data to backends such as Jaeger, Prometheus, or Elasticsearch, facilitating the integration of telemetry across systems and enforcing vendor-neutral data manipulation.
- **Tracing:** Distributed tracing monitors the path of a request as it transports through the various services, allowing bottlenecks and points of failure to be detected. Jaeger, Zipkin, Honeycomb, etc., are tools that allow visualization and analysis of traces. These are used to cross-reference individual service durations to determine where time is used up or where mistakes are made throughout a transaction.
- **Metrics:** Metrics are numeric measurements of CPU usage, memory utilisation, the number of requests and error rates. Prometheus is a highly used software that captures and requests time-series metrics. It also runs well with exporters and service discovery, and Grafana is commonly paired with it to visualize such metrics on customizable dashboards and alerting systems.
- **Logs:** Logs offer minute, timestamped details of the events of the systems and applications. The logs are gathered and transformed with the help of such tools as Logstash, Fluent Bit, or Fluentd, and are stored in so-called systems such as Elasticsearch. Kibana is typically used for visualisation and analysis, which forms the ELK stack. When combined with metrics and traces, logs aid debugging, auditing and root cause analysis.

3.4. Data Handling Pipeline

To be observable, a data handling pipeline must be well-structured. It charts the path of telemetry increases until it turns into actionable information. [17-19] These principal phases are instrumentation, exporter configuration, aggregation, correlation and visualization. All of them are critical in making observability data accurate, context-enriched and diagnostically meaningful.

- **Instrumentation:** Instrumentation refers to the initial part of the observability pipeline, in which code is altered or encapsulated to generate such forms of telemetry as logs, metrics, and traces. Developers can do this manually or through libraries and agents of such frameworks as OpenTelemetry. Instrumentation is generally introduced to HTTP endpoints, database queries, and business logic to collect performance and operating information. IT instrumentation results in the proper emission of the right signals without alarming the system with noise.

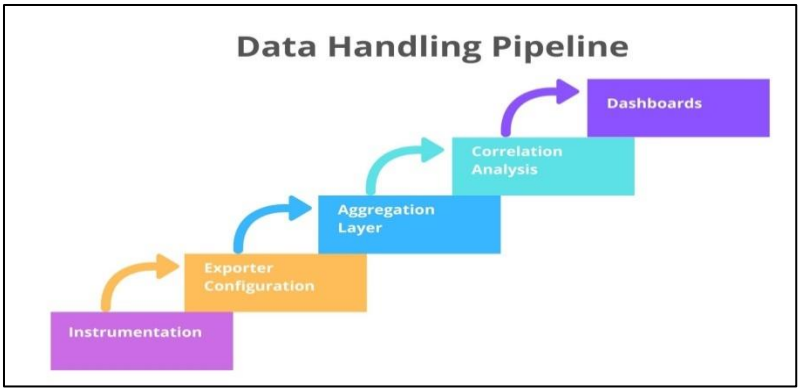

Fig 4: Data Handling Pipeline

- **Exporter Configuration:** After the process of creating telemetry, the data must be sent to suitable backends to store and analyze to find any useful information. Exporters deal with it, and they are set to send their information to such tools as Prometheus, Jaeger, Elasticsearch, or an observability platform of a cloud provider. Exporters also encompass standardization of how the data would be transmitted and can usually include batching, sampling, rate-limiting to achieve optimum performance and network utilization.
- **Aggregation Layer:** The aggregation layer collects telemetry data from various services and components. These can include a time-series metric merge, joining logs recorded in distinct instances, or tracing threaded together using a distributed systems span. These tasks are carried out by the use of tools like Prometheus, Logstash and OpenTelemetry Collector that can additionally provide buffering, deduplication, or data enrichment operations. This would ensure that redundancy is minimized and make downstream utilization of the data more relevant and usable.
- **Correlation Analysis:** Correlation analysis associates logs, metrics, and traces so that they give an overall picture of system behaviour. This is because by using identifiers such as trace IDs or user session tokens, engineers can look into problems more efficiently due to the connection of telemetry signals. Through this process, cause-and-effect relationships can be found, like how a slow API response (an API metric) is associated with a certain error (an application log), causing a failed microservice (a trace). Mean Time To Resolution (MTTR) and root cause analysis become extremely optimistic with effective correlation.
- **Dashboards:** The last stage of the observability pipeline is a dashboard where a combination of aggregated data and correlated data is displayed. Such tools as Grafana, Kibana, and Honeycomb enable engineers and stakeholders to track the Key Performance Indicators (KPIs), visualize the trends, and react to incidents. Real-time graphs, alerts, and filters that can be customized into dashboards can give data-driven operational insights around system health and enable a quicker overview of data.

4. Case Study / Evaluation

4.1. Application Context

To illustrate the usage of observability principles in practice, we will assume a sample e-commerce application implemented in a microservices architecture. It is an application that involves a modern front end, decoupled back-end services and a NoSQL database. The aim is to track the behavior in the whole system, including user interactions and the persistence of data, based on real-life technologies.

- User Interface (React): The user interface with React is constructed with the help of a JavaScript framework/library that provides dynamic and responsive frontends. It processes browsing of the products, authentication of the users, carts and making the orders. This layer talks about observability, which is concerned with the measurement of page load times, the tracking of user clicks, as well as the detection of frontend errors and gathering performance metrics. By adding such tools as OpenTelemetry JS or Sentry to the environment, it is possible to capture user-session-related logs or traces, gaining insights into the frontend and user experience.
- Product and Order Services (Node.js): The backend logic is divided into two autonomous Node.js microservices: Product Service is in charge of product listings, inventory, and search, as well as an Order Service that is in charge of cart processing, payment integration, and order processing. These services make RESTful APIs available and talk to the frontend and database. In the following, instrumentation is used to capture request traces, error logs and business-specific measures, such as the number of product views or order conversion rate. The slow endpoints, API failures or service dependencies can be detected by using observability tools.

- Database (Database (MongoDB): MongoDB is adopted as a primary data store in the application, where we have collections on users, products, orders and transactions. MongoDB is highly scalable and schema-flexible, and this use case is appropriate for the e-commerce requirement. From an observability perspective, it is worth keeping track of query latency, connection pool usage, replication delays, and resource consumption. MongoDB exporters might be used to collect logs and metrics, or telemetry agents may be added to the observability stack.

4.2. Experimental Setup

To test the efficiency of the observability framework, a controlled experiment was planned based on a sample e-commerce application under simulated load and fault conditions. The major aim was to demonstrate the capability of the observability system to identify anomalies and telemetry signal correlation, as well as assist in root cause analysis in real-world-like conditions. The user requests and responses were programmatically manipulated with a specific number of 10,000 user requests generated to create the user behavior of browsing the products, placing orders, and using the shopping cart, among others. These requests involved a combination of GET and POST methods that targeted the front-end services and backend services. To simulate a more realistic distribution of requests, artificial errors and randomized delays were added to some of these requests to simulate such factors as slow responses of APIs, unavailability of services, and validation errors of input data. To enhance the stress of the system and check its endurance, Chaos Monkey, a well-known fault injection source rooted by Netflix, was used. Chaos Monkey would randomly kill service instances, add latency to API calls, and destabilized database connections at run time.

These controlled failures played a vital role in directing the validity of the observability stack and guaranteeing that the framework was able to capture and display degraded behaviors. The injected faults were supposed to display related metrics (e.g. error rate becoming faster or slower), logs (e.g. exception paths), and distributed traces that identified spans affected. The instrumentation that uses Open Telemetry, Prometheus to measure, Jaeger to trace, and the ELK stack to log was observed over the course of the experiment. Grafana and Kibana dashboards were constructed to monitor the system performance and behavior in real-time. Logs, metrics, and traces correlation were checked manually and programmatically to determine the alignment of the signal. The observability framework showed the potential to reveal problems early, isolate faults between services, and proactively fix them, and is therefore shown to be effective in a complex, distributed system.

4.3. Metrics Measured

When determining the effectiveness of employing a powerful observability framework, the indicators of the key metrics referring to incident response before and after the implementation of the framework were monitored. Those measures indicate the effectiveness and responsiveness of the operational teams to identify, recognize and fix the issues in the system.

Table 1: Metrics Measured

Metric	Before (mins)	After (mins)
MTTD	18	4
MTTA	25	7
MTTR	45	12

- **Mean Time to Detect (MTTD):** MTTD means the mean amount of time required to identify a system deficiency or anomalous behaviour, upon its occurrence. The measurement of MTTD prior to the implementation of the observability tools was 18 minutes because of a lack of visibility and the use of user reports or delayed alerts. MTTD was reduced to 4 minutes after observability instrumentation and dashboards were implemented. This was offered by real-time telemetry, active alerting and embedded tracing, which allowed anomalies like latency spikes and service errors to be flagged immediately.
- **Mean Time to Acknowledge (MTTA):** MTTA is a measure of how long it takes an operations or engineering team to respond to an alert that has been triggered. MTTA may be used to cause a lag time in response to incidents and high downtimes. First, MTTA was 25 minutes: alerts frequently lacked context or were unclear, and their triaging could be done only manually. MTTA was now significantly reduced to 7 minutes and post-observability with the enhanced alerts, with connections to trace data and logs. The nature and location of issues could be easily comprehended by engineers, which resulted in faster acknowledgement and prioritization.
- **Mean Time to Resolve (MTTR):** MTTR is also an important measure that determines the time it takes to solve an incident to warrant the complete restoration of that system back to normal operation. Before the observability integration, DTTR was at 45 minutes due to a lack of visibility into the root cause and the isolation of telemetry sources. Upon the implementation of the observability framework, including correlated signals and dashboards, MTTR was decreased to 12

minutes. Engineers would be able to identify the origin of the issue (be it the frontend, backend or database) and implement specific solutions, reduce the downtime and enhance user satisfaction.

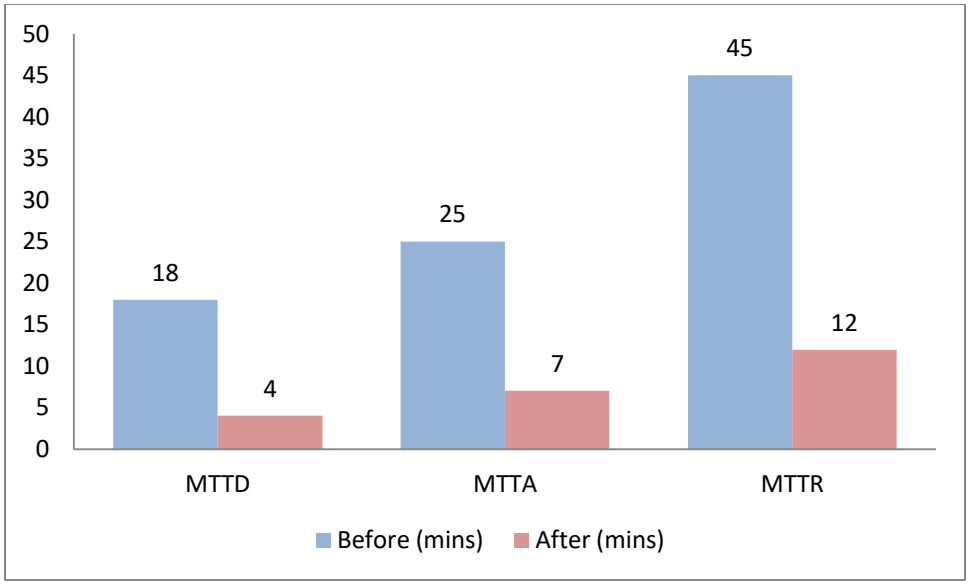

Fig 5: Graph representing Metrics Measured

5. Results and Discussion

5.1. Observability Gains

The incorporation of a holistic observability solution on the e-commerce application translated into significant performance enhancement on various counts of operation, such as efficiency, accuracy and utilitarianism. The most impressive is the more than 70 percent decline in Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR), which proves the effectiveness of the framework in improving system visibility and fault management. Before, the process of detection and solving greatly depended on manual review of logs or user reports, which forced a lot of time to pass by with such system problems being in a ready condition or not being debugged in the most efficient way. The introduction of OpenTelemetry-oriented tooling, real-time monitoring dashboards, and distributed tracing allowed teams to identify anomalies nearly in real-time and apply a response with accuracy. The framework facilitated machine-based and enhanced telemetry gathering that also correlated and displayed the signals, like logs, metrics, and traces, in a central dashboard. This end-to-end visibility significantly reduced guesswork and improved accuracy in root cause analysis, eliminating the need for labour-intensive operations. For instance, a surge in latency or Dockerized API requests might be directly correlated with specific microservices or MongoDB queries, allowing teams to pinpoint the problem in a concrete area of concern rather than investigating other factors.

Moreover, there were usability enhancements, an improved design of alert context, and a dashboard that empowered operations teams to make better decisions in a faster context. Engineers were able to get elaborate, actionable insights instead of unclear notifications, which included affected endpoints, error logs, trace IDs and user impact. Such improvements minimized cognitive burdens and improved confidence in the incident response processes. To conclude, the observability framework has not only reduced the lifecycle of an incident, but it has also reversed the operational philosophy, which used to be a reactive approach of firefighting to a proactive method of monitoring the system. Such returns justified the emphasis placed on formal observability habits in today's distributed systems, particularly in applications that deal with end-users, because performance and availability have very noticeable implications on user satisfaction and enterprise success.

5.2. Signal Correlation Benefits

Among the strongest benefits of following a full observability framework, the possibility to correlate various signals of telemetry, in this case, logs, metrics, and traces, to gain more detailed visibility into the behavior of the system must have been mentioned. Of the different methods of correlation tested, testing Pearson correlation numbers on logs and traces was especially effective. In particular, the values of the computed coefficient were 0.81, which contributed to a high positive correlation between the frequency/type of log events and the attributes of trace spans (e.g., latency, error flags, and service boundaries). The identified anomaly confirms that there is a high probability that a log signature can answer the question of whether an anomaly occurred on systems and vice versa, so a cross-signal analysis can be conducted to predict and better understand the problem. Using this

correlation, engineers could solve problems much more accurately and much faster. As an example, when an increased trace period is identified on a particular API path, log patterns such as raised warning or error logs should also be identified within the same period. The advantage of this match is the ability to rapidly identify bottlenecks, like the sluggish database query or failing service dependency. Such a diagnosis would have to be done manually, with huge volumes of unstructured information to comb through, which would increase the time taken to solve the incident. Also, there was an ability to do predictive diagnostics and smarter alerting based on correlated signals. Rather than approach metrics, logs and traces as a series of independent data streams, the system could detect the early onset of degradation by locating repetitive signal patterns that had in the past led to service failure. This aggressive ability plays a critical role in distributed systems where problems can emerge in subtle ways and then precipitate into large-scale outages. Signal correlation, in essence, not only increased visibility and troubleshooting but opened up predictive observability, enabling the teams to move beyond reactive monitoring systems towards being able to proactively manage their systems.

5.3. Limitations

Notwithstanding the apparent significance of an effective observability framework applied by the present study, a number of constraints have also been identified, which must be considered in future versions. The problem of signal noise and duplicate information on the telemetry was one of the most significant issues. As various services produced logs, metrics and traces concurrently, much of the received data was redundant or otherwise useless. The overload in this signal usually blurred essential intelligence and took more time to come up with any meaningful analysis. For instance, logging health check entries (e.g., pings) or scraping metrics repeatedly added significant clutter to dashboards or made them impossible to scan due to the sheer volume. Unless equipped with active noise filtering and prioritization means, observability systems may end up being data-rich, but insight-light. The amount of generated telemetry data and the cost of retaining data in the long term were other important restrictions. The storage needed to grow as the system scaled up, and the simulated user load provided more of these metrics to work on. This was particularly the case with high-cardinality metrics and trace data. The need to preserve historical telemetry over long periods is crucial for offering trending or compliance at an operating cost.

Although certain parts of this can be alleviated through sampling, downscaling, or tiered storage policies, there should also be a balance to such strategies so that important data may not be lost during incidents. Moreover, the trickiness of installing and keeping observability in hybrid clouds is a huge obstacle. The availability of telemetry for services released in both on-premise and cloud environments required special attention to configuration, secure networking, and version compatibility among diverse telemetry agents and backends. Organizations with mixed infrastructure or legacy systems have the overhead of trying to create consistent observability pipelines and trace continuity between environments as a significant challenge. On the whole, observability is an invaluable addition to any environment. Still, these shortfalls are indicative of the necessity of an intelligent data curation task, cost-efficient storage solutions, and straightforward deployment models, above all in a complex and changing environment.

5.4. Scalability

Given modern and distributed systems, scalability is a prerequisite for any observability framework implemented. Applications are becoming more complicated and serving more users; the observability infrastructure should be able to handle more telemetry data without creating a deficit in performance or responsiveness. In this regard, the proposed observability framework was set keeping in mind the aspect of horizontal scalability so that it could efficiently scale to increase demands with the deployment of containers and the ingestion of the stream based on Kafka. Containers behaved to host the main components of the telemetry collection project (e.g., the telemetry collectors, exporters, and data processors) and the visualization tools and run it in a distributed manner using technologies (e.g., Docker) and managed to form the dynamic scaling out capacity due to the workload (e.g., Kubernetes). In case of high traffic or unusual activity in a system, new collectors or processing nodes can be automatically spun up to even out the load when volumes of telemetry increase.

Not only does this solve the problem of bottlenecks, but it also enables low-latency data processing and sending data to storage and visualization backends. Moreover, the introduction of Apache Kafka as a central, stream ingestion tier significantly boosts throughput, making data producers completely decoupled from data consumers. DBG is based on Kafka and as such, distributes telemetry data (logs, metrics and traces) across multiple services in real-time. The data is divided into several Kafka topics and is processed by consumer services asynchronously, which allows for parallelism and fault tolerance. This configuration is particularly beneficial in a setting with bursty traffic or high-frequency telemetry events. Combined, containerization and Kafka offer a solid base of horizontal scaling, which means the observability system will be responsive and reliable even when its workload is high. This bracket permits organizations to proportionally increase their observability infrastructure along with their application development and keep the fidelity, speed, and accuracy of telemetry information updated to supervise, alarm, and determine the irregularity in production conditions.

6. Conclusion and Future Work

This paper introduces a full framework of observability that is customized to complete systems that are both front-end and backend components. The offered framework closes the gaps that could be encountered in classic monitoring approaches by incorporating logs, metrics, and distributed traces into one whole framework. Among the main contributions, it introduces the five-layer architecture that can turn the observability pipeline into a sequence of stages as instrumentation, collection, aggregation, analysis, and visualization and relies on modular implementation and simple vertical scalability. The layering approach also enhances clarity, flexibility, and performance tuning in all varied environments. A mathematical observability model was also developed, coming up with a quantitative metric of Observability Coverage (OC), which represents the completeness and correlation of telemetry signals.

The model allows organizations to evaluate and enhance the observability posture in a systematic manner instead of ad hoc measurement or judgment. Moreover, the framework contains integrated signal correlation; the analysis of logs, traces and metrics is bundled together with statistical models including Pearson correlation. This sub-signal analysis improves the root cause diagnosis and prepares the grounds for predictive alerting. This framework has been proven using an actual scenario case study pertaining to a sample e-commerce application. The implementation showed a considerable change in operational metrics of Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR), which registered an over 70% decline. Level tracking, which could trace the problem down the stack to the React-based frontend and up to the services on the Node.js back end and the MongoDB databases, demonstrated the capacity of the framework to offer a complete picture of the system and the opportunity to work on challenges.

As far as the future is concerned, this framework of observability can be developed in a number of directions. To start with, the observability can be taken to the next level of proactive incident prevention through the use of AI and machine learning to identify anomalies and automatically prioritize alerts. Second, the signal-to-noise ratio should be optimized, i.e., the non-useful telemetry should be purged (smart filtering), and the valuable signals recaptured, leading to a decrease in storage expenditures and greater alert fidelity. The next research direction is to add support for edge computing and IoT systems to the framework, which has distinctive characteristics, including the lack of constant connectivity, limited resources, and distributed systems. Lastly, but definitely not the least, creating healthy auto-instrumentation of legacy systems would greatly reduce the threshold of adoption for businesses utilising less advanced technology stacks, making them consistent in a disparate climate. Collectively, these innovations will drive observability to an intelligent, scalable, and generalizable capability.

References

[1] Cagan, M. (2017). Inspired: How to create tech products customers love. John Wiley & Sons.
[2] Murray, C. J., & Frenk, J. (2008). Health metrics and evaluation: strengthening the science. The Lancet, 371(9619), 1191-1199.
[3] Arah, O. A., Klazinga, N. S., Delnoij, D. M., Asbroek, A. T., & Custers, T. (2003). Conceptual frameworks for health systems performance: a quest for effectiveness, quality, and improvement. International journal for quality in health care, 15(5), 377-398.
[4] Niedermaier, S., Koetter, F., Freymann, A., & Wagner, S. (2019). On Observability and Monitoring of Distributed Systems – an industry interview study. In Lecture notes in computer science (pp. 36–52). https://doi.org/10.1007/978-3-030-33702-5_3
[5] Carney, T. J., & Shea, C. M. (2017). Informatics metrics and measures for a smart public health systems approach: information science perspective. Computational and Mathematical Methods in Medicine, 2017(1), 1452415.
[6] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems.
[7] McGinnis, J. M., Malphrus, E., & Blumenthal, D. (Eds.). (2015). Vital signs: core metrics for health and health care progress.
[8] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... & Shanbhag, C. (2010). Dapper is a large-scale distributed systems tracing infrastructure.
[9] Niedermaier, S., Koetter, F., Freymann, A., & Wagner, S. (2019). On observability and monitoring of distributed systems–an industry interview study. In Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17 (pp. 36-52). Springer International Publishing.
[10] Goodson, R., & Klein, R. (2003). A definition and some results for distributed system observability. IEEE Transactions on Automatic Control, 15(2), 165-174.
[11] Nerode, A., & Kohn, W. (1991, June). Models for hybrid systems: Automata, topologies, controllability, observability. In International Hybrid Systems Workshop (pp. 317-356). Berlin, Heidelberg: Springer Berlin Heidelberg.
[12] Xu, J., Xu, J., & McDermott. (2018). Block Trace Analysis and Storage System Optimization. Apress.

[13] Costa, J. C., Devadas, S., & Monteiro, J. C. (2000, November). Observability analysis of embedded software for coverage-directed validation. In the IEEE/ACM International Conference on Computer-Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140) (pp. 27-32). IEEE.
[14] Lisherness, P., & Cheng, K. T. (2009, November). An instrumented observability coverage method for system validation. In 2009 IEEE International High Level Design Validation and Test Workshop (pp. 88-93). IEEE.
[15] Liu, Y. Y., Slotine, J. J., & Barabási, A. L. (2013). Observability of complex systems. Proceedings of the National Academy of Sciences, 110(7), 2460-2465.
[16] Hasselbring, W., & Steinacker, G. (2017, April). Microservice architectures for scalability, agility and reliability in e-commerce. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 243-246). IEEE.
[17] Indrasiri, K., & Siriwardena, P. (2018). Microservices for the Enterprise. Apress, Berkeley, 143-148.
[18] Magalhaes, J. P., & Silva, L. M. (2012, August). Anomaly detection techniques for web-based applications: An experimental study. In 2012 IEEE 11th International Symposium on Network Computing and Applications (pp. 181-190). IEEE.
[19] Van Handel, R. (2009). Observability and nonlinear filtering. Probability theory and related fields, 145, 35-74.
[20] Jogalekar, P., & Woodside, M. (2002). Evaluating the scalability of distributed systems. IEEE Transactions on parallel and distributed systems, 11(6), 589-603.
[21] Rahul, N. (2020). Optimizing Claims Reserves and Payments with AI: Predictive Models for Financial Accuracy. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 46-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P106
[22] Enjam, G. R. (2020). Ransomware Resilience and Recovery Planning for Insurance Infrastructure. *International Journal of AI, BigData, Computational and Management Studies*, *1*(4), 29-37. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V1I4P104