

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Laurea Magistrale in Ingegneria Informatica

Observability applied to a centralized system of IoT edges

Relatore:

Prof. Nicola Bicocchi

Correlatore:

Prof. Marco Picone

Candidato:

Manuel Valbusa

Academic year 2025/2026

Contents

1	Introduction to the observability concept	4
2	Foundations of Observability	5
2.1	Observability Concept	6
2.2	Observability Pillars	7
2.3	First Principles	9
2.4	Observability Characteristics	10
2.5	Observability Maturity Model	11
3	Observability and Monitoring	12
3.1	Observability vs Monitoring	13
3.2	Correlation between Monitoring and Observability	15
3.3	Dashboards and Data Analysis	17
3.4	System Analysis	18
3.5	Challenges of Observability in IoT Edge Systems	19
3.6	Monitoring Approaches	21
4	Telemetry Data in Observability	23
4.1	Events and Metrics	25
4.2	Structured and Unstructured Data	27
4.3	Time-Series Data	28
4.4	Sampling and Telemetry	29

5	Observability Signals	30
5.1	Logs	31
5.2	Metrics	33
5.3	Traces	35
5.4	Data Sources	36
5.5	Open-source Tools	37
6	System Architecture	39
6.1	Containers	42
6.2	Backend and Frontend	42
6.3	Docker Architecture	44
6.4	Edge Monitoring Architecture	45
6.5	Design Considerations and Trade-offs	53
7	System Implementation	54
7.1	Project Goals	60
7.2	Flask Prototype	63
7.3	Monitoring Platform Implementation	66
7.4	Azure Infrastructure	70
7.5	Implementation Considerations	71
8	Machine Learning Integration	71
8.1	Hybrid Local Approach	72
8.2	Azure ML Model	73

8.3	Design Considerations and Trade-offs	75
9	Monitoring Metrics and Tools	75
9.1	Base Metrics	77
9.2	Log Management (Loki and Promtail)	79
9.3	Prometheus Integration	80
9.4	Advantages of the Monitoring Infrastructure	83
9.5	Monitoring Results	84
10	Experimental Results and Conclusions	84
10.1	Errors Encountered	85
10.2	Results	86
10.3	Discussion and Conclusion	99

Abstract

This project starts from the study of the observability theory, what really means and what it consists, then we examine the IoT edges and their logs, in order to determine errors and anomalies. We will analyze the most important aspect, the three pillars of observability and the tools we can use. In particular, we examine the concept of logs, metrics and traces. Then we describe the open-source platforms used for analyze and visualize logs attributes and characteristics. For this task we will go to set the Dozzle, Prometheus and Grafana, in order to observe the data in text form and through different kinds of graphs. So we begin with a first analysis of the IoT edges logs, to determine the kinds of messages we have to deal with, to that we observe the content of messages and extract all the important features. After that we are going to set a series of different dashboard, in order to visualize all the extracted attributes, this help us to determine if an edge has errors or anomalies and in general its behavior.

1 Introduction to the observability concept

Modern software systems are increasingly complex, distributed, and dynamic. Applications are often composed of multiple services running across heterogeneous infrastructures such as cloud platforms, container orchestration systems, and distributed microservice architectures. In this context, understanding the internal behavior of a system and diagnosing failures has become significantly more challenging than in traditional monolithic environments.

Observability has emerged as a key concept for addressing these challenges. Originally derived from control theory, observability describes the extent to which the internal state of a system can be inferred from its external outputs. In control theory, a system is considered observable if it is possible to determine its internal state based solely on the information provided by its outputs.

When applied to software systems, observability refers to the ability to understand the internal state of an application by analyzing the data it produces during execution. This includes information such as logs, metrics, and traces, which collectively describe how the system behaves under different operating conditions.

In practical terms, observability allows engineers to investigate unexpected or novel states of a system without the need to deploy additional instrumentation or modify the code. Instead of relying only on predefined monitoring checks,

engineers can explore telemetry data interactively and iteratively, asking new questions about the system as new information becomes available. A first idea of how observability works is represented in the figure 1.

Unlike purely mathematical definitions, modern software observability focuses primarily on the interaction between people and technology. The goal is not simply to collect data, but to enable engineering teams to understand complex systems and diagnose issues effectively. Observability therefore represents both a technical capability and an operational practice that supports the continuous development and operation of production systems.

Traditional debugging and monitoring tools were often insufficient to diagnose deeply hidden issues in distributed environments. As systems grew in scale and complexity, new approaches were required to analyze system behavior in real time and across multiple dimensions of telemetry data. Observability tools were developed in response to this need, providing engineers with mechanisms to explore system state dynamically and correlate multiple data sources during incident investigation.

It is important to note that observability should not be confused with telemetry or monitoring alone. While telemetry refers to the collection of operational data and monitoring focuses on tracking predefined metrics, observability is concerned with the broader capability to explore system behavior and understand unexpected conditions.

For this reason, observability does not depend on a single tool or technology. Instead, it requires a different mindset regarding how telemetry data is collected, stored, and analyzed. By adopting observability practices, organizations can improve system reliability, accelerate incident resolution, and gain deeper insights into how their applications behave in production environments. [7].

2 Foundations of Observability

In the context of this work, these foundational concepts of observability provide the basis for the design of the monitoring architecture described in the following chapters. In particular, the integration of metrics and logs within a distributed IoT Edge environment reflects the need to balance data granularity, system performance, and scalability.

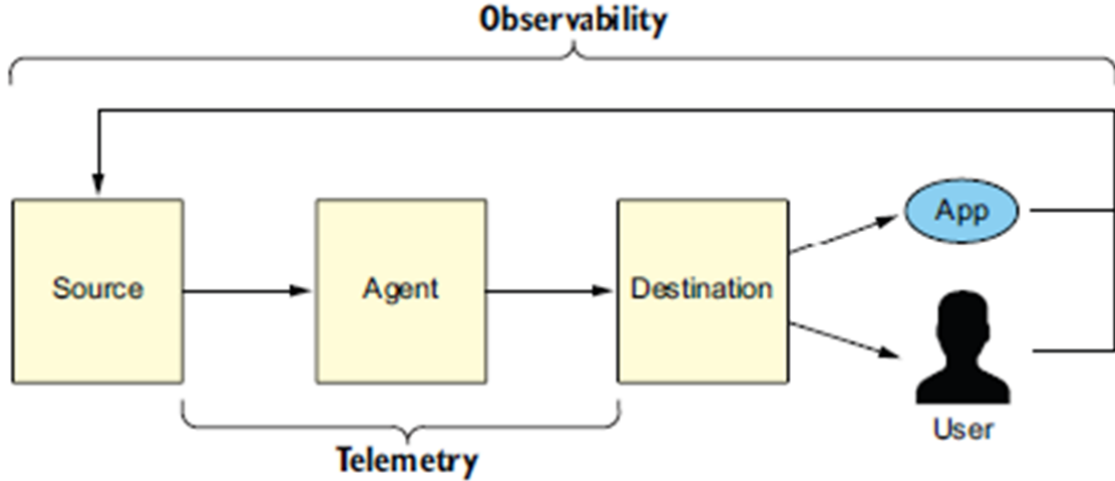


Figure 1: Overview of observability.

2.1 Observability Concept

Observability in software systems is closely related to its original definition in control theory, where it represents the ability to determine the internal state of a system based on its external outputs. In modern software engineering, this concept has been adapted to describe the capability of understanding system behavior by analyzing the telemetry data generated during execution.

From a practical perspective, observability reflects how effectively engineers can interpret and explain the internal state of a system, including unexpected or previously unseen conditions. Modern applications operate in highly dynamic environments, where failures may arise from complex interactions between services, infrastructure components, and external dependencies. As a result, anticipating all possible failure scenarios in advance is often impractical.

Observability addresses this limitation by enabling engineers to investigate system behavior interactively. Instead of relying solely on predefined alerts or metrics, engineers can explore telemetry data dynamically, examining different combinations of dimensions and correlations between events. This approach enables the identification of root causes even in previously unanticipated scenarios.

A key distinction between observability and traditional monitoring lies in the scope of analysis. Monitoring typically focuses on predefined conditions and thresholds that indicate whether a system is operating correctly. While this approach is useful for detecting known failure patterns, it becomes insufficient when dealing with complex distributed systems where the number of potential

failure modes is extremely large.

Observability, by contrast, enables a more flexible and exploratory analysis of system behavior. Engineers can examine telemetry data across multiple dimensions, identify unusual patterns, and follow investigative paths that were not anticipated beforehand. This ability to analyze system behavior without requiring additional instrumentation is a defining characteristic of observable systems.

Ultimately, observability represents a shift in perspective: rather than focusing only on detecting predefined issues, the goal is to enable engineers to understand the system in depth and diagnose novel problems as they occur.

2.2 Observability Pillars

The implementation of observability in software systems is commonly based on three fundamental categories of telemetry data, often referred to as the three pillars of observability: logs, metrics, and traces, as illustrated in [figure 2](#).

Logs are immutable records that capture discrete events occurring within a system. Each log entry typically contains contextual information such as timestamps, error messages, identifiers, or configuration details. Logs provide detailed insights into system behavior and are particularly useful for diagnosing errors and understanding the sequence of events leading to a failure. However, large volumes of logs can introduce noise, making it necessary to correlate them with other forms of telemetry data.

Metrics represent numerical measurements that describe the performance or health of a system over time. Examples include CPU usage, memory consumption, response latency, and error rates. Metrics provide a high-level overview of system behavior and are commonly used to track trends, define alert thresholds, and support capacity planning. While metrics are efficient for monitoring performance trends, they generally lack the detailed context needed for deep debugging.

Traces describe the path of a request as it travels through different components of a distributed system. A trace is composed of multiple spans, each representing an operation performed by a service during the execution of a request. Tracing enables engineers to visualize dependencies between services, identify perfor-

mance bottlenecks, and understand how requests propagate across microservices architectures.

Although each of these data sources provides useful information individually, their combined analysis significantly enhances the ability to understand complex system behavior. When logs, metrics, and traces are correlated and analyzed together, engineers gain a comprehensive view of system operations.

Observability tools are designed to support queries across high-cardinality and high-dimensional data. Cardinality refers to the uniqueness of values within a dataset. High-cardinality fields contain a large number of distinct values, such as request identifiers, user identifiers, or container IDs. These fields are particularly valuable when investigating specific events or isolating anomalies within large datasets.

Dimensionality, on the other hand, refers to the number of attributes associated with each event. Telemetry data in observable systems often contains many dimensions, allowing engineers to explore relationships between multiple variables. The greater the dimensionality of the data, the richer the context available for debugging and system analysis.

In modern distributed systems, the potential combinations of failure conditions are virtually unlimited. Consequently, limiting telemetry data to only a few predefined attributes can significantly reduce the effectiveness of observability. For this reason, modern observability practices encourage the collection of rich contextual data for each event, enabling engineers to perform flexible and iterative investigations when diagnosing system issues.

While logs, metrics, and traces represent the core pillars of observability, each of them presents inherent limitations when used in isolation. Logs provide detailed information but can be difficult to analyze at scale. Metrics are efficient for monitoring trends but often lack contextual detail. Traces offer insights into request flows but may introduce overhead in highly distributed systems.

For this reason, modern observability systems rely on the combined use of these data sources, balancing their strengths and limitations depending on the specific requirements of the system.

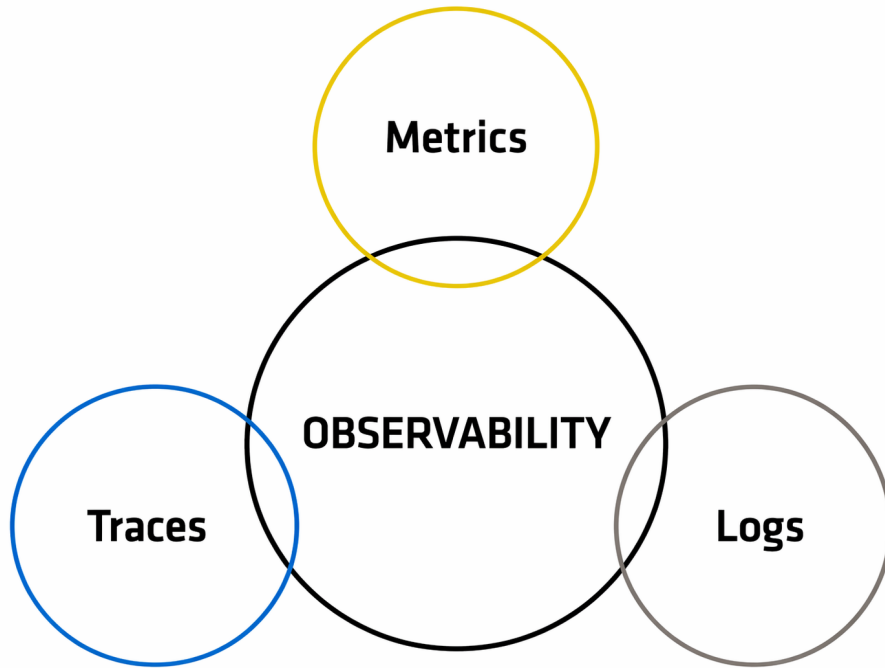


Figure 2: Observability pillars.

2.3 First Principles

A fundamental concept in observability is the practice of debugging from first principles. In observability, the concept of first principles refers to analyzing system behavior based on directly observable data rather than assumptions about expected behavior.

Debugging from first principles involves a scientific approach to understanding system behavior. Engineers begin by identifying observable signals that indicate a potential problem and then formulate hypotheses based on the available evidence. These hypotheses are validated or rejected through further observation and analysis of system telemetry data.

A common investigative methodology used in observability is the core analysis loop. This iterative process begins when a potential issue is detected, either through an alert or through user reports. The first step is to examine the overall system behavior and identify any deviations from expected performance patterns.

Once an anomaly is detected, engineers analyze the telemetry data to determine which dimensions may be responsible for the observed behavior. This may

involve examining sample events, grouping data across various attributes, or filtering specific subsets of data to isolate relevant patterns.

If the analysis provides sufficient information to explain the issue, the investigation can conclude. Otherwise, engineers refine their queries and continue exploring additional dimensions until the root cause is identified. This iterative process allows teams to progressively narrow down the scope of investigation and focus on the most relevant signals.

The effectiveness of this approach depends heavily on the availability of rich telemetry data. Without sufficient context or dimensionality, it becomes difficult to explore alternative hypotheses or isolate specific conditions responsible for anomalies. For this reason, observability tools aim to automate much of the analytical process, enabling engineers to quickly identify meaningful signals within large volumes of operational data.

By supporting this systematic and exploratory approach to debugging, observability enables teams to diagnose complex issues even when they have limited prior knowledge of the system.

2.4 Observability Characteristics

Observability provides several important benefits that directly impact the reliability and efficiency of software systems. One of the most significant advantages is the ability to reduce the time required to detect and resolve incidents.

In addition to improving incident response, observability also contributes to overall system stability. By enabling proactive analysis of telemetry data, engineers can identify performance bottlenecks and potential failures before they escalate into critical incidents.

Another important characteristic of observability is its role as an ongoing engineering practice. Achieving observability is not a one-time effort but rather a continuous process that involves instrumenting applications, collecting telemetry data, and analyzing system behavior throughout the software lifecycle.

Effective observability requires collaboration between development and operations teams. Engineers must ensure that applications generate meaningful

telemetry data and that this data can be analyzed efficiently using appropriate tools and platforms. Modern instrumentation frameworks, such as OpenTelemetry, have emerged as industry standards for collecting telemetry data across different services and platforms.

In addition to technical capabilities, observability also influences organizational culture. By providing visibility into system behavior, observability helps bridge the gap between development, operations, and business teams. Engineers gain a deeper understanding of how their software behaves in production, while business stakeholders can better evaluate the impact of system performance on user experience and business outcomes.

Ultimately, the goal of observability is to enable engineering teams to understand, operate, and improve complex systems effectively. By supporting exploratory analysis and real-time investigation, observability tools empower teams to respond to incidents more efficiently and continuously improve system reliability.

Despite its advantages, observability also introduces several challenges. The collection and storage of high-dimensional telemetry data can lead to increased resource consumption and system complexity. Additionally, designing effective observability systems requires careful instrumentation and may involve significant overhead in both development and operations.

2.5 Observability Maturity Model

The Observability Maturity Model (OMM) provides a framework for evaluating how effectively an organization adopts observability practices. Maturity models are commonly used to assess the capabilities of organizations in specific domains by defining progressive levels of development and associated best practices.

The concept of maturity models originates from the Capability Maturity Model (CMM), which was developed to evaluate the effectiveness of software development processes. Similar approaches have since been applied to many areas of organizational practice, including observability.

The OMM focuses on both technical and organizational aspects of observability adoption. It evaluates how engineering teams instrument their systems, analyze telemetry data, and integrate observability practices into their development workflows.

One of the key purposes of the model is to help organizations identify gaps in their current capabilities. By comparing existing practices against the maturity model, teams can determine which areas require improvement and prioritize initiatives that will have the greatest impact on system reliability and operational efficiency.

High-performing engineering teams typically demonstrate strong capabilities in several areas. These include the ability to respond effectively to system failures, deliver high-quality software releases, manage system complexity, maintain predictable deployment processes, and understand how users interact with their applications.

It is important to recognize that maturity models should not be interpreted as rigid frameworks. Every organization operates under different constraints, technologies, and business requirements. Therefore, the OMM should be used as a guideline rather than a prescriptive solution. Organizations are encouraged to adapt the model to their specific needs and continuously refine their observability practices as new insights and technologies emerge.

By adopting a structured approach to observability maturity, organizations can progressively improve their ability to monitor, analyze, and understand complex systems, ultimately leading to more reliable software and better user experiences.

3 Observability and Monitoring

Modern software systems operate in highly dynamic and distributed environments. Modern software systems operate in distributed environments composed of multiple interacting services across cloud, containerized, and edge infrastructures. In such contexts, maintaining system reliability and understanding system behavior requires advanced techniques for collecting, analyzing, and interpreting operational data.

Historically, monitoring represented the primary method used to observe system performance and detect failures. Monitoring systems typically relied on predefined metrics, alerts, and thresholds designed to identify known failure conditions. While this approach remains essential for ensuring system health, it is often insufficient when dealing with complex distributed architectures where

unexpected interactions between components may lead to unpredictable system states.

Observability has emerged as a complementary concept that extends the capabilities of traditional monitoring. Rather than focusing solely on predefined metrics and alerts, observability enables engineers to explore telemetry data in a flexible and investigative manner, supporting root-cause analysis in complex and distributed environments. By collecting rich contextual information such as logs, metrics, and traces, observability platforms allow engineers to understand how systems behave under both normal and abnormal conditions.

The relationship between monitoring and observability is therefore not one of replacement but rather one of integration. Monitoring provides structured mechanisms for detecting known issues, while observability enables engineers to investigate unknown problems and analyze complex system behaviors. Together, these approaches form the foundation for modern operational practices in software engineering.

This chapter examines the relationship between monitoring and observability, with particular focus on their role in the design of modern observability architectures. Particular attention is given to the role of dashboards, telemetry data analysis, and investigative methodologies used to diagnose system behavior. [4], [1], [2].

In this work, monitoring and observability are not treated as alternative approaches but as complementary components of the proposed architecture. Monitoring is used for real-time detection of known conditions, while observability supports root-cause analysis and investigation of unexpected system behaviors. This design choice reflects the need to balance operational simplicity with analytical flexibility in distributed IoT environments.

3.1 Observability vs Monitoring

Traditional monitoring approaches have historically relied on the use of metrics to understand the behavior of software systems. Historically, monitoring tools have relied primarily on metrics provided engineers with a way to track the performance of applications and infrastructure over time. Metrics represent numerical measurements that describe certain aspects of system behavior, such as CPU usage, memory consumption, or request latency.

These monitoring systems were originally designed under the assumption that software architectures were relatively simple and predictable. Engineers could anticipate many potential failure scenarios and therefore define specific metrics and alerts to detect known problems.

Monitoring systems typically collect and aggregate metrics over time, producing graphical representations of system performance through dashboards. These dashboards allow operators to visualize trends and detect anomalies by observing variations in the collected data.

However, traditional monitoring approaches present important limitations when applied to modern distributed systems. As software architectures become more complex and incorporate microservices, container orchestration platforms, and cloud-native technologies, it becomes increasingly difficult to predict all possible failure scenarios in advance.

Monitoring systems are inherently reactive, as they rely on predefined metrics and thresholds. While effective for detecting known issues, they provide limited support for diagnosing unexpected failures in complex distributed environments. In such situations, engineers must manually correlate information across multiple tools and data sources in order to diagnose the issue.

For this reason, modern software systems increasingly rely on observability practices, which aim to provide deeper insight into system behavior by collecting richer telemetry data and enabling more flexible exploration of system states.

Although monitoring and observability are closely related concepts, they serve different purposes within the context of system operations. To better understand the relationship between monitoring and observability, we can refer to figure [3](#).

Monitoring traditionally focuses on tracking predefined indicators of system health. Engineers define specific metrics and thresholds that represent acceptable operating conditions. When a metric exceeds its threshold, the monitoring system generates an alert that notifies operators of a potential problem. This approach is effective for identifying known failure scenarios and maintaining service availability.

Typical monitoring activities include tracking resource utilization such as CPU usage, memory consumption, network throughput, and response latency. Monitoring tools continuously collect these metrics and compare them against predefined thresholds to determine whether the system is functioning correctly.

However, monitoring systems are inherently limited by the assumptions made during their configuration. Because monitoring relies on predefined metrics and alerts, it can only detect conditions that engineers anticipated in advance. When unexpected failures occur, monitoring systems may provide limited information about the underlying cause of the issue.

Observability addresses this limitation by enabling engineers to investigate system behavior in a more flexible and exploratory manner. Instead of relying solely on predefined metrics, observability platforms collect high-dimensional telemetry data that can be queried dynamically. Engineers can analyze logs, metrics, and traces across multiple dimensions, allowing them to explore relationships between events and identify patterns that were not anticipated during system design.

In this sense, observability extends the capabilities of monitoring by supporting investigative workflows. When a monitoring alert indicates that something is wrong, observability tools allow engineers to analyze the available data in order to determine why the issue occurred.

Therefore, while monitoring answers questions such as “Is the system operating within acceptable limits?”, observability focuses on questions such as “Why is the system behaving this way?”. Both perspectives are essential for maintaining reliable systems in modern distributed environments.

3.2 Correlation between Monitoring and Observability

Monitoring and observability should not be considered mutually exclusive concepts. Instead, they represent complementary approaches that together provide a comprehensive understanding of system behavior.

Monitoring typically acts as the first line of defense in operational environments. By continuously evaluating predefined metrics and service-level indicators, monitoring systems are able to detect deviations from expected performance and trigger alerts when anomalies occur. These alerts notify engineers that an issue may be affecting system performance or availability.

Once a problem has been detected, observability tools play a critical role in identifying the root cause of the issue. Engineers can use observability platforms to analyze telemetry data in greater depth, exploring correlations between



Figure 3: Comparison between monitoring and observability.

different signals and investigating the sequence of events that led to the failure.

For example, a monitoring system may detect an increase in response latency for a web service. While this alert indicates that a problem exists, it does not necessarily explain why the latency increased. Observability tools enable engineers to analyze request traces, examine log entries, and correlate these signals with infrastructure metrics in order to determine whether the issue was caused by a database bottleneck, network congestion, or a malfunctioning service component.

The effectiveness of this investigative process depends on the availability of rich and well-structured telemetry data. Observability platforms therefore emphasize the collection of high-cardinality data that includes detailed contextual information about system events. This allows engineers to explore multiple dimensions of system behavior and identify patterns that may not be immediately visible through aggregated metrics alone.

In practice, monitoring and observability form a continuous operational cycle. Monitoring detects potential issues, while observability enables deeper analysis and root-cause identification. This integration enhances incident response

capabilities and improves the efficiency of root-cause analysis.

In the context of this project, this complementary relationship guided the design of the monitoring architecture. Monitoring mechanisms were used for real-time detection of known issues, while observability tools were integrated to support deeper analysis of telemetry data and investigation of unexpected system behavior.

3.3 Dashboards and Data Analysis

Dashboards play a central role in both monitoring and observability by providing visual representations of telemetry data. Through dashboards, engineers can quickly assess the current state of a system and identify potential anomalies in performance metrics.

Monitoring dashboards typically focus on key performance indicators and service-level objectives. These dashboards present aggregated metrics such as request rates, error rates, and response latency, allowing operators to evaluate system health at a glance. By organizing metrics into visual components such as graphs, charts, and gauges, dashboards make it easier to detect trends and deviations from normal behavior.

In addition to their role in monitoring, dashboards are also essential tools for exploratory analysis within observability platforms. Engineers can use dashboards to examine telemetry data across multiple dimensions, filtering and grouping information in order to identify correlations between events. This capability is particularly valuable when investigating complex incidents involving multiple services or infrastructure components.

Advanced observability platforms allow dashboards to support interactive exploration of telemetry data. Engineers can drill down into specific time intervals, analyze individual request traces, or correlate log entries with performance metrics. These capabilities transform dashboards from simple monitoring interfaces into powerful analytical tools for understanding system behavior.

The effectiveness of dashboards depends largely on how telemetry data is collected and structured. High-quality dashboards require well-defined metrics, consistent labeling, and accurate timestamps in order to ensure that data from different sources can be correlated effectively.

When properly designed, dashboards provide both operational visibility and analytical insight, enabling engineers to detect anomalies, investigate incidents, and understand the overall performance of complex distributed systems.

While dashboards are effective for providing a high-level overview of system behavior, they present several limitations. They rely on aggregated metrics, which may obscure important details, and on predefined queries, which restrict exploratory analysis. As a result, engineers often need to switch between multiple tools to investigate complex issues, increasing analysis time and complexity.

Observability platforms aim to address these limitations by enabling more flexible exploration of telemetry data. Instead of relying solely on static dashboards, engineers can query logs, metrics, and traces dynamically, allowing them to investigate unexpected behaviors and follow the data toward the root cause of system anomalies.

For this reason, in this project dashboards were designed not only for visualization purposes but also as entry points for further analysis. The system integrates dashboards with log and metric exploration tools, allowing operators to move from high-level monitoring views to detailed investigation workflows.

3.4 System Analysis

System analysis represents a fundamental aspect of observability, as it involves the investigation of system behavior in order to understand the causes of anomalies and failures. In modern distributed environments, systems may consist of dozens or even hundreds of interconnected services, making it difficult to diagnose issues using traditional debugging techniques.

Observability supports system analysis by enabling engineers to examine telemetry data from multiple perspectives. By correlating logs, metrics, and traces, engineers can reconstruct the sequence of events that occurred within the system and identify the components responsible for a particular failure.

This investigative process often follows an iterative methodology. Engineers begin by identifying a symptom, such as increased latency or error rates. They then analyze relevant telemetry data to determine which components may be contributing to the observed behavior. Additional queries and filters are applied to narrow down the scope of the investigation until the root cause of the issue

is identified.

A key advantage of observability lies in its ability to enable engineers to perform this analysis even when the failure scenario was not anticipated in advance. Because telemetry data includes detailed contextual information, engineers can explore different hypotheses and test them against the available data.

System analysis also benefits from the ability to correlate events across different layers of the infrastructure stack. For example, an application performance issue may be related to container resource limitations, network congestion, or misconfigured services. Observability tools enable engineers to analyze telemetry data from all these layers simultaneously, providing a holistic view of system behavior.

By supporting comprehensive system analysis, observability enables organizations to diagnose complex incidents more efficiently and continuously improve system reliability.

While these principles apply broadly to distributed systems, they become particularly relevant in IoT Edge environments, where additional constraints must be considered.

3.5 Challenges of Observability in IoT Edge Systems

Observability in IoT Edge environments introduces several additional challenges compared to traditional cloud-based systems. While modern observability tools are often designed for centralized infrastructures such as data centers or cloud-native platforms, IoT systems operate in distributed environments characterized by heterogeneous devices, constrained resources, and complex communication pipelines. These characteristics significantly influence the design and implementation of monitoring solutions.

One of the primary challenges arises from the distributed nature of IoT infrastructures. In traditional cloud environments, most services run within centralized clusters where monitoring tools can easily access system metrics and logs. In contrast, IoT systems consist of numerous geographically distributed devices that generate telemetry data at the edge of the network. These devices often communicate through intermediate edge nodes that perform local processing before forwarding data to cloud services. As a result, observability mechanisms

must operate across multiple layers of the infrastructure, including devices, edge nodes, and cloud platforms.

Another challenge concerns the heterogeneity of the monitored components. IoT ecosystems frequently involve devices produced by different manufacturers, each using distinct communication protocols and data formats. Industrial environments, for instance, often rely on protocols such as OPC-UA, Modbus, or MQTT to transmit machine telemetry. This diversity makes it difficult to design a unified monitoring approach capable of collecting and analysing telemetry data from all components in a consistent manner.

Resource constraints also play a critical role in the design of observability solutions for IoT environments. Many edge devices operate with limited computational resources, restricted memory availability, and constrained network bandwidth. Deploying heavy monitoring agents on such devices may significantly impact their performance or even interfere with their primary operational tasks. For this reason, monitoring solutions for edge systems must be designed to minimize overhead while still providing sufficient visibility into system behavior.

Another important difficulty concerns data volume and telemetry management. IoT devices can generate large volumes of telemetry data, especially when monitoring industrial processes or high-frequency sensor measurements. Collecting and transmitting all available telemetry data to centralized monitoring systems may not always be feasible due to network bandwidth limitations. Consequently, edge systems often implement local processing mechanisms that filter, aggregate, or enrich telemetry data before forwarding it to cloud infrastructures.

Latency is another factor that must be considered when designing observability solutions for edge environments. In many industrial scenarios, the ability to detect anomalies or operational issues in near real time is crucial for maintaining system reliability and safety. However, if telemetry data must travel through multiple network layers before being analyzed, delays may occur that reduce the effectiveness of monitoring systems. For this reason, modern observability architectures frequently combine edge-level monitoring with cloud-based analysis in order to balance responsiveness and analytical capabilities.

Security considerations also represent an important aspect of IoT observability. Telemetry data may contain sensitive operational information related to industrial processes or system configurations. Ensuring that monitoring pipelines handle this information securely is essential for protecting both operational

integrity and organizational data assets. Secure communication channels, access control mechanisms, and proper data handling procedures must therefore be integrated into the observability infrastructure.

Finally, another challenge involves the correlation of heterogeneous telemetry signals. Effective observability requires the ability to analyze logs, metrics, and traces together in order to understand complex system behaviors. In distributed IoT environments, however, telemetry data may originate from multiple independent components operating asynchronously. Correlating these signals in a meaningful way requires careful design of monitoring pipelines and consistent use of timestamps, metadata, and contextual information.

Despite these challenges, the adoption of modern observability tools and architectures makes it possible to build effective monitoring solutions even in complex IoT Edge environments. By combining lightweight telemetry collection mechanisms at the edge with scalable cloud-based analysis platforms, it becomes possible to achieve a balance between system visibility, performance efficiency, and operational scalability.

These challenges directly influenced the architectural choices of this project. In particular, a hybrid approach was adopted, combining local data processing at the edge with centralized analysis in the cloud. This design allows reducing network overhead while maintaining the ability to perform advanced analytics on aggregated telemetry data.

3.6 Monitoring Approaches

Monitoring strategies have evolved significantly as software architectures have become more distributed and dynamic. Traditional monitoring approaches focused primarily on infrastructure-level metrics such as server health, network availability, and hardware performance. While these metrics remain important, modern systems require more sophisticated monitoring techniques that consider application behavior and user experience.

One widely adopted approach involves the use of service-level objectives (SLOs) and service-level indicators (SLIs). These metrics focus on measuring the quality of service delivered to users, rather than simply monitoring infrastructure performance. Examples include request success rates, response latency, and system availability. By defining acceptable thresholds for these indicators,

organizations can ensure that their systems meet reliability requirements.

Another important monitoring strategy involves anomaly detection. Instead of relying solely on static thresholds, anomaly detection systems analyze historical data to identify patterns of normal behavior. When current metrics deviate significantly from these patterns, the monitoring system generates alerts that indicate potential issues.

In modern cloud-native environments, monitoring approaches also integrate closely with observability platforms. Telemetry data collected through instrumentation frameworks is used not only for monitoring but also for deeper analysis and debugging. This integration allows organizations to maintain continuous visibility into system performance while retaining the ability to investigate complex incidents when they occur.

Ultimately, effective monitoring strategies must balance proactive detection of known issues with the flexibility required to analyze unexpected behaviors. By combining monitoring techniques with observability practices, organizations can build resilient systems capable of operating reliably in complex and rapidly evolving environments.

When designing an observability platform, organizations often face an important architectural decision: whether to build a custom monitoring solution or adopt an existing one. This decision is commonly referred to as the “build versus buy” dilemma.

The choice between building or purchasing an observability solution is rarely binary. In many cases, organizations adopt a hybrid approach that combines existing tools with internally developed components. Several factors must be considered when making this decision, including financial costs, development effort, long-term maintenance, and the expected return on investment.

Building a custom observability solution can offer several advantages. By developing internal tools, organizations gain the ability to tailor the observability platform to their specific infrastructure, workflows, and operational requirements. Custom solutions can be designed to integrate deeply with existing systems and internal practices, allowing teams to leverage their institutional knowledge and adapt the monitoring logic to the specific characteristics of their environment.

However, building an observability platform internally also introduces several challenges. Custom-built systems require continuous development, maintenance,

and operational support. As infrastructures evolve, the monitoring platform must also evolve accordingly. Updates to underlying technologies, infrastructure changes, and new monitoring requirements often require additional development work. If these systems are not maintained properly, they may eventually become outdated or difficult to operate.

On the other hand, purchasing a ready-made observability solution allows organizations to significantly reduce the time required to deploy a monitoring system. Commercial observability platforms typically provide preconfigured dashboards, data pipelines, and user interfaces that allow engineers to begin monitoring their systems within a very short time. This faster time-to-value represents one of the most significant advantages of adopting existing platforms.

Nevertheless, commercial solutions may also introduce certain limitations. Pricing models based on data ingestion, storage, or query execution can sometimes discourage extensive exploration of observability data. In observability practices, however, it is often important to analyze telemetry data with increasing curiosity and flexibility in order to understand unexpected system behavior.

For this reason, many modern monitoring architectures adopt a hybrid approach that combines open-source tools with internally developed components. This strategy allows organizations to benefit from the flexibility of open-source monitoring ecosystems while still developing custom logic that addresses specific analytical requirements. Figure 4 provides a summary of the build-versus-buy concept.

In this project, a hybrid approach was adopted. Open-source tools such as Prometheus, Grafana, and Loki were selected to leverage existing, well-supported technologies, while custom configurations and integrations were developed to adapt the monitoring platform to the specific requirements of the IoT Edge environment. This approach balances flexibility, cost efficiency, and development effort.

4 Telemetry Data in Observability

Telemetry data represents the primary source of information used to understand the behavior of modern software systems. In distributed architectures, applications generate continuous streams of operational data describing internal

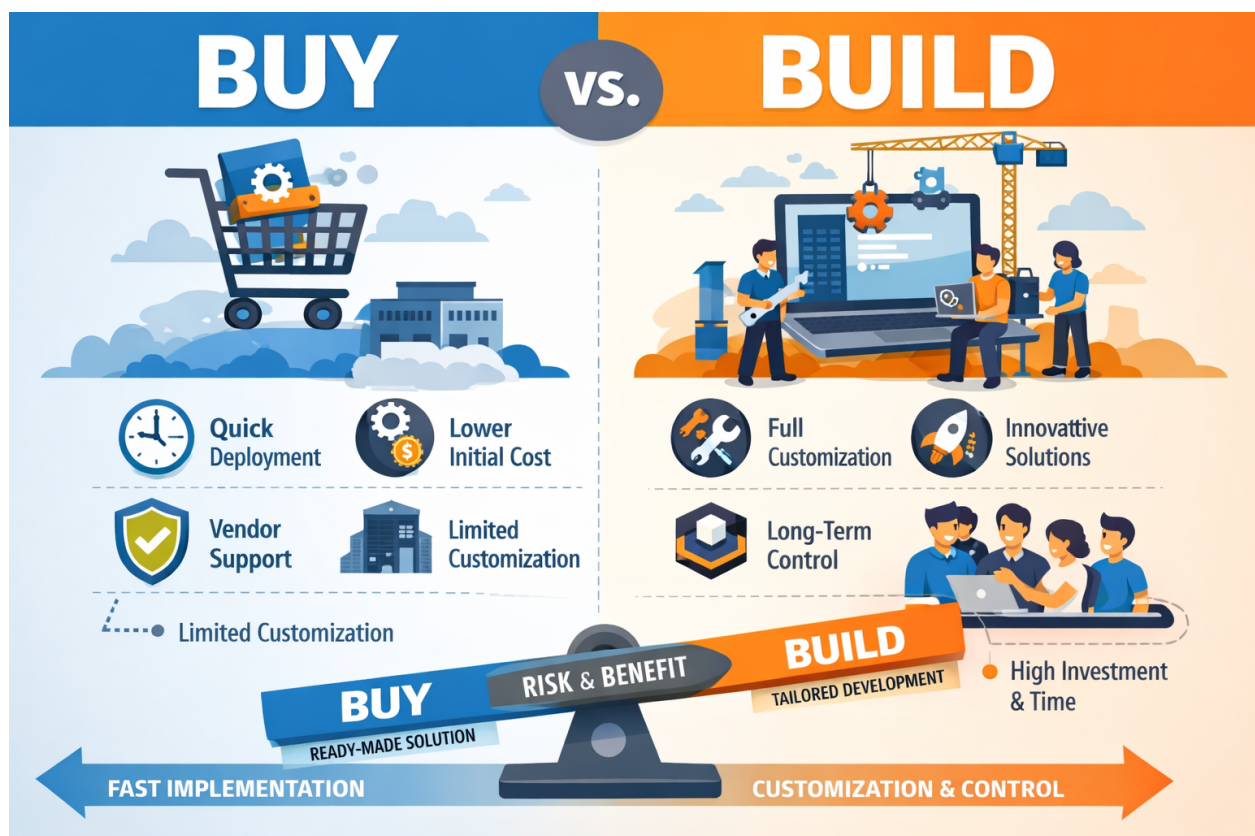


Figure 4: Buy or build schema.

processes, system interactions, performance characteristics, and error conditions. This data enables observability by providing visibility into system behavior and supporting the analysis of production environments.

In the context of observability, telemetry refers to the automatic collection and transmission of operational data from software systems to external analysis platforms. These data are generated by applications, infrastructure components, containers, and network services, and can be collected, aggregated, and analyzed to provide insights into system behavior.

Observability platforms rely on telemetry data to reconstruct system behavior and identify patterns that may indicate anomalies or performance issues. By analyzing telemetry signals, engineers can detect deviations from expected system behavior, investigate the causes of failures, and evaluate the performance of different components within the infrastructure.

Several different types of telemetry data are used in observability systems. These include events, metrics, logs, and traces, each of which provides a different perspective on system behavior. Although these signals serve different purposes, they share a common objective: to provide meaningful insights into how a system operates under various conditions.

The following sections examine the main categories of telemetry data used in observability systems, focusing on events and metrics, data structure formats, time-series data representation, and sampling techniques used to manage large volumes of telemetry information.

In this project, telemetry data plays a central role in enabling the monitoring and observability infrastructure. Different types of telemetry signals were collected from distributed components, including containers and edge nodes, and integrated into a unified data pipeline to support both real-time monitoring and post hoc analysis.

4.1 Events and Metrics

Events and metrics represent two fundamental categories of telemetry data used to describe system behavior.

An event is defined as a discrete occurrence that takes place within a system at

a specific moment in time. Events often correspond to significant system actions such as user requests, service calls, configuration changes, or error conditions. Each event typically contains contextual information describing what happened, when it occurred, and which components were involved.

Events provide detailed contextual information that allows engineers to reconstruct system behavior and understand interactions between components during execution.

In contrast, metrics represent numerical measurements that describe the state or performance of a system over time. Metrics are commonly used to track quantitative indicators such as CPU utilization, memory consumption, request latency, throughput, and error rates. These measurements provide a high-level overview of system behavior and are frequently used in monitoring dashboards to evaluate system performance.

While events focus on capturing individual occurrences, metrics emphasize aggregated measurements that summarize system activity across time intervals. This distinction makes metrics particularly useful for identifying trends and detecting performance degradation.

In practice, events and metrics are often used together to provide a comprehensive understanding of system behavior. Metrics can reveal that a performance issue exists, while event data can help explain why the issue occurred. For example, an increase in request latency detected through metrics may correspond to specific error events recorded in application logs.

Because of their complementary roles, modern observability platforms integrate both events and metrics into unified telemetry pipelines. This integration allows engineers to correlate high-level performance indicators with detailed operational events, enabling more effective debugging and system analysis.

In the context of this project, metrics were primarily used for continuous monitoring and performance evaluation, while event-based data and logs were used to support detailed analysis and debugging activities. This combination allows detecting anomalies through aggregated indicators and investigating their causes through detailed event data.

4.2 Structured and Unstructured Data

Telemetry data can be represented in different formats depending on how the information is generated and stored. Telemetry data can be broadly classified into two categories: structured data and unstructured data.

Structured data follows a predefined schema that defines how information is organized. Each data record contains clearly defined fields with consistent formats, making it easier to store, process, and analyze the data programmatically. Examples of structured telemetry data include metrics collected by monitoring systems and logs formatted using standardized schemas such as JSON.

Structured data offers several advantages for observability systems. Because the structure of the data is known in advance, it becomes easier to perform automated queries, filtering operations, and statistical analysis. Structured data also supports efficient indexing and storage, which is particularly important when dealing with large volumes of telemetry information.

Unstructured data, in contrast, does not follow a rigid schema. It often consists of free-text messages or loosely formatted information generated by applications during runtime. Traditional log messages are a common example of unstructured data, as they may contain human-readable descriptions of events without strict formatting rules.

While unstructured data provides valuable contextual information, it can be more difficult to analyze automatically. Engineers may need to apply parsing techniques or pattern-matching algorithms to extract meaningful information from these data sources.

For this reason, modern observability practices increasingly encourage the use of structured logging. By adopting standardized log formats, organizations can improve the ability of observability platforms to process and correlate telemetry data across different services and components.

Nevertheless, both structured and unstructured data play important roles in observability. Structured data supports efficient analysis and automation, while unstructured data often provides rich contextual descriptions that are essential for understanding complex system behavior.

For this reason, the monitoring architecture developed in this project prioritizes structured telemetry data whenever possible. In practice, telemetry is handled

in structured formats to improve consistency, enable automated processing, and facilitate correlation across distributed system components.

4.3 Time-Series Data

Time-series data represents one of the most important data formats used in monitoring and observability systems. A time-series dataset consists of a sequence of data points recorded at successive time intervals, where each data point represents the value of a particular metric at a specific moment in time.

Many operational metrics inherently follow a time-series structure. Examples include CPU usage measured every second, the number of requests processed per minute, or the response time of an application over a defined time interval. By recording these measurements continuously, observability platforms can track how system performance evolves over time.

Time-series data provides several advantages for system monitoring and analysis. First, it allows engineers to visualize trends and patterns in system behavior. Graphical representations of time-series data, such as line charts and heat maps, make it possible to identify anomalies, spikes, or gradual performance degradation.

Second, time-series analysis supports the detection of correlations between different metrics. For example, an increase in response latency may correspond to a spike in CPU utilization or network traffic. By analyzing multiple time-series signals together, engineers can gain deeper insights into the relationships between different system components.

Another important feature of time-series data is the ability to perform historical analysis. By storing telemetry data over extended periods, organizations can analyze long-term trends, evaluate system scalability, and plan infrastructure capacity more effectively.

However, managing time-series data also introduces challenges related to storage efficiency and data volume. Modern distributed systems can generate millions of metric samples per second, requiring specialized databases and storage systems optimized for time-series workloads. Technologies such as Prometheus, InfluxDB, and other time-series databases have been developed specifically to handle these requirements.

In observability platforms, time-series data serves as a foundation for monitoring dashboards, alerting systems, and performance analysis tools, making it an essential component of modern operational infrastructures.

Time-series data representation was a key design choice in this project, as it aligns with the requirements of modern monitoring systems such as Prometheus. The use of time-series metrics enables efficient storage, real-time querying, and integration with visualization tools for performance analysis.

4.4 Sampling and Telemetry

One of the primary challenges in observability systems is the large volume of telemetry data generated by modern applications. Distributed systems composed of numerous services may produce enormous amounts of logs, metrics, and traces every second. Processing and storing all this information without optimization can lead to significant performance and storage overhead.

Sampling techniques are commonly used to address this challenge by reducing the amount of telemetry data that needs to be collected and analyzed. Sampling involves selecting a representative subset of events or traces rather than capturing every individual occurrence.

Several sampling strategies can be applied in observability systems. One common approach is probabilistic sampling, where only a certain percentage of events are recorded. Another strategy involves adaptive sampling, where the sampling rate dynamically adjusts based on system conditions or detected anomalies.

Sampling is particularly important in distributed tracing systems, where each request may generate multiple spans across different services. Without sampling, the amount of trace data produced by high-traffic applications could become unmanageable. [5].

Although sampling reduces the volume of telemetry data, it must be applied carefully to ensure that important information is not lost. Effective sampling strategies aim to preserve representative system behavior while minimizing the storage and computational costs associated with processing telemetry data.

Modern observability frameworks often include built-in mechanisms for managing telemetry pipelines, allowing engineers to configure sampling policies, filter

specific events, and control the flow of data across different components of the observability stack.

By combining efficient telemetry collection with intelligent sampling strategies, organizations can maintain comprehensive visibility into system behavior while ensuring that observability infrastructures remain scalable and cost-effective.

In this project, sampling strategies were considered as a trade-off between data completeness and system efficiency. While collecting all telemetry data would provide maximum visibility, it would also introduce significant storage and processing overhead. Therefore, selective data collection and filtering mechanisms were adopted to balance observability capabilities with resource constraints, particularly in edge environments.

5 Observability Signals

Modern distributed systems generate large volumes of telemetry data that provide insight into system behavior, performance, and failures. Observability leverages this data to enable engineers to analyze system states, detect anomalies, and investigate unexpected issues in complex environments.

In contemporary cloud-native infrastructures, observability typically relies on three main categories of telemetry signals: logs, metrics, and traces. These signals provide complementary perspectives on system behavior. Metrics represent numerical measurements describing system performance over time, logs record discrete events generated by applications or infrastructure components, and traces allow operators to follow the execution path of requests as they travel across multiple services. [6].

By combining these telemetry signals, it enables a comprehensive understanding of the operational state of distributed systems. Each signal type provides different levels of granularity and context, enabling system administrators to investigate issues from multiple perspectives.

Within the observability platform implemented in this project, telemetry data is collected from containerized services and processed using a set of specialized monitoring tools that support the analysis and visualization of system activity. The observability pipeline integrates log aggregation systems, metrics monitoring

platforms, and visualization dashboards to support the analysis of system behavior.

The choice of these tools is motivated by their compatibility with containerized environments, scalability, and strong integration within the cloud-native ecosystem. This design allows efficient collection and correlation of telemetry data across distributed components.

The observability architecture relies on open-source technologies designed to operate efficiently within containerized environments. These technologies enable the automated collection of telemetry signals and provide mechanisms for storing, querying, and visualizing operational data.

The following sections describe the three primary observability signals in more detail and explain their role within the monitoring infrastructure implemented for the project.

5.1 Logs

Logs represent one of the most traditional and widely used forms of telemetry data. A log entry is a textual record generated by a software component to describe events that occur during system execution. These events may include application activities, error messages, system warnings, or informational notifications related to internal operations.

Logs are typically produced by applications, operating systems, middleware components, and infrastructure services. Each log entry usually contains a timestamp, a severity level, and a message describing the event that occurred. Additional contextual information may also be included, such as identifiers of the components involved in the event or metadata describing the execution environment.

One of the main advantages of logs is that they provide detailed insights into system behavior. While metrics offer aggregated numerical values, logs often contain descriptive information that explains what exactly happened within a system component. This makes logs particularly useful for debugging and incident analysis.

In distributed environments, however, managing logs becomes significantly more

complex. Modern microservice architectures may consist of dozens or even hundreds of services running simultaneously across multiple containers or virtual machines. Each service generates its own logs, resulting in large volumes of log data that must be collected and analyzed efficiently.

To address this challenge, centralized log aggregation systems are commonly used. These systems collect logs from multiple sources and store them in a centralized repository where they can be indexed and queried. Centralized log management simplifies the process of analyzing system events and allows operators to investigate incidents more effectively.

Within the monitoring infrastructure developed in this project, log collection is performed using Promtail, a log shipping agent designed to work with containerized environments. Promtail continuously monitors log files generated by containers and forwards the collected log entries to a centralized log storage system.

Promtail is particularly well suited for container-based environments because it can automatically detect running containers and associate log entries with metadata such as container identifiers, service names, and labels. This metadata allows log entries to be organized and queried more efficiently within the centralized log storage system.

Once collected, logs are stored in Loki, a log aggregation system designed specifically for large-scale distributed environments. Loki is optimized for storing and querying log data efficiently while minimizing storage costs. Unlike traditional log indexing systems, Loki focuses on storing log streams associated with metadata labels, which significantly reduces the amount of indexing required.

The adoption of Loki instead of traditional full-text indexing systems is motivated by its label-based architecture, which reduces storage overhead and improves query performance in containerized environments. This design aligns well with the needs of scalable and resource-efficient monitoring systems.

The integration between Promtail and Loki forms a log management pipeline capable of collecting and storing large volumes of log data generated by containerized services. Operators can then query these logs using specialized query languages and visualize them through dashboard platforms such as Grafana.

Visualization tools play an important role in making log data accessible and useful. Grafana allows users to create dashboards that integrate logs with

other telemetry signals such as metrics. This integration makes it possible to correlate log events with system performance indicators and investigate issues more effectively.

For example, if a monitoring dashboard indicates an increase in application error rates, operators can immediately inspect the corresponding logs to identify the root cause of the problem. By examining log entries generated during the time interval in which the anomaly occurred, it becomes possible to determine which component generated the error and what conditions led to the failure.

Another advantage of centralized log systems is the ability to perform historical analysis. Logs stored in the aggregation platform can be queried even long after the events occurred, allowing system administrators to reconstruct incidents and analyze system behavior over time.

In summary, logs represent a fundamental component of observability systems. They provide detailed information about system events and play a crucial role in debugging and troubleshooting complex software infrastructures.

5.2 Metrics

Metrics represent numerical measurements that describe system behavior over time. Unlike logs, which record discrete events, metrics provide quantitative information about system performance and resource utilization.

Metrics are typically collected at regular intervals and stored as time-series data. Each metric consists of a sequence of numerical values associated with timestamps, allowing system operators to observe how specific indicators evolve over time. This makes metrics particularly effective for identifying trends, detecting anomalies, and monitoring the overall health of a system.

In modern monitoring infrastructures, metrics can be generated from multiple sources. These sources include operating systems, container runtimes, application services, and network components. Each source produces different types of metrics that contribute to a comprehensive view of system performance.

System-level metrics describe the utilization of hardware resources such as CPU, memory, disk storage, and network bandwidth. Monitoring CPU usage helps identify processes that consume excessive computational resources, while

memory metrics allow operators to detect memory leaks or inefficient memory allocation patterns.

Disk-related metrics provide information about storage usage and input/output operations. Monitoring disk utilization ensures that sufficient storage capacity is available and helps identify potential bottlenecks caused by high I/O activity.

Network metrics describe the volume of data transmitted between system components and the latency of communication channels. In distributed systems, network performance plays a critical role in determining application responsiveness and reliability.

Application-level metrics provide insights into the internal behavior of software services. Examples include the number of requests processed by an application, response times for client requests, and error rates associated with service operations. These metrics allow system administrators to evaluate application performance and detect potential service degradation.

Within the monitoring infrastructure implemented in this project, metrics are collected and managed using Prometheus, an open-source monitoring system designed for cloud-native environments.

Prometheus periodically retrieves metrics from monitored services through HTTP endpoints that expose telemetry data in a standardized format. This architecture follows a pull-based model, in which the monitoring system actively queries monitored components rather than relying on them to push telemetry data.

Once collected, metrics are stored within the Prometheus time-series database. This database is optimized for storing large volumes of time-series data and allows efficient retrieval of historical metrics.

Prometheus was selected due to its pull-based model, which simplifies service discovery and integration in dynamic environments such as containers. Additionally, its native support for time-series data and powerful query language make it particularly suitable for real-time monitoring and analysis.

Prometheus also provides a powerful query language known as PromQL, which enables users to perform advanced analysis on collected metrics. PromQL supports aggregation functions, mathematical operations, and statistical calculations that allow operators to analyze system behavior in detail.

These capabilities make metrics an essential component of monitoring infrastructures. By analyzing metrics data, operators can evaluate system performance, detect anomalies, and implement automated alerting mechanisms that notify administrators when specific conditions occur.

5.3 Traces

Traces represent another important category of observability signals and are particularly useful in distributed systems composed of multiple interacting services. While logs provide detailed event records and metrics offer numerical measurements, traces allow operators to follow the path of individual requests as they propagate across different components of a system.

In microservice architectures, a single user request may trigger interactions between many different services. Each service may perform a portion of the overall operation before forwarding the request to another component. Understanding how requests travel through these services is essential for diagnosing performance issues and identifying bottlenecks.

Tracing systems address this challenge by recording information about each step involved in processing a request. A trace typically consists of multiple spans, where each span represents a specific operation performed by a service during the execution of the request.

Each span contains information such as the start time of the operation, its duration, and metadata describing the service that executed it. By linking spans together, tracing systems reconstruct the entire execution path of a request across the distributed infrastructure.

Tracing data provides valuable insights into system performance. For example, if a user request experiences high latency, traces can reveal which service contributed most to the delay. This allows system administrators to identify bottlenecks and optimize system performance.

Another advantage of distributed tracing is its ability to highlight dependencies between services. By analyzing traces, engineers can visualize the interactions between system components and better understand how the architecture behaves under real workloads.

Modern observability platforms often implement tracing using standardized telemetry frameworks such as OpenTelemetry. OpenTelemetry provides a unified framework for collecting telemetry data, including traces, metrics, and logs. By adopting a standardized instrumentation approach, developers can integrate observability capabilities into their applications more easily.

Although the primary focus of the monitoring infrastructure implemented in this project is on logs and metrics, the integration of tracing systems represents a natural extension of the observability platform. Combining traces with metrics and logs provides a more complete picture of system behavior and significantly improves the ability to diagnose complex issues in distributed environments.

When all three observability signals are available, operators can correlate them to investigate incidents more effectively. For example, an increase in response latency detected through metrics may correspond to specific error messages recorded in logs and to delays observed in traces that identify a bottleneck in a particular service.

This correlation between telemetry signals represents one of the key strengths of modern observability platforms. By integrating logs, metrics, and traces within a unified monitoring environment, engineers can achieve deeper visibility into system operations and respond more efficiently to operational issues.

By combining logs, metrics, and traces, observability platforms provide multiple perspectives on system behavior. Each signal type contributes different information that helps engineers diagnose issues more effectively.

In this project, tracing was not fully integrated due to the increased complexity and overhead associated with its implementation in edge environments. However, its inclusion represents a potential future improvement to enhance system observability and support more advanced performance analysis.

5.4 Data Sources

Observability signals originate from multiple sources within modern software infrastructures. These data sources generate telemetry information that describes system behavior at different levels of the technology stack.

Application services represent one of the primary sources of observability data.

Applications generate logs, metrics, and traces that describe internal operations, user interactions, and error conditions. Instrumentation libraries are often used to embed telemetry collection directly within application code.

Infrastructure components also contribute significant amounts of telemetry data. Operating systems, virtual machines, and container environments produce metrics describing resource utilization, process activity, and system performance. These metrics help engineers monitor the health of underlying infrastructure supporting application workloads.

Container orchestration platforms such as Kubernetes generate additional telemetry data related to container lifecycle events, resource scheduling, and network communication between services. Monitoring these signals is essential for understanding how applications behave within containerized environments.

Network components represent another important data source. Network telemetry provides insights into traffic patterns, latency, and connectivity issues between services. By analyzing network data alongside application telemetry, engineers can detect communication bottlenecks or infrastructure failures affecting system performance.

In modern observability architectures, telemetry data from all these sources is typically aggregated within centralized observability platforms. This aggregation enables engineers to analyze system behavior holistically and correlate events across multiple layers of the infrastructure stack. [6].

In the proposed architecture, particular emphasis is placed on container-level and application-level data sources, as they provide the most relevant information for analyzing system behavior in edge-based environments.

5.5 Open-source Tools

A wide range of open-source tools has been developed to support observability practices in modern software systems. These tools enable organizations to collect, process, store, and visualize telemetry data generated by applications and infrastructure components.

Log management platforms are commonly used to collect and analyze log data from distributed systems. Tools such as Elasticsearch, Logstash, and Kibana

are often deployed together as part of the Elastic Stack, providing scalable storage and powerful search capabilities for log analysis. Other platforms such as Grafana Loki provide log aggregation systems designed to integrate closely with modern observability dashboards.

For metrics collection and monitoring, Prometheus has become one of the most widely adopted open-source solutions. Prometheus collects time-series metrics from instrumented services and provides a flexible query language for analyzing system performance. The collected metrics can be visualized through dashboards using tools such as Grafana, which allows engineers to build interactive monitoring interfaces.

Distributed tracing systems also play an important role in observability infrastructures. Tools such as Jaeger and Zipkin enable the collection and visualization of tracing data across microservice architectures. These platforms allow engineers to analyze request flows and identify performance bottlenecks within distributed applications.

In recent years, frameworks such as OpenTelemetry have emerged as standardized solutions for telemetry instrumentation. OpenTelemetry provides a unified framework for collecting logs, metrics, and traces from applications, simplifying the integration of telemetry data into observability platforms.

By combining these open-source technologies, organizations can build flexible observability infrastructures capable of monitoring complex distributed systems. The integration of logs, metrics, and traces within unified platforms enables engineers to investigate system behavior more effectively and maintain reliable application environments.

The selection of open-source tools in this project is driven by flexibility, cost efficiency, and ease of integration. Compared to proprietary solutions, open-source technologies allow greater control over data management and system configuration, which is particularly important in edge environments where customization and resource optimization are critical.

6 System Architecture

The monitoring and observability platform developed in this project is based on a modular and containerized architecture designed to support the collection, processing, and visualization of telemetry data generated by distributed systems. Modern software infrastructures, particularly those involving cloud-native services and Internet of Things (IoT) environments, often consist of multiple interconnected components deployed across different computational nodes. As a consequence, the monitoring infrastructure must be capable of collecting data from heterogeneous sources while maintaining scalability, reliability, and ease of deployment.

The architecture implemented in this project was designed to support these requirements by combining containerization technologies with a set of open-source observability tools, as shown in figure 5. The monitoring stack integrates components responsible for telemetry collection, data aggregation, visualization, and system diagnostics. These components operate together to provide a comprehensive view of system behavior, allowing operators to monitor system health and investigate potential issues.

The system architecture follows a distributed model in which telemetry data is generated by services running on edge nodes and transmitted to a centralized monitoring environment. The monitoring infrastructure includes several specialized components responsible for collecting logs, metrics, and other telemetry signals. These components communicate through well-defined interfaces and are deployed within containerized environments to simplify management and ensure consistent execution across different systems.

The main components of the monitoring architecture include telemetry collectors, metrics monitoring systems, log aggregation tools, and visualization platforms. Together, these components form an integrated observability pipeline capable of transforming raw telemetry data into meaningful insights about system performance and operational status.

The following sections describe the main architectural elements of the monitoring platform, including the container-based infrastructure, the separation between backend and frontend services, the container orchestration strategy, and the distributed monitoring architecture connecting edge nodes to the central monitoring system.

In traditional cloud-based architectures, data generated by devices is typically transmitted directly to centralized cloud infrastructures for processing and storage. While this approach provides scalable computational resources, it can introduce significant latency and bandwidth consumption, particularly in scenarios involving large numbers of devices or real-time data processing requirements.

Edge computing addresses these challenges by moving part of the data processing closer to the devices that generate the data. Instead of sending all information to centralized cloud infrastructures, edge systems perform initial data processing locally, reducing the amount of data that must be transmitted over the network.

This approach provides several advantages in IoT environments. Processing data at the edge allows systems to react more quickly to events and reduces dependency on remote cloud infrastructures. In addition, local data processing helps improve system resilience in situations where network connectivity to the cloud may be intermittent or unreliable.

The adoption of edge computing has become increasingly important in modern IoT systems, where large numbers of devices continuously generate telemetry data that must be processed efficiently.

Edge computing provides several advantages in distributed IoT environments. One of the most significant benefits is reduced latency, since data processing occurs closer to the source rather than being transmitted to remote cloud infrastructures. This approach also reduces bandwidth usage because only processed or aggregated data needs to be sent to the cloud. Another important advantage is improved reliability, as edge nodes can continue operating even during temporary network disruptions. Edge computing also enhances system scalability by allowing the infrastructure to manage an increasing number of devices without overloading centralized resources. Finally, processing data locally may improve privacy and security, since sensitive information does not need to be transmitted to remote servers.

These advantages make edge computing particularly suitable for monitoring and observability systems in distributed IoT infrastructures.

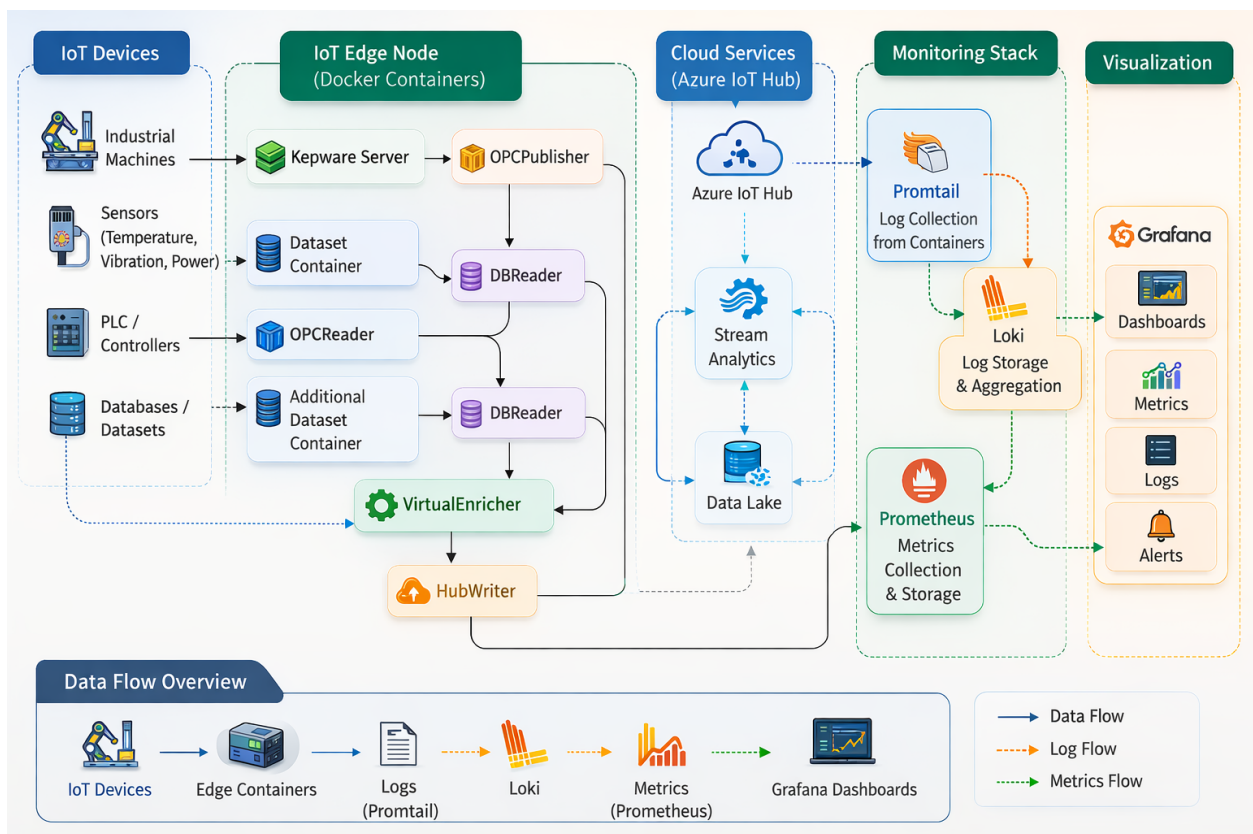


Figure 5: General architecture of the system.

6.1 Containers

Containers represent a fundamental building block of modern software infrastructures, providing lightweight virtualization and isolated execution environments. Unlike traditional virtual machines, containers package applications together with their dependencies, ensuring portability and consistent deployment across different environments.

In this project, containerization enables the modular deployment of monitoring components such as telemetry collectors, log aggregation systems, and visualization tools. Each service runs independently within its own container, improving scalability, fault isolation, and maintainability.

Additionally, containers simplify system deployment and reproducibility, allowing the monitoring infrastructure to be easily replicated across multiple edge nodes and environments.

6.2 Backend and Frontend

The monitoring platform is structured according to a layered architecture that separates backend services responsible for data processing from frontend components dedicated to data visualization and user interaction. This separation improves system organization and allows each component to focus on a specific aspect of the monitoring process.

Backend components are responsible for collecting, processing, and storing telemetry data generated by the monitored systems. These services operate in the background and continuously gather information about system activity, resource usage, and operational events. Backend services include telemetry collectors, metrics storage systems, and log aggregation tools.

In this project, backend components include Prometheus for metrics collection and storage, and Loki for log aggregation. These systems operate as independent services within the containerized infrastructure and are responsible for processing telemetry data collected from edge nodes.

One of the core backend components used in the monitoring architecture is the metrics monitoring system. Metrics generated by the monitored infrastructure are collected and stored in a time-series database that allows efficient querying

and long-term storage of telemetry data. This system enables operators to analyze trends in system performance and detect anomalies over time.

Another important backend component is the log aggregation system. Logs generated by applications and infrastructure services are collected and centralized in a dedicated log storage platform. This allows system administrators to investigate errors, analyze operational events, and perform detailed diagnostics when issues occur within the monitored environment.

Promtail plays a crucial role within the log collection pipeline. Promtail acts as a log shipping agent that runs alongside monitored services and collects log files generated by containers or applications. These logs are then forwarded to a centralized log aggregation system, which stores and indexes the log data for further analysis.

In the monitoring architecture implemented in this project, the centralized log aggregation service is responsible for collecting logs from multiple sources and making them available for querying and visualization. This allows operators to analyze system behavior and correlate log events with other telemetry signals such as metrics.

Frontend components, on the other hand, provide the user interface through which monitoring data can be explored and analyzed. Visualization platforms allow users to interact with the telemetry data through dashboards, charts, and graphical panels that represent system behavior over time.

One of the main frontend tools used in the monitoring platform is the dashboard visualization system. This component enables the creation of interactive dashboards where metrics and logs can be displayed in graphical form. By presenting telemetry data visually, operators can quickly identify anomalies and evaluate system performance without manually inspecting raw telemetry data.

The separation between backend and frontend components ensures that telemetry collection and processing can occur independently from data visualization, improving system scalability and simplifying maintenance. [6].

6.3 Docker Architecture

Docker was used as the primary platform for managing and deploying the containerized services within the monitoring infrastructure. Docker provides a standardized environment for building, distributing, and executing containers, making it an ideal solution for managing complex multi-service systems such as monitoring platforms.

Docker was selected due to its simplicity, portability, and strong support for multi-service environments. Compared to alternative orchestration solutions, it provides a lightweight and flexible approach suitable for edge deployments, where resource constraints and ease of configuration are critical factors.

Within the monitoring architecture, Docker is responsible for orchestrating the execution of the various observability services required by the platform. Each monitoring component is packaged into a dedicated container image and deployed as an independent service. This approach simplifies the configuration and management of the monitoring stack.

The Docker-based infrastructure allows multiple monitoring services to run simultaneously while maintaining isolation between components. Each service operates within its own container and communicates with other services through Docker networks. These networks enable containers to exchange data securely while remaining isolated from external processes.

The deployment of the monitoring stack is typically managed through container configuration files that define the services required by the system, their dependencies, and their network configurations. These configuration files describe how each container should be executed and how the services should interact with each other.

For example, the monitoring architecture includes containers responsible for metrics collection, log aggregation, data visualization, and system diagnostics. Each of these services is configured to communicate with the others through internal network endpoints, allowing telemetry data to flow across the monitoring pipeline.

Another useful capability provided by Docker is the ability to easily start, stop, or update monitoring services without affecting the rest of the system. If a monitoring component needs to be updated or reconfigured, the corresponding

container can be replaced without disrupting the operation of the remaining services.

Docker also simplifies the replication of the monitoring environment across different machines. Since the entire monitoring stack is defined through container images and configuration files, the system can be deployed on new machines with minimal setup. This capability is particularly useful when deploying monitoring infrastructures across distributed environments.

Overall, Docker provides the foundation for a flexible and scalable monitoring architecture capable of supporting multiple observability tools while maintaining consistent deployment procedures.

6.4 Edge Monitoring Architecture

The monitoring architecture implemented in this project follows a distributed model that separates edge nodes from the central monitoring system. This design is particularly suitable for environments where services or devices are deployed across multiple locations and must be monitored remotely.

Edge nodes represent the systems where the monitored applications and services are executed. These nodes generate telemetry data describing their operational state, including metrics, logs, and other observability signals. In distributed environments, each edge node may produce large volumes of telemetry data that must be collected and analyzed in order to maintain system reliability.

Within the monitoring architecture, telemetry data generated by edge nodes is collected locally and then transmitted to the central monitoring infrastructure. This approach allows the monitoring system to maintain a global view of the distributed infrastructure while reducing the complexity of managing telemetry data across multiple locations.

Log data generated by edge nodes is collected through local agents responsible for reading application log files and forwarding them to the central log aggregation system. Promtail acts as the log collection agent within this architecture. Running alongside the monitored services, Promtail continuously monitors log files and sends log entries to the centralized log storage system.

Metrics generated by the monitored infrastructure are collected through ded-

icated monitoring systems capable of periodically retrieving telemetry data from configured endpoints. These systems gather resource utilization metrics, application performance indicators, and other operational measurements from the edge nodes.

The centralized monitoring infrastructure aggregates telemetry data received from multiple edge nodes and stores it within specialized storage systems designed for time-series data and log management. By aggregating telemetry data at a central location, the monitoring platform can provide a unified view of system behavior across the distributed environment.

Visualization tools integrated within the monitoring platform allow operators to analyze telemetry data collected from the edge nodes. Through interactive dashboards and monitoring panels, system administrators can observe system activity, identify anomalies, and diagnose potential problems affecting the infrastructure.

This edge-to-central monitoring architecture ensures that telemetry data can be collected efficiently from distributed systems while maintaining a centralized platform for analysis and visualization. The approach supports scalability and allows the monitoring infrastructure to expand as additional edge nodes are added to the system.

The edge node represents the central component of the IoT edge infrastructure implemented in this project. The node is responsible for collecting data from connected devices, processing telemetry signals, and managing communication with monitoring and observability systems.

In order to support these functionalities, the edge node integrates several software components that operate together within a containerized environment. Containerization provides a flexible and scalable approach for deploying multiple services on the same physical device while maintaining isolation between different components.

The architecture of the edge node is designed to support multiple tasks simultaneously, including device communication, telemetry collection, monitoring operations, and log management.

The edge node performs several important functions within the monitoring architecture. It is responsible for collecting telemetry data generated by IoT devices connected to the system and for processing monitoring signals such as

logs and metrics produced by system services.

The edge node also manages communication between the local infrastructure and remote monitoring platforms. In addition, it supports containerized services responsible for data collection and processing tasks. Through these activities, the edge node contributes to ensuring reliable system operation by continuously monitoring the status of running services.

By integrating these capabilities within a single node, the system can efficiently manage telemetry data while maintaining flexibility in the deployment of monitoring tools and services.

This structured data pipeline allows the system to process telemetry signals efficiently and provides operators with real-time visibility into system operations.

The edge architecture implemented in this project is tightly integrated with the monitoring infrastructure described in later chapters. Monitoring tools operate directly within the edge node environment, collecting telemetry data generated by containerized services and system components.

By deploying monitoring services within the edge infrastructure, the system is able to observe its own operational state and detect anomalies in real time. Monitoring agents continuously collect metrics and logs that provide insights into system performance and service activity.

The integration between the edge architecture and the monitoring stack allows the system to support advanced observability capabilities. Through the combined use of log aggregation systems, metrics monitoring platforms, and visualization dashboards, operators can monitor the entire infrastructure from a centralized interface.

This integration enables the monitoring platform to provide valuable insights into system behavior and ensures that potential issues can be identified and addressed before they impact system reliability.

In order to design an effective monitoring solution, it was first necessary to analyze the internal architecture of the IoT Edge environment on which the monitoring platform would operate.

One of the initial steps in the development of the monitoring system consisted of analyzing the architecture of the IoT Edge device interacting with the proposed

observability platform. Before implementing the monitoring infrastructure, a preliminary study of the existing edge environment was required in order to understand the structure of the deployed containers, the internal data flow, and the interaction between the different components responsible for data processing and transmission.

The analysis was conducted using a dedicated architecture diagram provided in the project documentation. This diagram illustrated the internal composition of the Docker-based edge environment as well as the logical flow followed by the data collected from the industrial devices connected to the system. A schema of the IoT edge can be seen in the figure 7.

At the beginning of the data pipeline, telemetry data is generated by industrial devices and collected through the Kepware server, which acts as an interface between physical equipment and the digital infrastructure. The collected data is then processed through a series of containerized components, including OPCPublisher and OPCReader, which are responsible for extracting, transforming, and forwarding telemetry information within the edge environment.

In parallel, additional data sources are introduced through a dataset container, processed by components such as DBReader. These parallel data streams are combined and enriched by the VirtualEnricher component, which enhances the data with contextual metadata.

Finally, the processed data is transmitted to the cloud infrastructure through the HubWriter component, which acts as a bridge between the edge environment and the Azure IoT Hub service. The HubWriter therefore represented the final step of the edge processing pipeline, acting as the bridge between the edge device and the cloud platform.

Once the data reached the Azure IoT Hub, the architecture allowed the information flow to diverge into two different processing paths. One path directed the incoming data to a Data Lake, which served as a long-term storage solution for historical data and offline analysis. The other path directed the data to a Stream Analytics service, which enabled real-time processing and analysis of the incoming telemetry streams.

This dual-path architecture allowed the system to support both batch-oriented analytical workloads and real-time monitoring operations. Historical datasets stored in the Data Lake could be used for deeper analysis, machine learning training, or long-term performance evaluation, while the Stream Analytics service

enabled immediate detection of anomalies and operational events occurring in the monitored infrastructure.

Understanding the structure of this IoT Edge architecture was essential for the design of the observability platform described in this thesis. By analyzing the containerized components and the data flow across the edge device, it was possible to identify the most appropriate monitoring points and telemetry sources required for building an effective observability solution.

In order to provide effective monitoring capabilities for the IoT Edge infrastructure described in the previous sections, a dedicated observability stack was deployed as part of the system architecture. This monitoring stack was designed to collect, store, analyze, and visualize telemetry data generated by the containerized services running within the edge environment.

The monitoring solution adopted in this project was based on a set of open-source technologies that are widely used in modern observability platforms. In particular, the architecture combined Prometheus, Grafana Loki, Grafana, and Dozzle, each of which performed a specific role within the monitoring pipeline.

Prometheus acted as the primary component responsible for metrics collection and storage. As a time-series database specifically designed for monitoring applications, Prometheus periodically scraped metrics exposed by the services running in the containerized infrastructure. Prometheus was selected due to its pull-based model, which simplifies service discovery and integration in dynamic environments such as containers. Additionally, its native support for time-series data and powerful query language make it particularly suitable for real-time monitoring and analysis. These metrics included system-level indicators such as CPU usage, memory consumption, and container resource utilization, as well as application-level metrics related to the behavior of the monitored services.

The collected metrics were stored in Prometheus using a time-series data model, which allowed efficient retrieval and analysis of historical system performance information. This approach enabled operators to analyze the evolution of system behavior over time and identify potential performance degradation or abnormal operational patterns.

Alongside metrics collection, the monitoring architecture also required an effective mechanism for aggregating and analyzing log data generated by the various containers deployed within the edge environment. For this purpose, the system incorporated Grafana Loki, a log aggregation platform specifically designed to

integrate seamlessly with modern observability stacks.

Loki operates by organizing log data into streams identified through metadata labels rather than indexing the full content of log messages. This design significantly reduces storage requirements and allows efficient querying of log data even when dealing with large volumes of telemetry information. The adoption of Loki instead of traditional full-text indexing systems is motivated by its label-based architecture, which reduces storage overhead and improves query performance in containerized environments. This design aligns well with the needs of scalable and resource-efficient monitoring systems. Logs produced by the containerized services were collected and forwarded to Loki through the Promtail log shipping agent, which continuously monitored container log files and transmitted them to the Loki server.

Once stored within Loki, log data could be queried and analyzed using the LogQL query language, which provides powerful filtering and aggregation capabilities. This functionality enabled system operators to search for specific log events, identify error patterns, and correlate operational messages with metrics collected through Prometheus.

The visualization layer of the monitoring stack was implemented using Grafana, which served as the central interface for analyzing the telemetry data collected by both Prometheus and Loki. Grafana dashboards provided interactive visualizations of time-series metrics, log streams, and operational indicators, allowing operators to monitor system behavior in real time.

Through its integration with Prometheus and Loki, Grafana enabled the creation of comprehensive dashboards that combined multiple telemetry signals within a single interface. Operators could simultaneously examine system metrics, log events, and temporal correlations between different components of the monitored infrastructure.

In addition to these components, the monitoring architecture also incorporated Dozzle, a lightweight container log viewer designed specifically for Docker environments. Dozzle provided a simplified interface for quickly inspecting real-time container logs without requiring complex query operations. This tool proved particularly useful during development and debugging phases, when rapid access to container logs was required in order to diagnose issues occurring within the system.

The combination of Prometheus, Loki, Grafana, and Dozzle resulted in a compre-

hensive observability platform capable of providing both high-level monitoring and detailed diagnostic capabilities. Metrics collected through Prometheus offered quantitative insights into system performance, while logs aggregated by Loki provided detailed descriptions of events occurring within the containerized services. Grafana dashboards allowed operators to visualize and correlate this information, while Dozzle offered a direct view of real-time container logs when immediate inspection was required.

This integrated monitoring stack played a fundamental role in supporting the analysis and evaluation activities described in the later chapters of this thesis. By providing continuous visibility into the behavior of the IoT Edge infrastructure, the observability platform enabled the identification of operational issues, the analysis of system performance, and the validation of the monitoring approaches implemented in the project.

From an architectural perspective, the monitoring system developed in this project is organized into two main layers: the edge monitoring layer and the master monitoring layer.

The edge monitoring layer consists of multiple production edge devices where containerized applications are executed. Each edge node runs several monitoring components responsible for collecting telemetry data generated locally by the system. These services are responsible for collecting logs, exporting system-level metrics, and exposing container-level metrics. In this way, each edge node acts as a local telemetry source that continuously produces operational data describing the behavior of the system.

The master monitoring layer represents the central point of aggregation for all telemetry data produced by the edge nodes. The purpose of this central node is to collect monitoring data from multiple edges, store the collected information, process telemetry data, and transform it into meaningful metrics that can be visualized and analyzed. The master node therefore performs the role of centralized observability platform, allowing engineers to inspect system behavior across the entire infrastructure.

This architectural separation between edge nodes and the central monitoring node allows the monitoring system to scale more easily as the number of edge devices increases. Each edge node remains responsible for generating telemetry data locally, while the master node focuses on aggregation, analysis, and visualization of the collected information.

The monitoring architecture implemented in this project relies on a structured data flow designed to collect, process, and visualize telemetry data generated by the edge devices. The system processes two primary categories of telemetry information: system metrics and container logs. Each type of data follows a dedicated processing pipeline, allowing the monitoring platform to efficiently handle large volumes of operational data produced by the edge environment.

Metrics data is collected through specialized exporters running on the monitored nodes. In particular, system-level metrics are exposed by monitoring agents such as Node Exporter, while container-related metrics are provided by container monitoring tools such as cAdvisor. These exporters continuously expose operational information regarding resource usage, including CPU consumption, memory utilization, disk activity, and network traffic.

The metrics exposed by these exporters are periodically scraped by Prometheus, which acts as the central time-series database of the monitoring system. Prometheus stores the collected metrics and makes them available for further analysis and visualization. Through its query language, PromQL, engineers can perform complex queries on the stored data in order to identify patterns, detect anomalies, or analyze system performance over time.

In parallel with the metrics pipeline, the system also processes log data generated by the Docker containers running on the edge nodes. Container logs represent a crucial source of information for understanding application behavior and identifying potential failures. These logs are collected locally by Promtail, a lightweight log shipping agent designed to integrate with the Loki logging backend.

Promtail continuously monitors the log files produced by Docker containers and forwards the collected entries to the Loki log aggregation system. Unlike traditional log management systems, Loki indexes logs based on metadata labels rather than the full text of the log entries. This approach significantly reduces storage requirements while still allowing efficient querying of log streams.

Once collected and stored within Loki, log data becomes available for analysis through Grafana. Grafana acts as the central visualization platform of the monitoring architecture and enables engineers to query both metrics and logs from a unified interface. Through interactive dashboards and ad-hoc queries, users can correlate different telemetry signals in order to investigate anomalies and better understand the internal behavior of the monitored system.

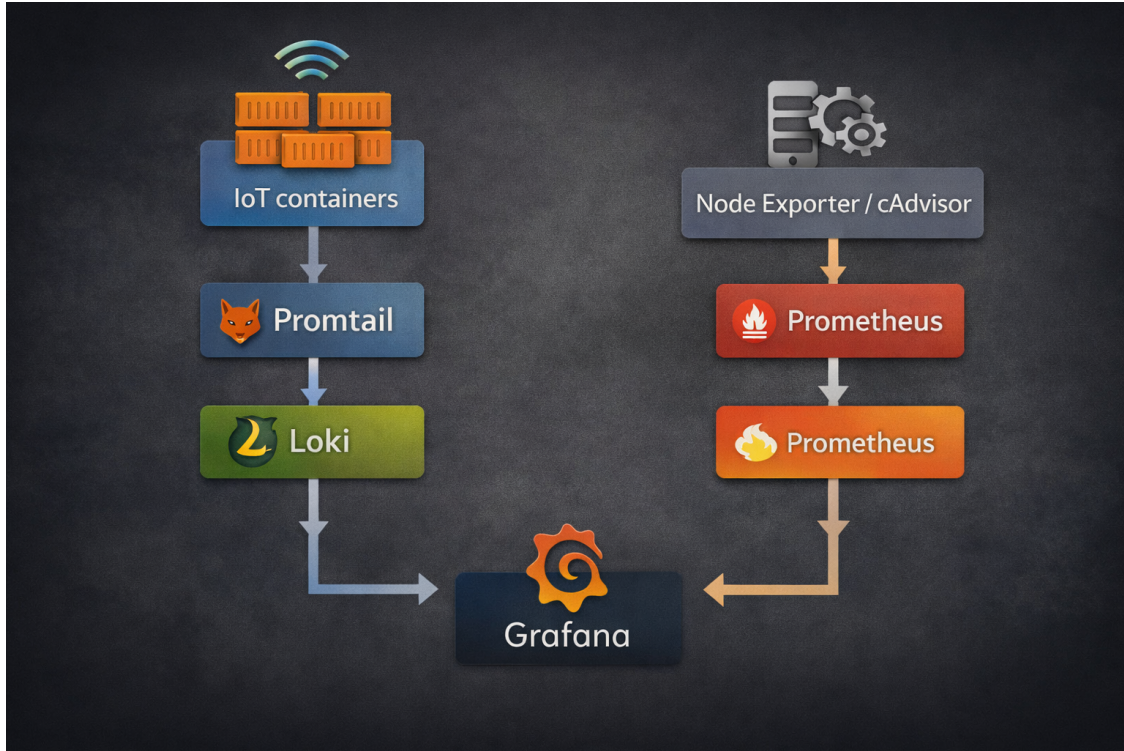


Figure 6: Flow data telemetry for the platform and services.

By separating metrics and log processing pipelines while maintaining a unified visualization layer, the monitoring architecture provides both scalability and flexibility. This design allows the system to monitor multiple edge nodes simultaneously while ensuring that engineers can efficiently explore telemetry data when diagnosing issues or analyzing system performance. Figure 6 illustrates a schema of how all the platforms and service are connected.

6.5 Design Considerations and Trade-offs

The design of the monitoring architecture involved several trade-offs related to performance, scalability, and system complexity. The adoption of a containerized architecture simplifies deployment and improves modularity, but introduces additional overhead in terms of resource usage and orchestration.

The choice of a centralized monitoring approach enables a global view of the system, but may introduce latency and dependency on network connectivity between edge nodes and the central monitoring system.

Another important design trade-off concerns the distribution of data processing

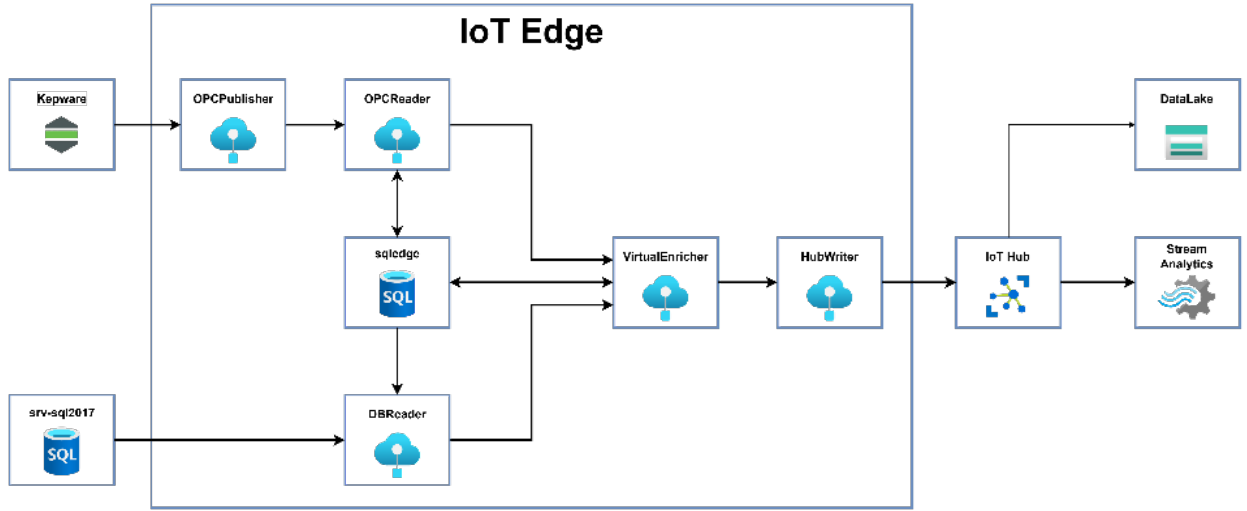


Figure 7: IoT edge structure.

between edge and cloud environments. Processing telemetry data at the edge reduces latency and bandwidth usage, enabling faster local reactions to system events. However, this approach limits the computational resources available for advanced analytics and introduces additional complexity in managing distributed monitoring components.

Similarly, the use of Prometheus and Loki provides efficient handling of metrics and logs, but requires careful configuration to manage data retention and resource consumption.

While Loki improves storage efficiency through label-based indexing, it limits full-text search capabilities compared to traditional log management systems. This trade-off was considered acceptable in order to achieve better scalability and performance in containerized environments.

Finally, the decision to prioritize logs and metrics over distributed tracing reduces system complexity and overhead, but limits the ability to analyze request-level behavior across services. This trade-off was considered acceptable within the scope of this project, with tracing identified as a potential future enhancement.

7 System Implementation

This chapter describes the practical implementation of the monitoring architecture introduced in Chapter 6. The focus is on the integration of Azure IoT

Edge, containerized services, and the observability stack within a functional distributed environment.

In the implemented system, Azure IoT Hub is used to manage communication between cloud services and edge devices, enabling secure deployment and remote configuration of containerized modules. Each edge node runs the Azure IoT Edge runtime, which allows monitoring components and data processing services to be executed locally.

The implementation emphasizes modular deployment, real-time telemetry collection, and centralized monitoring through a master–edge architecture. The structure of Azure components can be seen in the figure [11](#).

The cloud layer of the system is managed through Azure IoT Hub, which acts as the central coordination and communication platform for the distributed IoT devices. Through IoT Hub, edge devices can be registered, authenticated, and remotely managed. The cloud platform is responsible for maintaining connectivity with the edge nodes and enabling bidirectional communication between devices and cloud services.

On the edge side, the system relies on the runtime environment provided by Azure IoT Edge, which allows applications to be deployed as containerized workloads. These workloads are packaged as Docker containers and executed locally on the edge devices. The use of Docker provides a flexible and portable environment where monitoring services, data processing modules, and telemetry collectors can be deployed independently.

Containerization plays an important role in the architecture because it allows different system components to run in isolated environments while sharing the same host resources. Each monitoring service can therefore be deployed as a dedicated container, simplifying configuration, updates, and scalability. This approach also enables the system to easily integrate open-source monitoring technologies with the IoT edge infrastructure.

The integration between Azure services, containerized workloads, and open-source monitoring tools enables the system to maintain a high level of observability across both cloud and edge layers. Azure IoT Hub provides the cloud-based device management and communication infrastructure, Docker ensures the flexible deployment of containerized modules on the edge nodes, and the monitoring stack allows telemetry data and system performance indicators to be continuously collected and analyzed.

This architecture therefore creates a unified monitoring ecosystem in which telemetry data generated at the edge can be processed, stored, and visualized through open-source tools, while still maintaining connectivity with the cloud control layer provided by Azure. We can see an example of the architecture in the figure [12](#).

In the implemented monitoring architecture, the system follows a master–edge model, where a central monitoring node collects and aggregates telemetry data coming from multiple distributed edge nodes. This approach allows the monitoring infrastructure to remain centralized while the computational workloads and IoT services remain distributed across the edge devices.

The master node hosts the main monitoring services and acts as the central aggregation point of the system. In the implemented setup, this node runs several components of the observability stack, including Prometheus, Loki, Grafana, and Dozzle. These services are responsible for collecting, storing, and visualizing the telemetry information generated by the distributed infrastructure. The master node therefore acts as the control and visualization layer of the monitoring architecture.

Each edge node represents a device or computing unit where IoT workloads are executed, typically through containerized applications. These nodes run monitoring agents that expose metrics and logs related to system performance and container execution. In particular, exporters such as Node Exporter and cAdvisor are deployed on each edge node in order to expose system-level and container-level metrics. Node Exporter provides metrics related to CPU usage, memory consumption, disk I/O, and network activity of the host machine, while cAdvisor collects detailed information about the resource utilization of running containers.

Prometheus, running on the master node, periodically scrapes the metrics exposed by the exporters running on each edge device. This pull-based architecture allows the monitoring system to scale efficiently as new edge nodes are added to the infrastructure. Each node exposes its metrics through HTTP endpoints that Prometheus queries at regular intervals, storing the collected data in its time-series database.

At the same time, logs generated by containers and applications running on the edge nodes are collected and forwarded to the centralized logging system. Log agents such as Promtail run on the edge devices and monitor container logs or

system log files. These logs are then transmitted to Loki, which stores them and makes them available for querying and correlation with system metrics.

Grafana, also running on the master node, provides the visualization layer of the monitoring system. Through dashboards, it allows operators to inspect metrics, analyze logs, and monitor the health and performance of each edge node in real time. This centralized interface makes it possible to detect anomalies, investigate system behavior, and correlate different telemetry sources.

This master-edge architecture allows the monitoring infrastructure to scale efficiently as the number of edge nodes increases. Instead of deploying full monitoring stacks on each device, lightweight exporters and log collectors are installed on the edge nodes, while the heavy data storage and visualization components remain centralized on the master node. This design reduces resource consumption on constrained edge devices while still enabling comprehensive observability of the entire distributed IoT infrastructure. In the figure 8, it is represented the architecture master-edge.

In order to monitor the operational status of the edge infrastructure, the system relies on dedicated exporters that expose system and container metrics to the monitoring platform. Two fundamental components used in this context are Node Exporter and cAdvisor, which play a key role in enabling the collection of detailed telemetry data from each monitored edge node.

Node Exporter is a lightweight monitoring agent designed to expose hardware and operating system metrics in a format compatible with Prometheus. It runs directly on each monitored machine and collects low-level information related to the host system. The metrics provided by Node Exporter include CPU utilization, memory usage, disk activity, filesystem statistics, and network traffic. These metrics allow the monitoring platform to observe the overall health and performance of the machine hosting the IoT workloads.

Within the monitoring architecture implemented in this project, Node Exporter runs on each edge node and exposes a dedicated HTTP endpoint that can be periodically queried by Prometheus. Through its pull-based monitoring model, Prometheus retrieves the metrics from the exporters at regular intervals and stores them as time-series data. This mechanism enables continuous observation of system behavior and allows the detection of anomalies or performance degradations affecting the infrastructure.

While Node Exporter focuses on host-level monitoring, containerized environ-

ments require additional visibility into the behavior of individual containers. For this purpose, the monitoring stack integrates cAdvisor, a tool specifically designed to collect, aggregate, and expose resource usage information related to running containers.

cAdvisor continuously analyzes the container runtime environment and extracts metrics regarding CPU consumption, memory allocation, network traffic, and filesystem usage for each running container. These metrics provide a detailed view of how containerized applications interact with the system resources and allow operators to identify potential bottlenecks or abnormal behaviors.

In the implemented infrastructure, cAdvisor runs alongside the container runtime on each edge device and exposes its metrics through an endpoint that is also scraped by Prometheus. By combining the metrics provided by Node Exporter and cAdvisor, the monitoring system is able to obtain both host-level and container-level observability, offering a comprehensive overview of the entire edge computing environment.

The collected metrics are then visualized through dashboards in Grafana, where operators can inspect system behavior in real time. These dashboards typically display key performance indicators such as total CPU usage, memory consumption, container resource utilization, network traffic, and the number of active containers. By correlating these metrics, it becomes possible to identify system load patterns, detect abnormal resource usage, and evaluate the stability of the deployed services.

Figure 10 shows an example of a monitoring dashboard where metrics collected from Node Exporter and cAdvisor are aggregated and visualized. The dashboard provides an overview of the current infrastructure state, including the number of running containers, total memory usage, CPU consumption trends, and network throughput. Such visualizations play an important role in the monitoring workflow, as they allow system administrators to quickly assess the health of the edge infrastructure and react promptly to potential issues.

After defining the architecture of the monitoring platform, the next step of the project consisted in implementing the observability infrastructure and integrating the various monitoring tools required to collect, process, and visualize telemetry data. The implementation phase focused on transforming the theoretical architecture described in the previous chapter into a functional monitoring system capable of operating within a distributed environment.

The implementation phase involved the deployment and configuration of the monitoring stack within a containerized environment. Docker was used to manage all services, while configuration files defined service interactions, network settings, and data flows.

Particular attention was given to configuring Prometheus scraping targets, setting up log collection through Promtail, and defining Grafana dashboards for real-time visualization of system metrics and logs.

The monitoring platform integrates several open-source technologies designed to support observability operations. These technologies include tools for metrics monitoring, log aggregation, data visualization, and system diagnostics. By combining these tools within a containerized infrastructure, the platform provides a comprehensive monitoring solution capable of collecting telemetry data from multiple sources and presenting this information through centralized dashboards.

The implementation also involved the configuration of telemetry pipelines responsible for transmitting data from monitored services to the central monitoring platform. These pipelines ensure that logs, metrics, and other telemetry signals generated by distributed services can be collected, processed, and stored in an efficient and scalable manner.

The following sections describe the main stages of the implementation process, including the definition of project objectives, the development of the first prototype using a lightweight web framework, the integration of monitoring tools within the platform, and the deployment of the system within a cloud-based infrastructure.

During the analysis phase, visualization tools play an important role in helping operators interpret monitoring data. Dashboards provide graphical representations of metrics and logs that make it easier to detect anomalies or performance issues.

Visualization platforms allow users to explore telemetry data interactively and correlate different types of observability signals. Through dashboards, system administrators can quickly identify abnormal trends or unexpected events. Figure 9 shows an example of a dashboard that visualizes telemetry data collected from an IoT Edge device.

Visualization platforms provide several features that support monitoring activities. These tools typically offer interactive dashboards capable of displaying

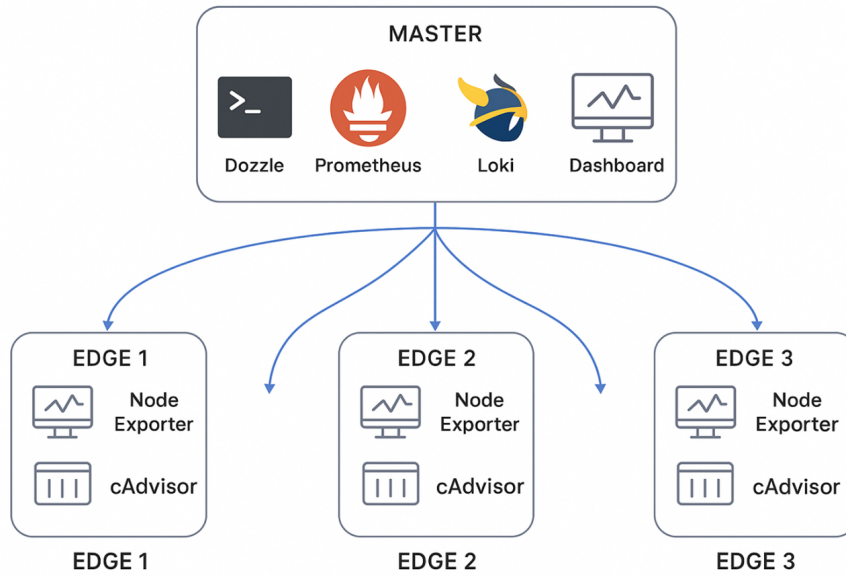


Figure 8: Schema of edges.

multiple metrics simultaneously and time-series charts that allow operators to observe how metrics evolve over time. They also include log exploration interfaces that enable users to search and filter log entries generated by monitored services.

Another important feature consists of alert visualization panels that highlight active incidents within the monitored system. In addition, visualization tools usually provide customizable panels that allow engineers to tailor dashboards according to their specific monitoring requirements.

These features make visualization tools essential components of modern monitoring infrastructures.

7.1 Project Goals

The primary objective of the project was the development of a monitoring and observability platform capable of collecting telemetry data from distributed services and presenting this information through a centralized monitoring interface. The system was designed to support the analysis of operational data generated by applications, containers, and infrastructure components.

One of the key goals of the project was to create a monitoring environment



Figure 9: Example dashboard edge master monitoring.

capable of supporting modern distributed systems, including container-based applications and IoT infrastructures. These environments typically consist of multiple services operating across different nodes, making it difficult to monitor system behavior using traditional monitoring approaches. As a result, the monitoring platform needed to support scalable telemetry collection and centralized data aggregation.

Another important objective was the integration of multiple observability signals within a unified monitoring platform. The system was designed to collect logs, metrics, and other telemetry data generated by the monitored infrastructure. By combining different observability signals, the monitoring platform allows operators to obtain a more complete view of system behavior and identify the root causes of potential issues.

The project also aimed to explore the integration of modern monitoring tools commonly used in cloud-native environments. Several open-source tools were evaluated and integrated within the monitoring infrastructure in order to support telemetry collection, log aggregation, and data visualization. These tools provide advanced capabilities for monitoring distributed systems and have become widely adopted in modern DevOps environments.

Another important design objective concerned system scalability. The moni-

toring platform was designed in such a way that new monitored services and nodes could be added without requiring significant changes to the monitoring infrastructure. By relying on containerized services and modular monitoring components, the system can be easily expanded to support larger infrastructures.

Finally, the project aimed to investigate the potential integration of machine learning techniques within the monitoring process. By analyzing historical telemetry data, machine learning models may be used to detect anomalies or predict potential system failures. Although the implementation of advanced machine learning models was not the primary focus of the monitoring infrastructure, the system was designed in such a way that machine learning components could be integrated in future developments.

The development of the project began with the analysis of an experimental Docker-based edge device composed of several containers. This machine was designed specifically for testing and experimentation activities and therefore represented a development environment in which potential errors or failures would not affect production systems.

The primary objective of this phase was to study and analyze the logs, records, and system information generated by the development machine in order to understand the internal behavior of the edge infrastructure. By connecting to the device through an SSH server, it was possible to access the edge system and execute several Bash commands in order to inspect the environment. The first step consisted of examining the list of containers running on the edge device. After identifying the active containers, each container was analyzed individually by inspecting the log information it generated.

This preliminary investigation allowed a deeper understanding of the operational behavior of the different services running within the edge environment. During this analysis phase, particular attention was given to understanding the structure and content of the log records produced by the containers. A typical log entry generally contains three main elements: a timestamp indicating when the event occurred, the type of message generated by the system, and the message content describing the event itself. The timestamp provides the temporal context of the event, while the message type allows logs to be categorized according to their severity or purpose.

Through this classification it was possible to identify several categories of log entries, including informational messages, debugging messages, warnings, and

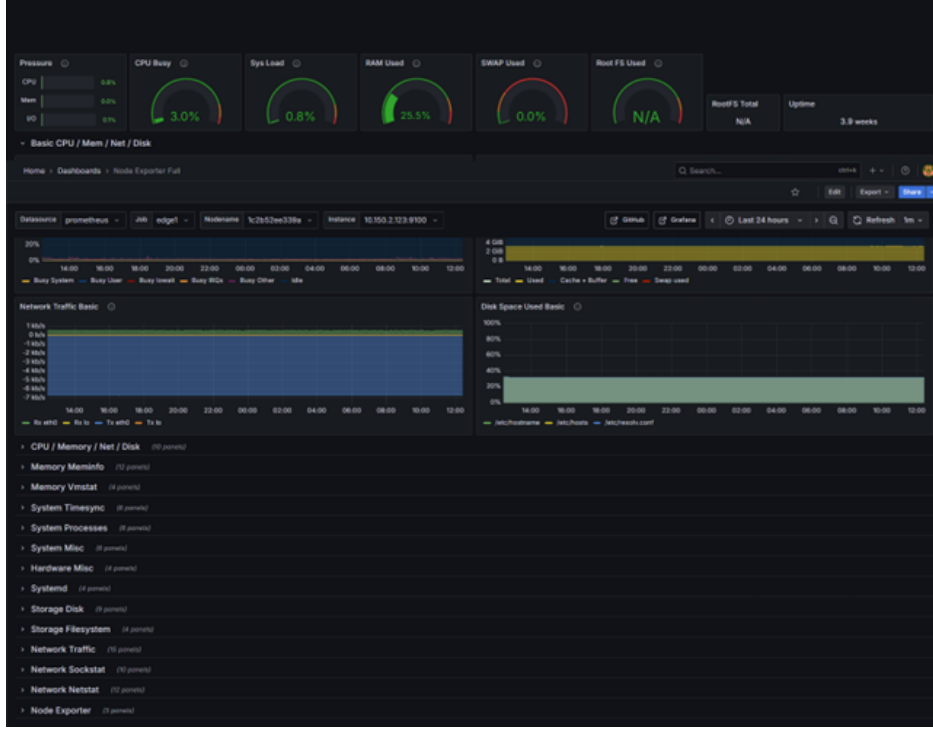


Figure 10: Example dashboard NodeExporter.

error messages. Since the primary objective of the project was to identify anomalous behavior within Docker containers running on the edge device, the analysis focused primarily on warning and error messages. These log entries often indicate abnormal system behavior or failures that may require investigation or corrective actions.

7.2 Flask Prototype

During the early stages of the project, several experimental approaches were evaluated in order to determine the most suitable architecture for implementing the monitoring and analysis platform. One of the first prototypes developed during this exploratory phase was based on the Flask framework, a lightweight Python web framework widely used for rapid development of web applications and APIs.

The initial idea behind the Flask-based prototype was to create a simple web service capable of receiving telemetry data from the monitoring infrastructure, processing the collected information, and displaying the results through a web interface. The prototype was designed to act as an intermediary layer between

the raw monitoring data and the analytical components of the system.

In this early implementation, the Flask application exposed a set of REST endpoints that could receive telemetry data produced by the monitoring stack. The incoming data was then processed by Python scripts that performed basic parsing and analysis operations before storing the results in temporary storage structures. Once processed, the information could be visualized through a simple web dashboard generated by the Flask application itself.

This approach allowed the rapid development of a functional proof-of-concept system. The simplicity of the Flask framework made it possible to quickly implement the required functionality and test the integration between different components of the monitoring infrastructure. Moreover, Python provided a flexible environment for experimenting with data analysis techniques and testing different approaches to data processing.

However, despite the advantages in terms of development speed and flexibility, the Flask-based prototype revealed several limitations when evaluated in the context of the overall project objectives. In particular, the architecture of the prototype was primarily designed for batch-oriented data processing, meaning that telemetry data had to be collected and processed in discrete intervals rather than being analyzed continuously as it was generated.

This limitation proved to be significant because one of the main goals of the project was to enable near real-time monitoring and analysis of system behavior. In modern observability systems, the ability to analyze telemetry streams in real time is essential for detecting anomalies, identifying performance issues, and reacting quickly to operational incidents.

The Flask-based architecture did not provide an efficient mechanism for handling continuous streams of telemetry data at scale. While it was technically possible to process data periodically, the framework was not designed to support the type of streaming data pipelines typically required by real-time monitoring systems. As a consequence, the prototype required the accumulation of data before analysis could take place, introducing delays between data generation and data interpretation.

Another limitation concerned scalability. The architecture of the prototype relied on a centralized web application responsible for receiving and processing telemetry information. As the volume of monitoring data increased, this design would have required significant modifications to support higher throughput and

distributed processing capabilities.

For these reasons, the Flask-based prototype was ultimately considered insufficient for meeting the real-time monitoring requirements of the project. Although the experiment provided valuable insights into the structure of the monitoring data and the types of analysis required, it became clear that a different architecture would be necessary to support a fully operational observability platform.

Consequently, the development effort shifted toward the adoption of a monitoring stack based on specialized observability tools capable of handling telemetry streams in real time. Technologies such as Prometheus, Grafana, and Loki provided native support for time-series data, log aggregation, and real-time visualization, making them more suitable for the implementation of a scalable observability platform.

The experience gained during the development of the Flask prototype nevertheless played an important role in shaping the final architecture of the system. By experimenting with an initial implementation, it was possible to better understand the characteristics of the telemetry data produced by the IoT Edge infrastructure and to identify the requirements that the final monitoring platform would need to satisfy.

The Flask prototype was primarily used to experiment with different approaches for collecting and presenting monitoring data. Through the prototype, it was possible to test data ingestion mechanisms and evaluate how telemetry data could be displayed through a web-based interface. This initial implementation allowed the development process to focus on the core functionalities of the monitoring platform before integrating more advanced monitoring tools.

The prototype implemented several basic features required for the monitoring system. These features included simple data visualization capabilities, the ability to receive telemetry data from monitored services, and mechanisms for displaying monitoring information through a web interface. Although the prototype was not intended to replace specialized monitoring platforms, it provided a useful environment for experimenting with monitoring concepts and verifying the feasibility of the proposed architecture.

Another important role of the Flask prototype was the experimentation with data pipelines used to transmit telemetry data between monitored services and the monitoring platform. By testing these pipelines in a controlled environment,

it was possible to better understand the requirements for telemetry collection and to evaluate different approaches for integrating monitoring tools.

As the project progressed, the limitations of the prototype became evident. While Flask provided a flexible platform for developing custom monitoring interfaces, implementing a complete monitoring infrastructure solely through a custom application would require significant development effort. For this reason, the project gradually transitioned toward the integration of established open-source monitoring tools capable of providing advanced observability capabilities.

Despite this transition, the Flask prototype remained an important step in the development process. It allowed the validation of the architectural concepts proposed for the monitoring platform and provided valuable insights into the requirements for telemetry collection and visualization.

This design choice reflects a trade-off between development flexibility and system scalability. While Flask allowed rapid prototyping, it lacked native support for real-time data streams and scalable telemetry processing, making specialized observability tools a more suitable solution for production environments.

7.3 Monitoring Platform Implementation

After the initial experimentation phase, the monitoring platform was implemented by integrating several open-source tools designed for observability and telemetry management. These tools were selected based on their compatibility with containerized environments and their ability to support scalable monitoring architectures.

The monitoring stack includes components responsible for collecting metrics, aggregating logs, visualizing telemetry data, and monitoring container activity. Each component performs a specific role within the observability pipeline and contributes to the overall monitoring capabilities of the system.

One of the key components of the monitoring platform is the metrics monitoring system. Metrics generated by applications and infrastructure services are collected and stored in a time-series database that allows efficient querying and analysis of telemetry data. This system continuously retrieves performance indicators such as CPU usage, memory utilization, and service activity metrics from the monitored infrastructure.

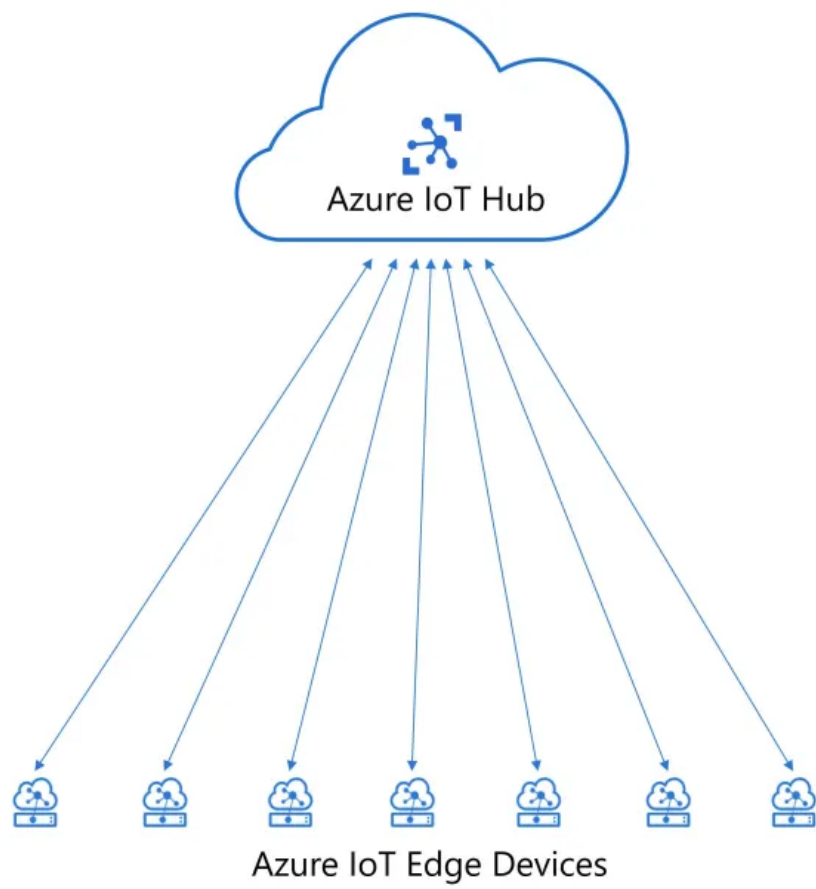


Figure 11: Azure edges structure.

Prometheus was adopted as the main monitoring system for metrics collection. Prometheus periodically retrieves metrics from configured targets through HTTP endpoints exposed by monitored services. This pull-based model allows the monitoring platform to collect metrics from multiple sources without requiring complex data pipelines.

Prometheus was selected due to its pull-based architecture, which simplifies the integration with dynamically deployed containers and edge nodes. This approach avoids the need for complex push configurations and improves scalability in distributed environments.

Log management represents another essential aspect of the monitoring infrastructure. Applications and services running within the monitored environment generate log files containing information about system events, warnings, and errors. These logs provide valuable insights into system behavior and are particularly useful for diagnosing operational problems.

To manage log aggregation, the monitoring platform integrates a centralized log storage system designed to collect and index log entries generated by distributed services. Log data is collected through Promtail, which acts as a log shipping agent responsible for monitoring log files and forwarding log entries to the centralized log storage service.

The adoption of Loki instead of traditional log management systems is motivated by its label-based indexing approach, which significantly reduces storage overhead. This design is particularly suitable for containerized environments where logs are generated at high frequency and need to be efficiently correlated with metrics.

Promtail continuously scans log files generated by containers and applications. When new log entries are detected, they are transmitted to the log aggregation system where they are stored and indexed for future analysis. This pipeline enables the monitoring platform to collect logs from multiple sources and centralize them in a single storage system.

The aggregated logs can then be queried and visualized through the monitoring dashboards, allowing system administrators to investigate errors and analyze system behavior over time.

Visualization plays a crucial role in the monitoring platform, as raw telemetry data must be transformed into meaningful insights that can be easily interpreted

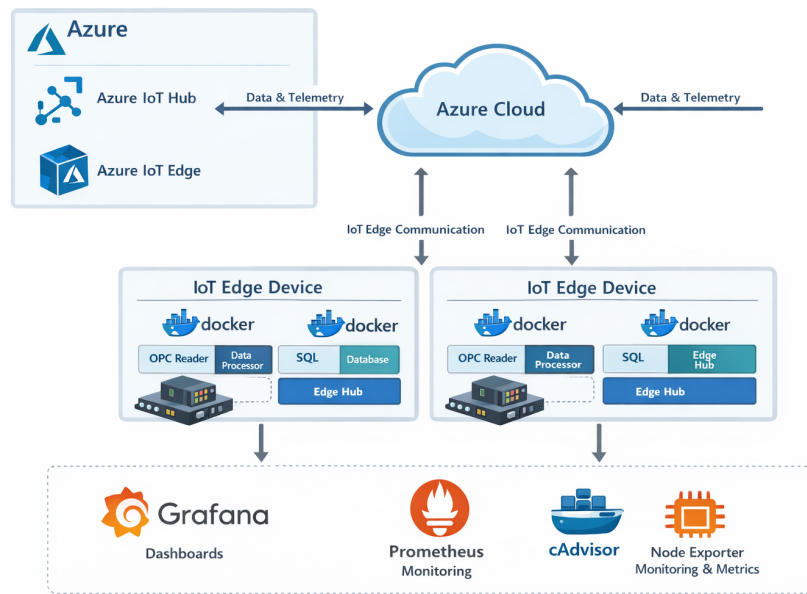


Figure 12: General structure Azure and docker.

by operators. For this purpose, the monitoring platform integrates a dashboard visualization system capable of displaying metrics and logs through interactive charts and graphical panels.

These dashboards allow users to observe trends in system performance, monitor resource utilization, and detect anomalies in real time. By providing an intuitive visualization interface, the monitoring platform enables operators to analyze large volumes of telemetry data efficiently.

Another useful tool integrated within the monitoring environment is a container log viewer designed to simplify the inspection of logs generated by running containers. This tool allows developers and system administrators to quickly access container logs and identify potential issues without manually inspecting container files.

All monitoring services were deployed within containers and executed as independent components within the monitoring infrastructure. This modular design ensures that each service can operate independently while contributing to the overall observability capabilities of the system.

7.4 Azure Infrastructure

Although the core monitoring system operates primarily at the edge and master levels, cloud integration was explored to evaluate potential extensions of the platform. Azure services were considered for advanced analytics and long-term storage of telemetry data.

The cloud environment was used to evaluate how monitoring data collected from the local infrastructure could be processed and analyzed using external computational resources. By transmitting telemetry data to a cloud-based environment, it becomes possible to perform more advanced data analysis and integrate machine learning techniques capable of identifying complex patterns in system behavior.

Azure services were considered as a potential platform for hosting machine learning models and performing large-scale data analysis on telemetry datasets. Cloud-based machine learning environments provide access to powerful computing resources and specialized tools designed for data processing and model training.

The integration of cloud infrastructure within the monitoring architecture also enables the possibility of hybrid monitoring solutions, where telemetry data is collected locally but analyzed using remote computational resources. This approach can be particularly useful in IoT environments where edge nodes may have limited computational capabilities.

Another advantage of cloud integration is the ability to store large volumes of historical telemetry data for long-term analysis. Historical datasets can be used to identify trends in system behavior, evaluate system performance over extended periods of time, and train predictive models capable of detecting anomalies in monitoring data.

Although the cloud-based components were not used as the primary monitoring infrastructure in this project, their integration demonstrates how modern observability platforms can combine local monitoring systems with cloud-based analytics services.

This hybrid approach provides a flexible framework for extending the monitoring capabilities of the platform and enables future research into intelligent monitoring systems based on machine learning and large-scale data analysis.

7.5 Implementation Considerations

The implementation of the monitoring platform involved several design trade-offs. The use of a centralized master node simplifies data aggregation and visualization, but introduces a dependency on network connectivity between edge devices and the monitoring system.

The adoption of containerized services improves modularity and deployment flexibility, but adds overhead in terms of resource usage, which must be carefully managed on constrained edge devices.

The decision to rely on Prometheus and Loki enables efficient handling of metrics and logs, but requires careful configuration of data retention policies to avoid excessive resource consumption.

Finally, the exclusion of distributed tracing simplifies the architecture but limits the ability to analyze fine-grained interactions between services. This aspect is considered a potential extension for future work.

8 Machine Learning Integration

Machine learning techniques were explored in this project as an extension of the monitoring platform, with the goal of improving anomaly detection and telemetry data analysis. Traditional monitoring approaches based on static thresholds were considered insufficient for capturing complex patterns in dynamic environments.

Machine learning integration was designed to complement the existing observability infrastructure without modifying its core architecture. Telemetry data collected through the monitoring stack was used as input for analytical models capable of identifying deviations from normal system behavior.

Two integration approaches were considered: a hybrid local approach, where lightweight analysis is performed at the edge, and a cloud-based approach leveraging Azure services for more computationally intensive tasks.

The integration of machine learning within the observability framework focused primarily on the analysis of metrics and telemetry streams collected by the

monitoring platform. These data sources provide valuable information about system performance, resource utilization, and operational activity, making them suitable inputs for machine learning models.

A hybrid approach was considered for the deployment of machine learning models. In this architecture, certain analysis tasks can be executed locally near the edge nodes, while more computationally intensive operations can be performed within cloud infrastructures. This hybrid model allows the system to balance computational efficiency with scalability.

In addition, the use of cloud-based machine learning platforms enables the training and deployment of models using large volumes of historical telemetry data. Cloud environments provide powerful computational resources and integrated machine learning services that simplify the development and management of predictive models.

The following sections describe the two main approaches explored in this project: a hybrid local machine learning approach and the integration of machine learning models within an Azure-based infrastructure.

8.1 Hybrid Local Approach

The hybrid local approach to machine learning focuses on performing certain analytical operations directly within the edge environment. In this model, machine learning models or lightweight analytical components are deployed close to the monitored systems, allowing telemetry data to be processed locally before being transmitted to the central monitoring platform.

This approach offers several advantages in distributed environments such as IoT infrastructures. One of the most significant benefits is the reduction of network traffic. Instead of transmitting all raw telemetry data to the cloud, local analysis can filter or preprocess the data, sending only relevant information or aggregated results to the central monitoring system.

Another advantage of local machine learning is the reduction of response latency. When anomaly detection algorithms are executed directly on edge nodes, abnormal behavior can be detected more quickly without requiring round-trip communication with remote cloud services. This capability is particularly important in systems where rapid reaction to failures or abnormal conditions is

required.

In the hybrid architecture considered in this project, the monitoring agents deployed on edge nodes collect telemetry data such as metrics related to system performance, device activity, and resource utilization. These data streams can serve as inputs to machine learning algorithms designed to detect unusual patterns in system behavior.

Lightweight anomaly detection techniques can be applied to time-series telemetry data in order to identify deviations from normal system behavior. These models learn baseline patterns from historical metrics and detect anomalies based on statistical variations.

In practice, the hybrid approach involves a combination of local preprocessing and centralized analysis. Edge nodes may perform initial filtering, sampling, or anomaly detection tasks, while the central monitoring platform receives aggregated telemetry data and performs more comprehensive analysis.

This architecture distributes computational tasks across the system, allowing edge nodes to perform lightweight processing while the central platform handles more resource-intensive operations such as model training and large-scale analysis.

The hybrid local approach therefore contributes to improving the efficiency and responsiveness of the monitoring infrastructure while maintaining the ability to perform deeper analysis at the central level.

In the context of this project, the hybrid approach was explored at a conceptual level. Due to resource constraints and the primary focus on monitoring infrastructure, full deployment of machine learning models on edge nodes was not implemented. However, the architecture was designed to support future integration of lightweight anomaly detection components directly at the edge level.

8.2 Azure ML Model

While local analysis can improve responsiveness and reduce network overhead, cloud-based machine learning platforms provide powerful capabilities for training and managing advanced models. For this reason, the project also explored the

integration of machine learning models within the Azure ecosystem.

The integration of machine learning within the Azure ecosystem was explored as a potential extension of the monitoring platform, rather than a fully implemented component. Azure services provide a suitable environment for training and deploying machine learning models using large-scale telemetry data.

Within the monitoring architecture developed for this project, telemetry data collected from edge nodes can be transmitted to the centralized monitoring platform hosted on Azure. Once stored and aggregated, this data becomes a valuable resource for training machine learning models that analyze system behavior over extended time periods.

One possible application of machine learning in this context is anomaly detection in time-series data. Metrics generated by monitoring systems often follow identifiable patterns related to normal system operation. Machine learning models can learn these patterns and detect deviations that may indicate failures, performance degradation, or abnormal activity.

Cloud-based machine learning services also simplify the management of the model lifecycle. In Azure environments, models can be trained using historical telemetry datasets, evaluated using validation procedures, and deployed as inference services that analyze incoming monitoring data in real time.

Another advantage of cloud-based machine learning is scalability, as computational resources can be dynamically allocated to support both training and inference tasks in large-scale monitoring environments.

In the architecture explored in this project, the Azure-based machine learning model complements the local monitoring infrastructure. Telemetry data collected by the observability platform can be processed by machine learning models that identify patterns, detect anomalies, and support predictive analysis of system behavior.

Local components provide rapid responses to operational events, while cloud-based models enable deeper analysis of historical telemetry data.

The integration of machine learning into the monitoring architecture therefore represents an important step toward more intelligent observability systems capable of adapting to complex and dynamic environments.

In this project, the Azure-based machine learning integration was evaluated conceptually and through preliminary analysis of telemetry data. Full deployment of machine learning pipelines within Azure was not implemented, but the system architecture supports future integration of predictive models and advanced analytics services.

8.3 Design Considerations and Trade-offs

The integration of machine learning within the monitoring architecture involves several trade-offs related to performance, complexity, and system design.

The hybrid local approach reduces latency and network usage by processing data closer to the source, but is limited by the computational constraints of edge devices. Deploying machine learning models at the edge requires lightweight algorithms and careful resource management.

On the other hand, cloud-based machine learning enables more advanced analysis and model training using large datasets, but introduces additional latency and dependency on network connectivity between edge nodes and cloud services.

Another trade-off concerns system complexity. Integrating machine learning components increases architectural complexity and requires additional data pipelines, model management, and monitoring of the models themselves.

For this reason, the project prioritizes a modular approach in which machine learning is treated as an optional extension rather than a core component of the monitoring system. This design allows future integration without impacting the stability of the existing infrastructure.

9 Monitoring Metrics and Tools

Monitoring systems rely on the continuous collection and analysis of telemetry data in order to evaluate the health and performance of software infrastructures. Among the different types of observability signals discussed in previous chapters, metrics represent one of the most widely used forms of telemetry data. Metrics provide numerical measurements describing system behavior over time and allow

system administrators to evaluate the operational state of services, infrastructure components, and applications.

In modern distributed environments, monitoring metrics play a fundamental role in maintaining system reliability. Large-scale systems generate vast amounts of telemetry data that must be collected, stored, and analyzed efficiently. Monitoring tools provide the necessary infrastructure to manage this information and transform raw telemetry signals into actionable insights that can support operational decision-making.

Within the monitoring platform implemented in this project, metrics collection and analysis are performed through a dedicated monitoring stack composed of several open-source tools. These tools are designed to operate together to support the entire monitoring pipeline, from telemetry collection to data visualization.

The monitoring infrastructure includes systems responsible for retrieving metrics from monitored services, storing telemetry data in time-series databases, and presenting monitoring information through visualization dashboards. This architecture allows operators to observe system performance in real time and analyze historical telemetry data to identify trends or abnormal behaviors.

In addition to metrics monitoring, the observability platform also integrates log management systems that allow operators to inspect log entries generated by applications and containers. Logs provide detailed information about system events and complement metrics monitoring by offering deeper insights into the internal behavior of software systems.

For this purpose, the monitoring platform integrates a centralized log aggregation pipeline in which log entries generated by containers and services are collected by specialized agents and forwarded to a centralized storage system. Tools such as Promtail are responsible for collecting log data from running containers, while centralized log aggregation systems store and index the logs to make them available for querying and visualization.

Visualization tools represent another essential component of the monitoring stack. Raw telemetry data stored within monitoring systems must be presented in a way that allows system administrators to quickly understand system behavior. Dashboard platforms provide interactive visual representations of monitoring metrics, allowing operators to observe trends in system performance and identify potential anomalies.

The following sections describe the types of metrics collected within the monitoring infrastructure and explain how the Prometheus monitoring system was integrated into the architecture to support telemetry collection and analysis.

The monitoring infrastructure implemented in this project relies on a set of specialized tools that work together to collect, store, analyze, and visualize telemetry data. These tools form a monitoring stack designed to support observability within the system.

The main components of the monitoring infrastructure include:

- Prometheus, responsible for collecting and storing time-series metrics generated by monitored services;
- Promtail, used to collect logs from containers and forward them to the log aggregation system;
- Loki, which stores and indexes log data in a centralized repository;
- Grafana, used for data visualization and dashboard creation.

Each of these components plays a specific role in the monitoring process. Together, they create an integrated observability platform capable of analyzing both logs and metrics generated by the system.

9.1 Base Metrics

In addition to metrics collected through Prometheus, the monitoring infrastructure implemented in this project also required an effective mechanism for collecting and analyzing log data generated by the different services running within the containerized environment. Logs represent one of the most important sources of information in observability systems because they provide detailed descriptions of events occurring within software components, including errors, warnings, and operational messages.

To support log collection and analysis, the monitoring stack incorporated Grafana Loki, a log aggregation system specifically designed for large-scale observability platforms. Loki is particularly well suited for containerized environments because it focuses on indexing metadata rather than the full content of log messages,

which significantly reduces storage requirements and improves query performance when analyzing large volumes of logs.

Within the architecture of the monitoring platform, Loki acted as the central component responsible for storing and managing log streams produced by the various containers running in the infrastructure. However, in order for logs to reach the Loki service, an additional component was required to collect log files from the containers and forward them to the aggregation system. For this purpose, the platform employed Promtail, which is a log shipping agent designed to work natively with Loki.

Promtail was deployed as part of the monitoring stack and configured to automatically discover log files generated by Docker containers running on the system. Once configured, the agent continuously monitored the specified log directories and detected new log entries as soon as they were written by the running services. These log entries were then enriched with metadata such as container identifiers, service names, and timestamps before being transmitted to the Loki server.

Loki organizes logs using labeled streams instead of indexing full log content. This reduces storage requirements and improves query performance when handling large volumes of log data.

The adoption of Loki instead of traditional full-text indexing systems is motivated by its label-based architecture, which significantly reduces storage overhead and improves query efficiency. This design is particularly suitable for containerized environments, where logs are generated at high frequency and need to be correlated efficiently with system metrics.

Once the logs are stored within Loki, they can be queried and analyzed through the Grafana interface. Grafana integrates natively with Loki and provides a powerful query language called LogQL, which allows users to search, filter, and analyze log streams in a flexible manner. Through LogQL queries, operators can extract relevant information from large volumes of log data, identify patterns, and detect abnormal system behavior.

Another important capability provided by Loki and Promtail is the possibility of correlating log data with metrics collected through Prometheus. Since both systems are integrated within the same visualization environment provided by Grafana, it becomes possible to analyze logs and metrics side by side. This correlation is extremely valuable when investigating system incidents, as it allows

operators to relate changes in system metrics with specific events recorded in the logs.

For example, an unexpected increase in error metrics detected through Prometheus can be analyzed further by examining the corresponding log streams stored in Loki. By identifying the log entries generated during the same time interval, it becomes easier to determine the root cause of the anomaly and identify the specific service or component responsible for the problem.

The integration of Loki and Promtail therefore complements the metric-based monitoring approach by adding a detailed event-level perspective to the observability platform. While metrics provide aggregated numerical indicators of system behavior, logs offer a deeper and more descriptive view of what actually occurs inside the monitored services.

By combining metrics collected through Prometheus with log streams managed by Loki and Promtail, the monitoring infrastructure developed in this project achieves a more complete observability model. This approach allows system operators to analyze both quantitative performance indicators and qualitative event descriptions, significantly improving the ability to detect, diagnose, and resolve operational issues within the monitored environment.

9.2 Log Management (Loki and Promtail)

Base metrics represent the core indicators used to evaluate the operational state of the monitored infrastructure. In this project, metrics are collected at both host and container levels through Node Exporter and cAdvisor.

The collected metrics include CPU utilization, memory usage, disk activity, and network traffic, providing a comprehensive view of system resource consumption. Container-level metrics further enable the analysis of resource usage per service, allowing more precise identification of performance bottlenecks.

All metrics are stored as time-series data in Prometheus, enabling both real-time monitoring and historical analysis. These metrics serve as the foundation for dashboards and alerting mechanisms within the monitoring platform.

Metrics collected by the monitoring system are typically stored as time-series data. Time-series data consists of sequences of measurements indexed by

timestamps, allowing system administrators to analyze how system metrics evolve over time. Time-series analysis enables the detection of performance trends, seasonal patterns, and anomalous behaviors that may indicate potential system issues.

Within the monitoring platform implemented in this project, base metrics represent the foundation of the monitoring process. These metrics serve as the primary input for dashboards, alerting systems, and analytical tools used to evaluate system performance.

9.3 Prometheus Integration

To manage the collection and analysis of monitoring metrics, the observability platform integrates Prometheus as the primary monitoring system.

Prometheus has become one of the most widely used monitoring systems in cloud-native environments due to its scalability, flexibility, and strong integration with container-based infrastructures. The monitoring system operates using a pull-based data collection model, in which the Prometheus server periodically retrieves metrics from monitored services through HTTP endpoints.

These endpoints expose metrics in a standardized format that can be easily interpreted by the monitoring system. Each monitored service exposes a metrics endpoint containing telemetry data describing the operational state of the application or infrastructure component. Prometheus periodically queries these endpoints and stores the retrieved metrics within its internal time-series database.

Prometheus uses a pull-based model to collect metrics from monitored services, simplifying the addition of new targets and enabling basic availability checks. First, it simplifies the process of adding new monitored targets, as the monitoring system only needs to be configured with the address of the metrics endpoint. Second, the pull model allows the monitoring system to verify the availability of monitored services, since failed metric requests may indicate connectivity issues or service failures.

Another key feature of Prometheus is its ability to support dynamic service discovery. In containerized environments where services may be created or destroyed dynamically, the monitoring system must be able to adapt to changes in the infrastructure. Prometheus can automatically discover new monitor-

ing targets and begin collecting metrics from them without requiring manual configuration.

Once metrics are collected, they are stored within the Prometheus time-series database. This database is optimized for storing large volumes of telemetry data and supports efficient querying of historical metrics. The stored data can then be analyzed using the Prometheus Query Language (PromQL), a powerful query language designed specifically for time-series data.

PromQL allows users to perform complex queries on telemetry data, including aggregation, filtering, and mathematical operations on metrics. Through PromQL expressions, operators can compute statistics such as averages, rates of change, and statistical deviations across monitoring metrics. These capabilities enable advanced analysis of system behavior and support the creation of sophisticated monitoring dashboards.

Prometheus also supports the implementation of alerting mechanisms. Alerts can be defined through PromQL expressions that evaluate specific conditions on collected metrics. When a condition is satisfied, the monitoring system triggers an alert that can notify system administrators about potential issues affecting the monitored infrastructure.

Alert notifications can be configured to trigger automated responses or send notifications to monitoring dashboards, messaging platforms, or incident management systems. This capability allows system administrators to respond quickly to abnormal conditions and reduce system downtime.

Once logs are collected and stored, they can be correlated with monitoring metrics to provide deeper insights into system behavior. For example, an increase in application error rates observed through monitoring metrics may correspond to specific error messages recorded in the application logs. By correlating these telemetry signals, system administrators can more easily diagnose the root causes of operational issues.

Visualization tools play a crucial role in presenting monitoring data in a meaningful way. Metrics collected by Prometheus and logs aggregated by the log management system can be visualized through dashboard platforms such as Grafana. These dashboards allow users to explore telemetry data interactively and observe system behavior through charts, graphs, and monitoring panels.

Grafana dashboards provide a powerful interface for monitoring system per-

formance. Operators can create customized dashboards that display specific metrics related to system resources, application activity, and infrastructure health. These dashboards transform raw telemetry data into visual representations that are easier to interpret and analyze.

The integration between Prometheus, log aggregation systems, and visualization platforms creates a comprehensive monitoring environment capable of supporting advanced observability operations. By combining metrics monitoring, log analysis, and visual dashboards, the monitoring infrastructure implemented in this project provides a powerful toolset for analyzing system behavior and maintaining the reliability of distributed systems.

Grafana provides the visualization layer of the monitoring platform, enabling the representation of metrics and logs through interactive dashboards. It is integrated with Prometheus and Loki, allowing users to query and visualize both time-series data and log streams within a unified interface.

The design of the dashboards represented a crucial phase in the development of the monitoring platform. While Prometheus is responsible for collecting and storing large volumes of metrics, the usefulness of these metrics strongly depends on how effectively they can be visualized and interpreted by system operators. Well-designed dashboards enable engineers to quickly understand the current state of the system, identify abnormal behavior, and detect potential issues before they escalate into operational failures.

In the context of the monitoring system developed in this work, several dashboards were created in order to analyze different aspects of the infrastructure and the telemetry data generated by the IoT Edge devices. These dashboards were designed to provide both a high-level overview of the system status and more detailed views of specific components of the monitoring stack.

The dashboards typically included time-series graphs representing the evolution of specific metrics over time. These visualizations allowed operators to observe trends, identify recurring patterns, and detect anomalies in the behavior of the monitored services. For instance, metrics related to log generation rates, error frequencies, and system resource usage were visualized through dynamic charts capable of updating automatically as new telemetry data was collected.

Another important aspect of the dashboard design concerned the ability to correlate different sources of telemetry data. In observability platforms, meaningful insights often emerge not from individual metrics but from the correlation

between multiple signals such as logs, metrics, and traces. Grafana facilitates this type of analysis by allowing multiple panels within the same dashboard to display related information simultaneously. This capability enables operators to compare different metrics over the same time interval and better understand the relationships between different events occurring in the system.

During the development of the monitoring dashboards, particular attention was given to the selection of appropriate visualization types for different categories of metrics. Time-series graphs were used to display continuous telemetry data, while other visual components such as gauges and counters were used to highlight the current value of specific system indicators. This variety of visualization techniques contributed to improving the readability of the dashboards and making them more suitable for operational monitoring scenarios. [3].

Another advantage of Grafana is its ability to support real-time data visualization. By periodically querying the Prometheus data source, Grafana dashboards can automatically refresh their content at predefined intervals, allowing operators to observe the evolution of system metrics almost immediately after they are generated. This capability is particularly important in environments where rapid detection of anomalies and system failures is required.

In addition to real-time visualization, Grafana also provides tools for exploring historical data stored in Prometheus. Users can adjust the time range displayed in a dashboard in order to analyze past system behavior, compare different time periods, and identify long-term trends in the collected telemetry data. This feature is particularly useful when performing retrospective analysis of incidents or investigating the root causes of anomalies detected during system operation.

Overall, the integration between Prometheus and Grafana enabled the creation of a comprehensive monitoring environment in which raw telemetry data could be transformed into actionable insights. The dashboards developed for this project played an essential role in supporting the analysis of system behavior and in facilitating the evaluation of the monitoring infrastructure implemented within the IoT Edge environment.

9.4 Advantages of the Monitoring Infrastructure

The monitoring infrastructure implemented in this project provides several practical advantages. Centralized data collection enables unified monitoring of

distributed edge nodes, while the integration of metrics and logs improves the ability to diagnose system issues.

The use of a containerized monitoring stack ensures modularity and scalability, allowing the system to adapt to increasing workloads. Additionally, real-time visualization through Grafana dashboards enhances system observability and supports rapid detection of anomalies.

These characteristics make the monitoring platform effective for managing distributed IoT environments and analyzing system behavior under different operational conditions.

9.5 Monitoring Results

The implemented monitoring platform enabled real-time observation of system behavior across both edge and master nodes. Metrics collected through Prometheus and visualized in Grafana dashboards allowed the identification of resource usage patterns and system performance trends.

In particular, CPU and memory utilization metrics highlighted variations in workload across different containers, while log analysis through Loki enabled the detection of warning and error conditions affecting system components.

The correlation between metrics and logs proved to be especially useful for diagnosing anomalies. For example, increases in error rates observed in Prometheus metrics were matched with specific log entries in Loki, allowing rapid identification of the root causes of system issues.

Overall, the monitoring infrastructure demonstrated the ability to provide a comprehensive and real-time view of the distributed system, supporting both operational monitoring and post-event analysis.

10 Experimental Results and Conclusions

The implementation of the monitoring and observability platform described in the previous chapters allowed the evaluation of the system within a distributed environment composed of containerized services and edge monitoring nodes.

This chapter presents the results obtained during the experimental phase of the project, highlighting both the operational performance of the monitoring infrastructure and the main challenges encountered during system development and deployment.

The experimental evaluation focused on assessing the reliability of the monitoring infrastructure, the effectiveness of the integrated observability tools, and the overall capability of the system to provide meaningful insights into distributed environments.

In addition, the experiments explored the potential role of machine learning techniques as a complementary component for telemetry analysis. Although not implemented as a full production pipeline, the integration demonstrated how data-driven approaches can enhance traditional monitoring strategies.

The following sections describe the main issues encountered during system implementation, present the results obtained during the monitoring experiments, and discuss the implications of these results for the design of observability systems in distributed environments. [\[1\]](#).

10.1 Errors Encountered

The encountered issues can be grouped into three main categories: configuration errors, integration challenges, and data collection limitations.

During the development and deployment of the monitoring infrastructure, several technical challenges and operational issues were encountered. These challenges were primarily related to the integration of different monitoring tools, the configuration of containerized services, and the management of telemetry data generated by distributed systems.

One of the first difficulties encountered concerned the configuration of containerized services within the monitoring stack. Because the monitoring platform was composed of multiple interdependent components, correct configuration of container networking and service communication was essential. In several cases, misconfigured container networks prevented monitoring services from communicating properly, resulting in incomplete telemetry collection. (e.g., incorrect bridge network configuration or missing port mappings).

Another critical challenge involved the integration of heterogeneous observability tools within the same infrastructure. Tools responsible for metrics collection, log aggregation, and visualization often require specific configuration parameters and compatible data formats. Ensuring proper interoperability between these components required careful configuration of data pipelines and service endpoints.

Additional issues were related to telemetry data collection in unstable network conditions. In distributed environments, edge nodes may experience network instability or temporary connectivity interruptions. These conditions can affect the transmission of telemetry data to the central monitoring platform and may lead to gaps in the collected monitoring information.

Another category of problems involved the configuration of monitoring agents and exporters responsible for exposing system metrics. In some cases, exporters were not correctly configured or did not expose the expected metrics endpoints, preventing the monitoring system from retrieving telemetry data. Resolving these issues required verification of service configurations and adjustments to the monitoring targets defined in the monitoring platform.

Additionally, challenges were encountered during the configuration of visualization dashboards. Although monitoring platforms provide powerful visualization capabilities, designing dashboards that effectively represent system behavior requires careful selection of metrics and appropriate graphical representations. Several iterations were necessary to create dashboards that provided clear and useful insights into system performance.

Despite these challenges, the debugging process contributed significantly to improving the stability and robustness of the monitoring infrastructure. By identifying configuration issues and optimizing system components, it was possible to achieve a functional observability platform capable of collecting and analyzing telemetry data from distributed environments.

10.2 Results

After resolving the main configuration issues encountered during the development phase, the monitoring infrastructure was successfully deployed and tested within the experimental environment. The system demonstrated the ability to collect, process, and visualize telemetry data generated by distributed services and edge

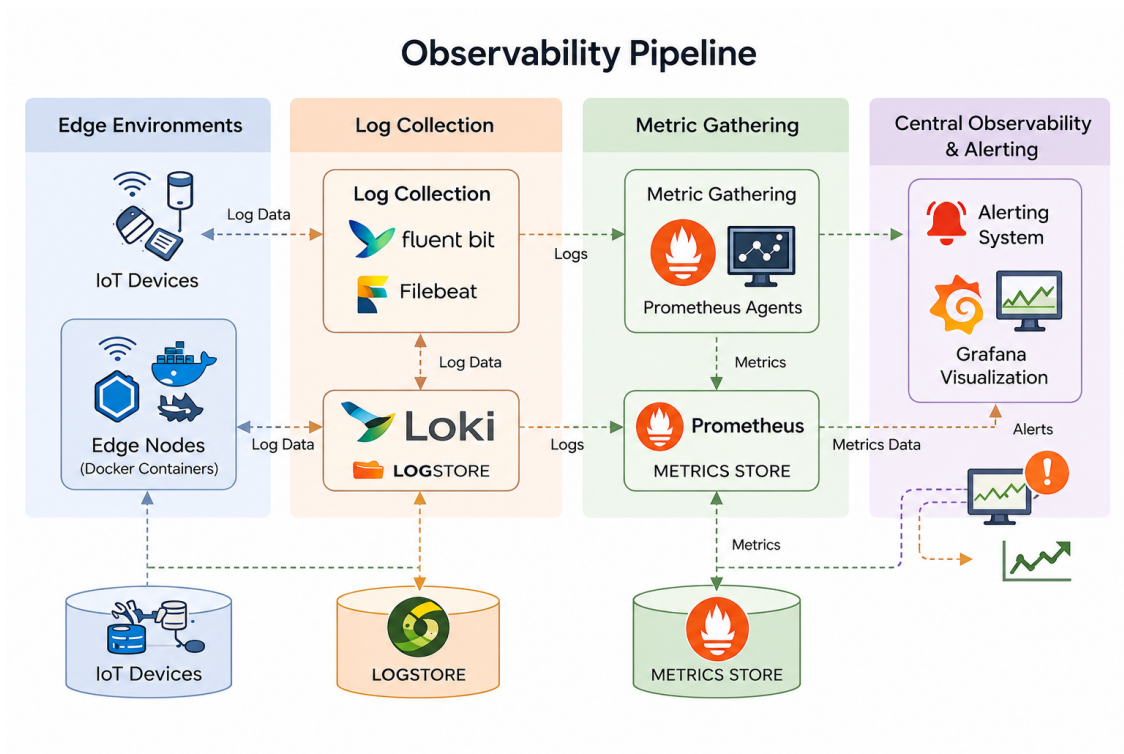


Figure 13: Data flow of the complete observability pipeline.

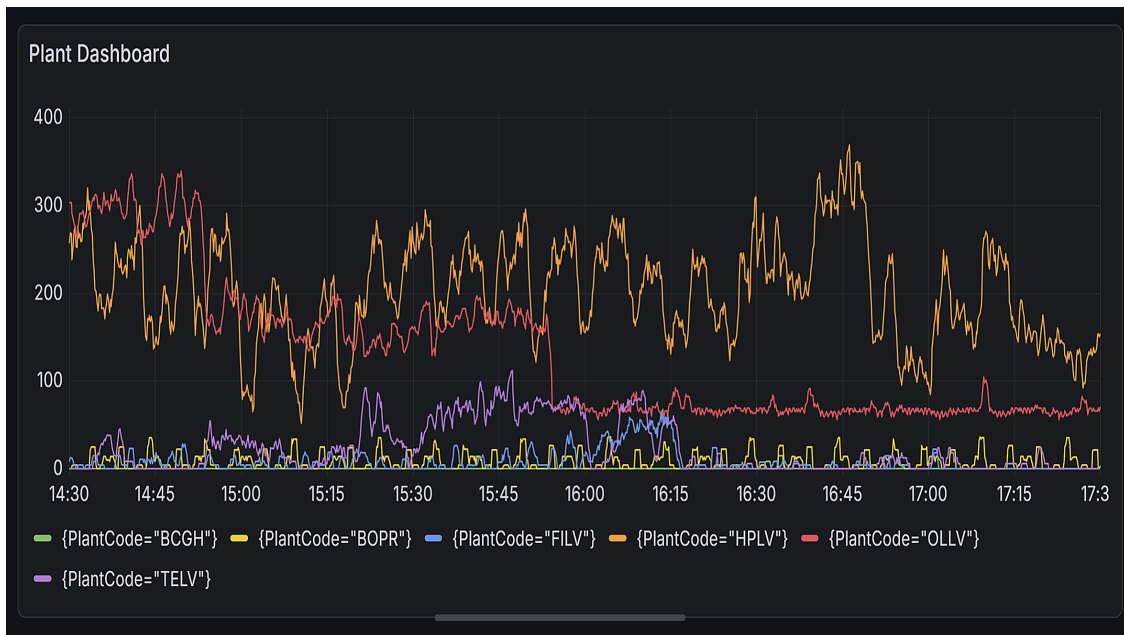


Figure 14: Log rate for each plant.

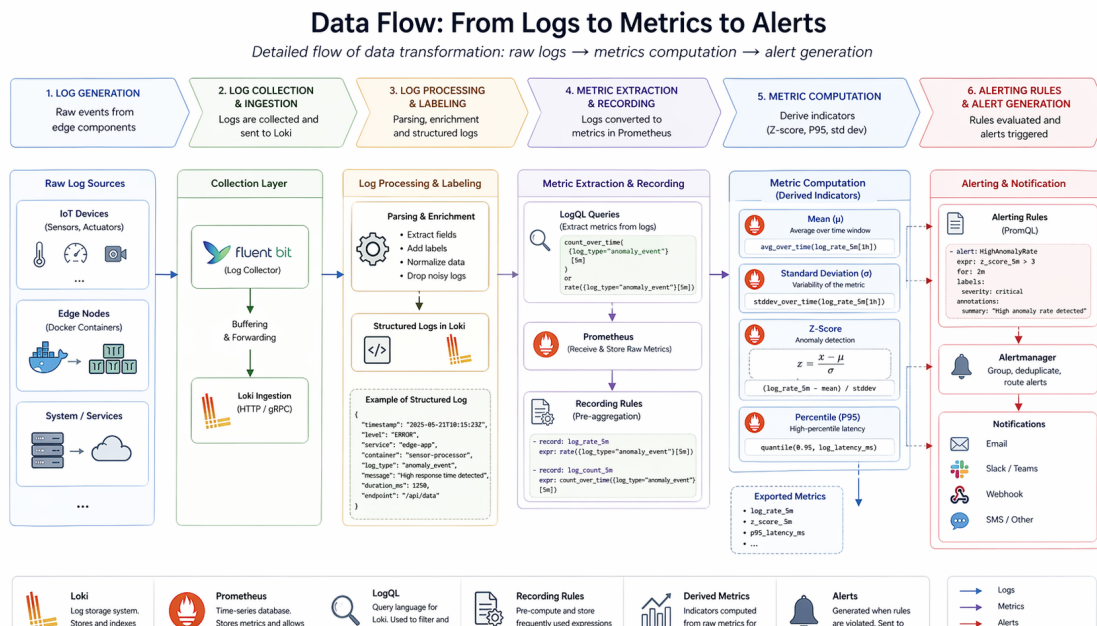


Figure 15: Data flow within the observability pipeline.

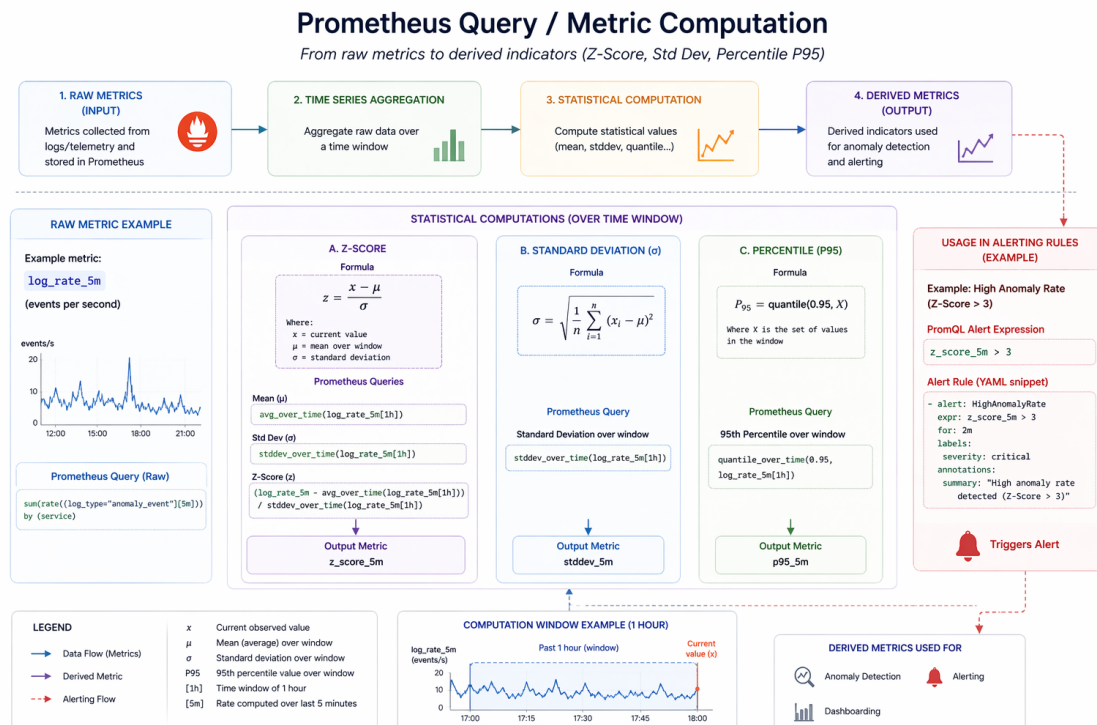


Figure 16: Prometheus computation metric pipeline.

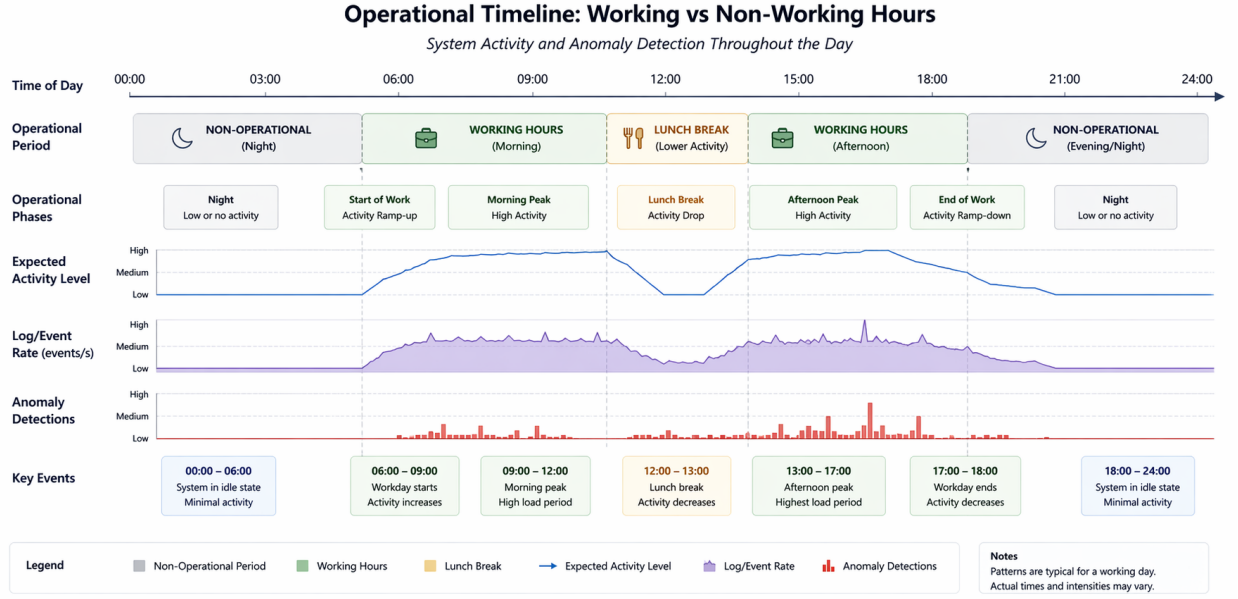


Figure 17: Distinction between working and non-working hours.

nodes. To better illustrate the overall data flow within the monitoring system, Figure 18 presents the complete observability pipeline, from data generation at the edge to centralized analysis and alerting. To better understand how telemetry data is transformed within the system, Figure 15 illustrates the detailed data flow from raw log collection to metric computation and alert generation.

It is important to note that the detailed metric analysis based on Prometheus (including Z-score, 95% percentile, standard deviation, and alert generation) was conducted on a subset of plants, specifically BOPR and FILV.

This choice was driven by both architectural and performance considerations. The computation of advanced statistical metrics requires complex Prometheus queries involving aggregation, rate calculations, and time-window operations. When applied across all available plants and machines, these queries can lead to a significant increase in the number of time series, resulting in higher computational overhead, increased latency, and, in some cases, incomplete or missing visualizations.

To ensure reliable and interpretable results, the analysis was therefore restricted to a representative subset of plants. BOPR and FILV were selected as they provide a sufficiently heterogeneous and realistic sample of the system behavior, allowing meaningful evaluation of the monitoring pipeline while maintaining acceptable query performance.

In contrast, the global log rate visualization (Plant Dashboard) includes all plants, as it relies on simpler aggregation queries that are less computationally intensive. This enables a complete overview of system activity without incurring the performance limitations associated with more advanced statistical computations.

Extending the same level of analysis to all plants would require deploying additional components, such as metric exporters or log processing agents (e.g., Fluent Bit), across all monitored environments. This would increase system complexity and resource usage, and was therefore not implemented in the current experimental setup.

The analysis of log rate aggregated over 5-minute intervals reveals a highly dynamic and non-uniform behavior across the monitored plants. The results show a strong imbalance in log generation, with one plant (BOPR) dominating the overall activity. In particular, the BOPR plant exhibits values ranging approximately from 300 up to over 1300 log entries per 5-minute window, significantly higher than the other monitored plants, which typically remain below 100 log entries. This indicates that system activity is not evenly distributed, but rather concentrated in specific components of the infrastructure.

From a temporal perspective, the system behavior can be divided into multiple phases. An initial phase shows a gradual increase in log activity, followed by a sudden drop in log volume, where values briefly approach zero. This abrupt transition may indicate a temporary service interruption, a configuration reset, or a change in operational conditions. A subsequent phase is characterized by moderate activity levels, while the final portion of the observation window shows sustained high log rates, with repeated peaks exceeding 1200 log entries.

This suggests periods of intensified system activity or increased load. These observations highlight the non-stationary nature of the monitored system, where workload and logging behavior vary significantly over time. The ability to capture such variations through rate-based metrics is essential for identifying both normal operational patterns and potential anomalies.

The system was able to collect metrics at regular intervals without significant data loss, even under moderate network instability conditions. No critical failures were observed during extended monitoring sessions, confirming the stability of the monitoring infrastructure under sustained operation.

During testing, metrics were successfully collected at regular intervals (e.g., 5-second scraping intervals) with minimal data loss. Dashboard refresh times

remained within acceptable limits, enabling near real-time visualization of system behavior.

A more detailed analysis was performed using log aggregation over 1-minute intervals, focusing on a three-hour window within the working period. This higher-resolution view allows a more precise characterization of system dynamics and inter-plant variability. The results show that log generation is not uniformly distributed across plants. In particular, the HPLV plant exhibits the highest activity levels, with values ranging approximately between 150 and 350 log entries per minute.

In contrast, the OLLV plant shows a more stable behavior, with log rates generally between 60 and 180 log entries per minute. Other plants, including TELV, FILV, and BCGH, remain at significantly lower levels, often below 100 log entries per minute. This confirms that system workload is highly dependent on the specific plant and is not evenly balanced across the monitored infrastructure.

From a temporal perspective, the observed interval—corresponding to a portion of the working day—reveals continuous fluctuations in log generation. The most active plant (HPLV) exhibits rapid oscillations, suggesting a dynamic workload characterized by frequent state changes or discrete processing events.

In contrast, other plants maintain relatively stable patterns, indicating more consistent operational behavior. Despite these variations, no prolonged periods of inactivity or system downtime were observed. This indicates that the monitoring infrastructure remains stable and continuously operational during the analyzed time window.

Overall, this analysis highlights the importance of fine-grained temporal monitoring, as system behavior can vary significantly even within short time intervals and during the same operational phase. The ability to capture such variations is essential for accurately understanding system dynamics and identifying potential anomalies. As shown in Figure 14, the log rate trend over 1-minute intervals highlights significant variability across plants, with HPLV exhibiting the highest activity levels.

These observations suggest that anomaly detection and alerting mechanisms should be adapted to the specific behavior of each monitored component, rather than relying on uniform thresholds across the entire system.

In scenarios involving moderate network instability, temporary delays in teleme-

try transmission were observed, but the system was able to recover without permanent data inconsistencies.

One of the key outcomes of the experimental evaluation was the successful collection of system metrics across the monitored infrastructure. Metrics related to resource usage, system activity, and service performance were continuously collected and stored within the monitoring platform. These metrics provided a real-time overview of system behavior and allowed the identification of operational trends over time.

The monitoring dashboards implemented within the visualization platform proved effective in presenting telemetry data in a clear and accessible manner. Through graphical panels and time-series charts, system operators were able to monitor resource utilization, observe service activity, and detect potential anomalies within the system.

Another important result concerned the integration between the monitoring platform and the containerized infrastructure. The use of container-based deployment allowed the monitoring services to operate consistently across different environments and simplified the management of system components. The modular nature of the architecture also made it easier to add new monitoring targets or modify system configurations when necessary.

Before presenting the results, it is important to describe how the derived metrics are computed from raw telemetry data. Figure 16 illustrates the Prometheus-based computation process for key statistical indicators, including Z-score, standard deviation, and the 95% percentile.

To better interpret the observed system behavior, it is useful to contextualize the analysis within the operational timeline of the system. Figure 17 illustrates the distinction between working hours, lunch breaks, and non-operational periods.

The Z-score metric was used to normalize the log ingestion rate and detect deviations from the expected system behavior. This metric expresses how far a given observation deviates from the mean in terms of standard deviation, enabling the identification of anomalous patterns independently of absolute log volumes. The observed values range approximately between -4 and +4, with the majority of observations concentrated within the interval $[-2, +2]$, which corresponds to normal operating conditions. However, several peaks exceeding ± 3 are visible throughout the entire observation window, indicating the presence of statistically significant anomalies.

A clear difference can be observed between non-operational and operational periods. During non-working hours, the Z-score distribution appears more compact, with fewer extreme deviations. In contrast, during working hours, the density and magnitude of the peaks increase, reflecting a higher frequency of anomalous events. In particular, the afternoon working period shows the highest variability, with more frequent excursions beyond ± 2 and multiple peaks approaching ± 4 . This suggests that the system is more prone to instability or transient anomalies during intensive operational phases. As shown in Figure 18, these anomalies are significantly more frequent during working hours.

The temporal analysis of the 95th percentile metric over the full working day reveals a clear correlation between system activity and operational time periods. During the morning working hours, the percentile values remain relatively low and stable, typically within the range of 0 to 3, indicating that even the upper tail of the monitored metric does not deviate significantly from baseline conditions. A further reduction in variability is observed during the lunch break interval, where percentile values remain consistently low.

This behavior suggests a reduced system workload and limited log activity during non-operational periods. In contrast, a significant change occurs during the afternoon working session. Starting from the early afternoon, multiple machine-level time series exhibit a sharp increase in 95% percentile values, reaching peaks between approximately 15 and 18. These elevated values persist over extended periods and involve multiple machines simultaneously.

This pattern indicates that abnormal or high-intensity behavior is not limited to isolated events, but rather corresponds to sustained operational conditions affecting a subset of machines. The strong temporal alignment with working hours suggests that these anomalies are directly related to production activity rather than random system noise.

From a statistical perspective, the observed increase in the 95% percentile during the afternoon indicates a shift in the upper tail of the distribution of the monitored metric. While the morning phase is characterized by low dispersion and limited extreme values, the afternoon phase shows a clear expansion of the upper tail, reflecting repeated occurrences of elevated metric values.

This behavior confirms that the percentile-based approach effectively captures persistent abnormal conditions, distinguishing them from isolated spikes that would not significantly affect the distribution. The simultaneous increase across

multiple machine streams further suggests the presence of coordinated system-level dynamics or shared operational stress conditions. As shown in Figure 20, the 95% percentile significantly increases during afternoon working hours, indicating sustained high-intensity system activity.

The analysis of the standard deviation of the log ingestion rate provides additional insight into the stability of the monitored system. Unlike raw counts or percentile-based metrics, the standard deviation captures the variability of the log generation rate over time, highlighting fluctuations in system behavior.

During non-operational periods, the observed standard deviation remains consistently low, typically within the range of 0.05 to 0.15, indicating a stable and predictable system state. A slight increase is observed during the morning working hours, where values range approximately between 0.1 and 0.3, reflecting moderate and controlled activity. A significant change emerges during the afternoon working period.

In this phase, the standard deviation increases substantially, reaching values between 0.6 and 1.2, with peaks exceeding 1.5. This indicates a high degree of variability in the log generation rate, suggesting that the system is subject to rapid and frequent fluctuations. The transition between these regimes is particularly sharp. Immediately after the end of the working period, the standard deviation quickly returns to baseline levels, confirming that the observed variability is strongly correlated with operational activity rather than intrinsic system noise.

From an operational perspective, the increase in standard deviation during active production hours indicates that the system does not operate under steady-state conditions, but rather exhibits dynamic and heterogeneous behavior. This variability may be caused by differences in machine workload, intermittent processing phases, or the occurrence of transient events affecting multiple components. When considered together with the 95th percentile analysis, these results suggest that the system experiences not only higher levels of activity during working hours, but also increased instability. In particular, the coexistence of elevated percentile values and high standard deviation indicates that abnormal conditions are both intense and persistent, rather than isolated. As shown in Figure 19, the standard deviation increases significantly during working hours, reflecting higher variability in system behavior.

This behavior suggests that monitoring systems should not rely solely on average

or cumulative metrics, but also incorporate variability indicators to better capture dynamic system conditions.

The alerting system provides a discrete representation of anomaly detection events by reporting when specific conditions are met. Each point in the visualization corresponds to an alert in the "firing" state, indicating that a predefined threshold has been exceeded. The alerts are distributed across the entire observation period, but their density varies significantly between non-operational and operational phases. During non-working hours, alert events appear sparse and isolated, reflecting stable system conditions.

In contrast, during working hours, the frequency of alerts increases substantially, with a dense concentration of events, indicating a higher level of system activity and a greater likelihood of anomalous behavior.

Two main categories of alerts can be identified. The first category, labeled as "IoTLogP95Spike", is associated with warning-level severity and reflects sustained increases in the upper tail of the monitored metric. These alerts indicate that the system is experiencing elevated activity levels but not necessarily critical conditions. The second category, labeled as "IoTLogZScoreHigh", corresponds to critical severity and identifies statistically significant deviations from expected behavior, often associated with abrupt or extreme anomalies. The coexistence of warning and critical alerts demonstrates the ability of the monitoring system to distinguish between gradual performance degradation and sudden anomalous events. This hierarchical alerting strategy improves the interpretability of system behavior and supports both proactive and reactive operational responses.

From an operational perspective, the observed alert distribution confirms that anomaly detection is closely correlated with system workload. The higher concentration of alerts during working hours suggests that increased activity leads to more frequent deviations from baseline behavior. Moreover, the presence of repeated alerts across multiple machines indicates that anomalies are not confined to isolated components, but may affect broader portions of the system. This highlights the importance of centralized monitoring and cross-machine correlation analysis.

The alerting behavior is consistent with the patterns observed in the Z-score and 95th percentile analyses. Periods characterized by elevated percentile values and increased statistical deviation correspond to intervals in which alerts are triggered

more frequently. This confirms the coherence of the monitoring pipeline, in which raw logs are transformed into statistical indicators and subsequently into actionable alert events.

The alerting layer represents the final stage of the observability pipeline, transforming statistical anomalies into actionable events. The results demonstrate that the system is capable of continuously detecting and classifying anomalies in real time, enabling both early warning and critical incident identification in a complex industrial environment. As shown in Figure 21, the standard deviation increases significantly during working hours, reflecting higher variability in system behavior.

The experiments also demonstrated the benefits of centralized telemetry aggregation. By collecting monitoring data from multiple edge nodes and transmitting it to a central platform, the system provided a unified view of the distributed infrastructure. This capability is particularly important in IoT environments, where devices may be deployed across geographically distributed locations.

Preliminary experiments involving machine learning techniques were conducted on historical telemetry data. These experiments showed that simple anomaly detection approaches could identify deviations in system behavior, although further validation and tuning would be required for production use.

Overall, the results demonstrate that anomaly-related metrics are strongly influenced by operational workload patterns. The highest levels of abnormal behavior are observed during active production phases, while non-operational periods remain largely stable. This finding highlights the importance of contextualizing monitoring data within the operational schedule of the system.

Without this temporal perspective, periods of high activity could be misinterpreted as anomalous, whereas they may represent expected system behavior under load. At the same time, the persistence and magnitude of the observed percentile peaks suggest that certain machines are more prone to sustained abnormal conditions, indicating potential targets for further investigation or optimization.

This behavior suggests that static threshold-based alerting may not be sufficient, and that adaptive or context-aware monitoring strategies could further improve anomaly detection accuracy.

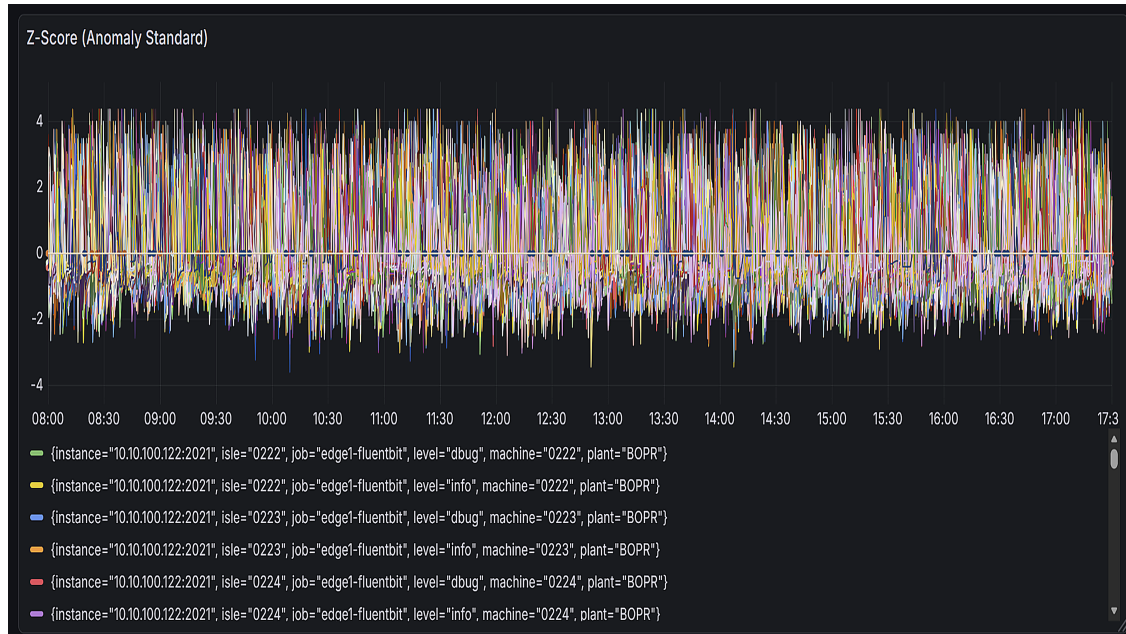


Figure 18: Z-score metric for each plant.

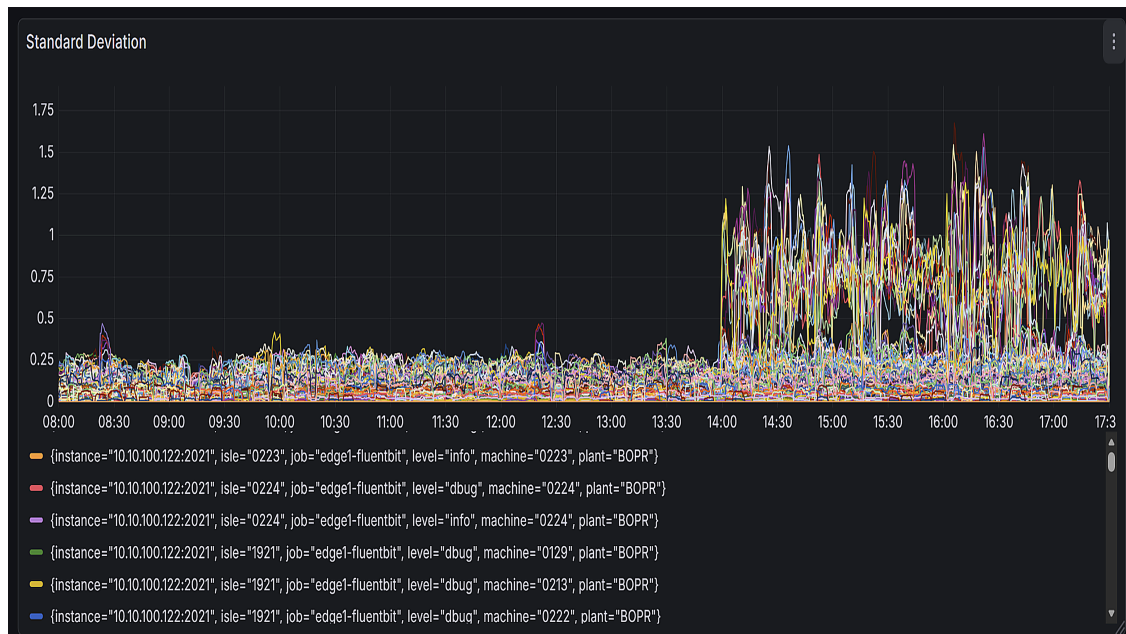


Figure 19: Standard deviation for each plant.

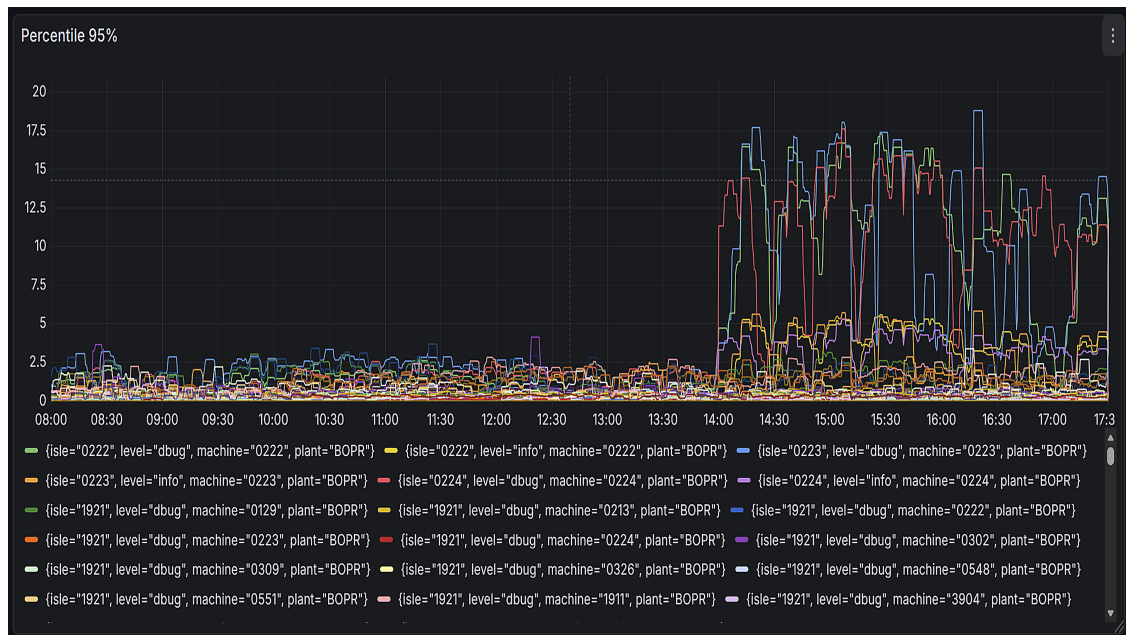


Figure 20: Percentile 95% metric analysis for each plant.



Figure 21: Alert anomaly analysis.

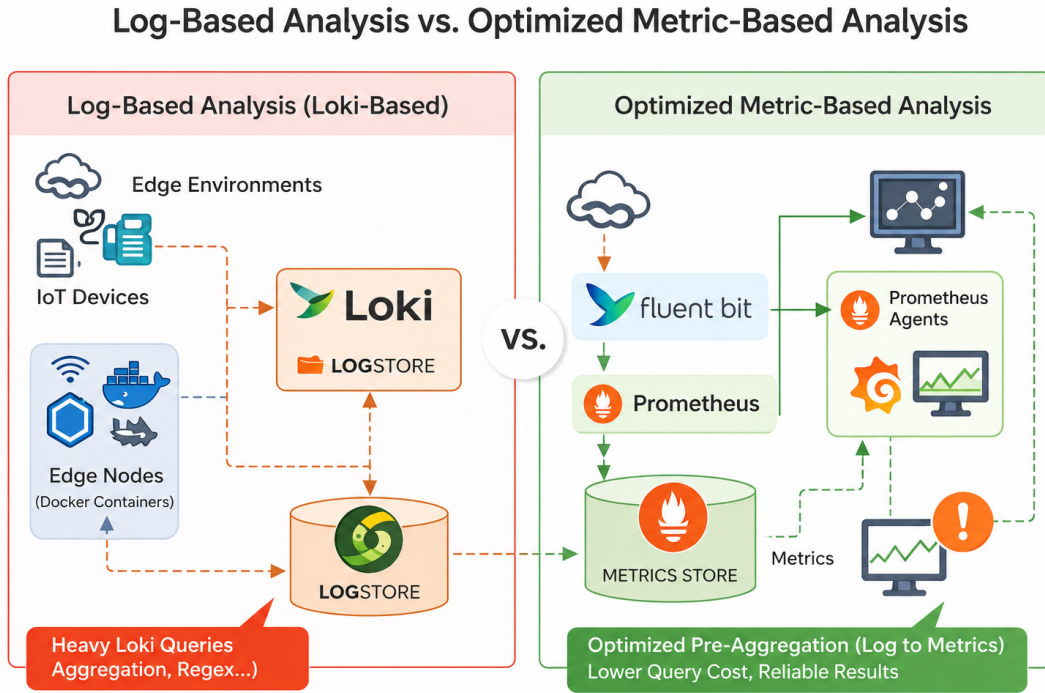


Figure 22: Comparison between log-based analysis using Loki and the optimized metric-based approach using Prometheus.

10.3 Discussion and Conclusion

The experimental results demonstrate that the proposed monitoring architecture is capable of effectively supporting observability in distributed and containerized environments. The system successfully achieved continuous telemetry collection, reliable data aggregation, and real-time visualization of system behavior.

The detailed analysis of log-derived metrics, including Z-score, percentile values, and alert distribution, provides deeper insights into system behavior beyond simple telemetry visualization. A key limitation identified during the experimental phase concerns the use of Loki as a data source for aggregated log analysis. While Loki is highly effective for log storage and retrieval, it is not specifically optimized for complex analytical queries over large time ranges, particularly when regular expressions and high-cardinality filters are involved.

During testing, it was observed that queries executed on the Loki data source may exhibit performance degradation as the time range increases. In some cases, this results in incomplete visualizations or flattened trends, which can affect the interpretability of long-term analysis. This behavior is not due to actual

data loss, but rather to the computational cost and internal query limitations associated with log-based aggregation.

To address this issue, a hybrid approach was adopted, where raw logs are processed using log collectors and transformed into structured metrics. These metrics are then exported to Prometheus, which is specifically designed for efficient time-series aggregation and long-term analysis. The architectural transition from a purely log-based approach to a hybrid log-to-metric pipeline is illustrated in Figure 22, which highlights the differences between the initial Loki-based analysis and the optimized Prometheus-based solution.

This approach significantly improves query performance and enables more reliable visualization of trends over extended time periods, while still preserving the full detail of raw logs within Loki for debugging and root-cause analysis.

Another relevant limitation encountered during the experimental evaluation concerns the execution of computationally expensive queries on large time intervals. Several dashboards relying on complex log-based queries were unable to return results when the selected time range exceeded a few hours.

In particular, it was observed that queries involving aggregation functions combined with regular expressions could only reliably visualize data within a limited time window (approximately up to three hours). Beyond this threshold, the system either returned no results or significantly degraded in responsiveness.

This limitation is primarily related to the computational overhead of processing large volumes of log data in real time, as well as resource constraints within the monitoring stack. It highlights the importance of designing efficient queries and reinforces the need for pre-aggregated metrics when long-term analysis is required.

To mitigate these limitations, an architectural optimization was introduced by leveraging log processing agents such as Fluent Bit. Instead of performing heavy aggregations directly on log data within Loki, selected log fields were transformed into numerical metrics at ingestion time and forwarded to Prometheus.

This design reduces the computational burden on the log querying layer and enables efficient time-series analysis using Prometheus, while maintaining the full fidelity of raw logs within Loki. As a result, the monitoring system achieves a balance between detailed log inspection and scalable analytical capabilities.

This design choice highlights a trade-off between scalability and analytical depth. While it is theoretically possible to extend advanced metric computation to all plants, in practice this would introduce significant performance challenges and infrastructure overhead. Therefore, the adopted approach balances system-wide visibility with detailed analysis on a controlled subset of data sources.

Furthermore, this decision is consistent with the observed limitations of log-based queries in Loki, where complex aggregations and regex-based filtering may lead to performance degradation and loss of detail over extended time ranges. By delegating advanced metric computation to Prometheus and limiting the scope of analysis, the system achieves more stable and reliable results.

The selective analysis of a subset of plants represents a deliberate architectural decision aimed at balancing computational efficiency and analytical depth, ensuring both system scalability and meaningful interpretation of monitoring results.

One of the key strengths of the implemented solution lies in its modular architecture. The use of containerization enabled flexible deployment and simplified the integration of multiple monitoring components. This modularity also allows the system to be extended with additional services without significant architectural changes.

The adoption of a centralized telemetry aggregation approach proved essential for managing distributed edge nodes. By collecting metrics and logs into a unified platform, the system provides a global view of infrastructure behavior, improving the ability to detect and diagnose anomalies.

However, the experimental phase also highlighted some limitations. The system remains sensitive to network instability, which can temporarily affect telemetry transmission from edge nodes. Additionally, the configuration of multiple integrated tools requires careful setup and may introduce complexity during deployment.

From an analytical perspective, the integration of visualization tools significantly improved the interpretability of telemetry data. Dashboards enabled rapid identification of abnormal patterns and supported efficient monitoring operations.

The exploratory use of machine learning techniques suggests potential future improvements in anomaly detection. While the current implementation re-

mains preliminary, it demonstrates how data-driven approaches could enhance traditional monitoring systems by identifying complex patterns in telemetry data.

Future work could focus on improving the robustness of data collection in unstable network conditions, optimizing dashboard design, and developing a more advanced machine learning pipeline for predictive monitoring.

Overall, the results demonstrate that system behavior is strongly influenced by workload conditions, with anomaly patterns emerging primarily during active operational phases. This highlights the importance of adaptive and context-aware monitoring strategies, as static threshold-based approaches may not be sufficient in dynamic industrial environments.

At the same time, the identified limitations emphasize that real-world observability systems must balance analytical depth, computational efficiency, and scalability. These findings confirm that effective observability requires not only robust data collection, but also scalable architectures capable of adapting to dynamic workloads and operational constraints.

References

- [1] Hitesh Allam. “Metrics That Matter: Evolving Observability Practices for Scalable Infrastructure”. In: *International Journal of AI, BigData, Computational and Management Studies* 3.3 (2022), pp. 52–61.
- [2] P Bhosale. “Metrics, Logs, and Traces: A Unified Approach to Observability in Microservices”. In: *Journal of Artificial Intelligence, Machine Learning and Data Science* 1 (2022), pp. 2084–2088.
- [3] Rob Chapman and Peter Holmes. *Observability with Grafana: Monitor, control, and visualize your Kubernetes and cloud platforms using the LGTM stack*. Packt Publishing Ltd, 2024.
- [4] Radoslav Gatev. *Introducing Distributed Application Runtime (Dapr)*. Springer, 2021.
- [5] B Gregg and BPF Performance Tools. *Linux System and Application Observability*. 2019.
- [6] Michael Hausenblas. *Cloud Observability in Action*. Simon and Schuster, 2024.
- [7] Charity Majors, Liz Fong-Jones, and George Miranda. *Observability engineering*. " O'Reilly Media, Inc.", 2022.

Acknowledgements

“I would like to express my sincere gratitude to my supervisor, Professor Nicola Bicocchi, and my co-supervisor, Professor Marco Picone, for their guidance, patience, and dedication throughout the development of this project and the preparation of this thesis.

I would also like to thank Bondioli & Pavesi for giving me the opportunity to work on this project. Special thanks go to Diego Veneri for his support, valuable advice, and continuous assistance throughout the different stages of the project.

Finally, I am deeply grateful to my family for their unwavering support and encouragement. Especially during the most difficult moments of this journey, they inspired me to persevere, give my best, and never give up.”