FHV
Vorarlberg University
of Applied Sciences

Refactoring Strategies for Optimizing and Consolidating Telemetry Systems

Master Thesis submitted in Fulfillment of the Degree

Master of Science (MSc)

University of Applied Sciences Vorarlberg
Computer Science

Supervised by
Andrea Janes

Handed in by
Mert Öztürk

Dornbirn, 09 July 2024

Abstract

Many IT organizations require telemetry data to keep an eye on their broad set of applications and underlying infrastructure. Historically, as the market provided specialized tools to capture different kinds of data, teams accumulated multiple monitoring tools for the same or similar purposes. The tool sprawl led to increased complexity, reduced visibility and inefficient resource utilization. Nowadays, the trend is to opt for all-in-one observability solutions (e.g. Datadog, New Relic) to unify the handling of telemetry data. However, many teams need to sacrifice visibility to control the costs when using such platforms.

The company Leica Geosystems faced the same challenges. Due to the absence of an organization-wide observability strategy, a complex telemetry environment emerged, causing not only data silos, but also high operative and maintenance costs. Thus, the aim of this thesis was to find out refactoring strategies for optimizing and consolidating the company's telemetry systems. Next to an extensive literature review, the research methodology was a study composed of four distinct telemetry systems that were analyzed through semi-structured interviews with software engineers, software directors and product managers. Hereby, the Goal-Question-Metric approach helped to define the majority of the interview questions and analysis criteria. After considering best practices and trade-offs, the decision was to go with OpenTelemetry and Grafana Cloud, accompanied by data filtering and transformation strategies to control the costs. Initial evaluation shows that this approach works well for web services and web apps, but might encounter limitations for desktop and mobile apps in future. Although the results and conclusions are specific to the study, the high-level steps to develop an observability strategy may be valuable for other telemetry environments as well.

Kurzreferat

Viele IT-Organisationen benötigen Telemetriedaten, um ihre zahlreichen Anwendungen und die zugrunde liegende Infrastruktur im Auge zu behalten. Da der Markt in der Vergangenheit spezialisierte Tools zur Erfassung verschiedener Arten von Daten anbot, sammelten sich mehrere Monitoring Tools für die gleichen oder einen ähnlichen Zwecke an. Dieser Anhäufung an Tools führte zu erhöhter Komplexität, geringerer Sichtbarkeit und ineffizienter Ressourcennutzung. Heutzutage geht der Trend zu All-in-One Observability-Lösungen (z.B. Datadog, New Relic), um die Handhabung von Telemetriedaten zu vereinheitlichen. Allerdings müssen viele Teams bei der Nutzung solcher Plattformen auf Sichtbarkeit verzichten, um die Kosten zu kontrollieren.

Das Unternehmen Leica Geosystems stand vor denselben Herausforderungen. Aufgrund des Fehlens einer unternehmensweiten Observability-Strategie entstand eine komplexe Telemetrie-Umgebung, die nicht nur Datensilos, sondern auch hohe Betriebs- und Wartungskosten verursachte. Ziel dieser Arbeit war es daher, Refactoring Strategien zur Optimierung und Konsolidierung der Telemetrie-Systeme des Unternehmens zu finden. Neben einer umfangreichen Literaturrecherche bestand die Forschungsmethodik aus einer Studie, die sich aus vier verschiedenen Telemetrie-Systemen zusammensetzte, die durch semistrukturierte Interviews mit Teilnehmern aus der Software- und Produktentwicklungsabteilung analysiert wurden. Dabei half der Goal-Question-Metric-Ansatz, die meisten Interviewfragen und Analysekriterien zu definieren. Nach Abwägung von Best Practices und Trade-Offs fiel die Entscheidung auf OpenTelemetry und Grafana Cloud, begleitet von Methoden zur Datenfilterung und - umwandlung, um die Kosten zu kontrollieren. Erste Evaluierungen zeigen, dass dieser Ansatz für Webserver und Webanwendungen gut funktioniert, bei desktop und mobilen Anwendungen in Zukunft jedoch an Grenzen stoßen könnte. Obwohl die Ergebnisse und Schlussfolgerungen spezifisch für die Studie sind, können die übergeordneten Schritte zur Entwicklung einer Observability-Strategie ebenfalls für andere Telemetrie-Umgebungen wertvoll sein.

Contents

List of Tables

List of Figures

1 Introduction

In modern businesses, data is always flowing and rapidly growing. A lot of this data originates from their crucial applications that are globally spread across networks and data centers. To keep an overview, these applications and the underlying infrastructure produce telemetry data. In the scope of application observability, telemetry data includes metrics, logs, events and traces. The first examples that come to many people's mind are machine-related metrics such as CPU usage, bandwidth capacity and current storage. But telemetry is more than just machine data. It can consist of user behavior data as well, such as the number of times a user clicked on a button within a mobile app. For many Product Owners, Software and DevOps engineers, putting together such telemetry data in a meaningful way has become a challenge. The difficulty comes not only from the vast quantity of data and its distributed location, but also increased costs and wrong formatting. The complexity of data management becomes exponential as newly-added tools multiply.

So far, the answer to these struggles has been *telemetry systems*. A telemetry system consists of three stages. First, data is emitted by servers, databases or IoT devices. Then, the data is collected, processed and stored in arbitrary backend, e.g. observability platforms Datadog [1] or New Relic [2]. Depending on the data source, the telemetry can be queries by analytics platforms as well. However, organizations fail to see the benefits of such telemetry systems, often due to sub-optimal implementation and increasing costs. They are faced with similar problems that telemetry systems initially promised to solve. As reported in [1], a crypto company was billed $65 million by the observability platform Datadog in Q1 2022, which illustrates a good example of why telemetry systems should be designed carefully.

To simplify the overall telemetry landscape, developers, platform engineers and software directors seek for refactoring methods. In this context, *refactoring* refers to restructuring or optimizing the systems to improve efficiency, scalability, maintainability and resource utilization. It can reach from adapting source code to migrating from one visualization tool to another.

[1] Datadog, `https://www.datadoghq.com/`
[2] New Relic, `https://newrelic.com/`

1.1 Motivation

During my job as DevOps engineer at Leica Geosystems, maintaining telemetry systems has made up only a small part of my day-to-day work. Though, after talking to one of the managers about our complex telemetry environment (most of which I was not even aware of) and his need for simplification, I felt motivated to conduct research into refactoring telemetry systems.

Leica provides measurement solutions through total stations and laser scanners for industries such as surveying, transportation and heavy construction. As their product palette broadened over the years, their backend solutions and naturally their telemetry landscape expanded.

Since telemetry in a broader scope has always received a very low priority, various tools and technologies for collection and visualization of metrics, logs and traces were only introduced by team leaders and service teams in the course of crucial projects. Consequently, each telemetry system is isolated and dedicated for certain departments and products.

The lack of a common telemetry strategy is the root of many reasons why somebody would want to refactor telemetry systems. The major reason for this thesis revolves around data silos. As multiple dashboards of different visualization tools present data, it becomes especially hard to correlate usage behavior with sales numbers.

Quickly accessing and combining statistics of various product types is also not easily possible. Simultaneously, software directors perceive it as a worrying trend that the operative and maintenance expenses are rising at a pace much faster than the value you get out of the tool chains. For a few organizations it may even occur that the value is diminishing despite the costs going up.

Due to the modern approach of service compartmentalization, most of the metrics-driven application performance monitoring tools cause the costs to skyrocket. Every additional pod, host or service raises the bill. This results in a cost-visibility trade-off. Companies decide to either cut visibility to regulate budget or continue over-paying in telemetry tools. Obviously, limiting the collected data or the number of employees accessing the data is contra-productive because it restricts comprehending user behavior and system health.

Yet another reason for refactoring data pipelines can include inefficient debugging practices for existing software products (e.g. due to their low degree of observability). Additionally, telemetry systems may require adaptation in order to meet security or compliance needs such as GDPR.

1.2 Objective

The problems outlined in the previous section lead to the ambition to figure out refactoring strategies for telemetry systems. On the one hand, this thesis aims to scheme a migration concept that Leica Geosystems can apply to consolidate their existing telemetry systems. The concept should help to not only reduce operative as well as maintenance costs in the long-term, but also take care of a good price-value ratio. On the other hand, the experience from the migration concept together with the literature review should then result in a list of refactoring strategies on a technical and organizational level. In summary, this thesis strives to answer the two following research questions:

- **RQ1** : From a technical and organizational point of view, what kind of improvement strategies exist for a telemetry system?
- **RQ2** : What approach should be chosen in order to consolidate the existing telemetry systems of the company Leica Geosystems?

1.3 Approach

In order to define refactoring strategies for telemetry systems, an extensive literature research and review was performed. Afterwards, the thesis tried to map the findings to the telemetry environment of Leica Geosystems. This happened in form of a study, which involves an analysis of different telemetry systems in respect to usage and challenges. This helped to better understand the stakeholder's needs for refactoring. Then, various technical and organizational optimizations led to a consolidated telemetry environment.

1.4 Structure of the Thesis

The thesis is structured in eight chapters. It starts with the introduction which briefly describes the problem and objectives. The problem statement chapter defines telemetry systems and their consumers, helping to better understand the problem statement. The subsequent chapter reviews the relevant literature sources about observability and telemetry systems, and lists what was used from the state of the art. Then, the next chapter gives an overview of the research methodology. The solution chapter begins by differentiating the terms observability, monitoring and analytics, followed by the deep analysis of the study's telemetry systems. Afterwards, this chapter considers various trade-offs when opting for a consolidated telemetry architecture, accompanied by important best practices and a concise

proof of concept implementation. The results are then evaluated and discussed based on the experiences of the PoC and feedback of the participants. Finally, the major activities are summarized, and an outlook is given.

2 Problem Statement

This chapter explains the problem this master thesis aims to solve.

2.1 Definition of Telemetry Systems

To better understand the issue at hand, the term *telemetry system* needs to be defined. From an high-level perspective, a telemetry systems (also called observability system) is made up of three parts: emitting, shipping and presenting telemetry.

Users can establish multiple forms of telemetry system topologies. The most direct form is when application and machine data is directly sent to an all-in-one observability solution. This tool then processes, stores and visualizes the data. Another topology can consist of sequentially connected tools as presented in Figure 2.1. Two very different telemetry systems, but with the same flow of data-handling show the similarities between these telemetry types [2].

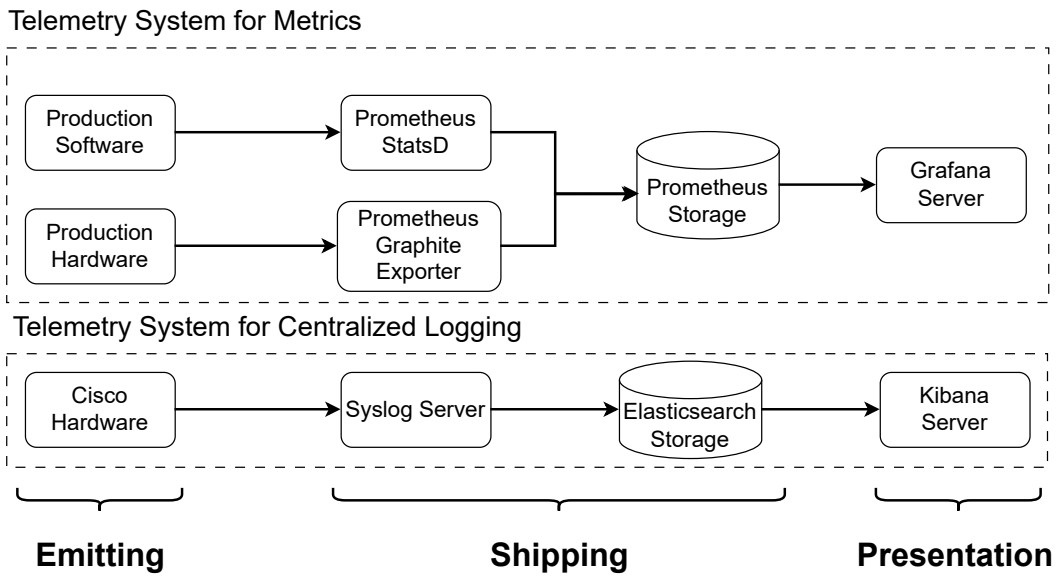

Figure 2.1: Example of two telemetry systems for two telemetry types. Black directional lines indicate telemetry flow. Adapted from [2].

There is also the phrase *telemetry pipelines*, which represents something different than telemetry systems. A telemetry system chains together multiple tools for collecting, storing and visualization, while a telemetry pipeline is often a tool that can route data from sources to destinations. Collecting telemetry data requires either an agent by the vendor of their underlying observability tool or popular open source forwarders such as Fluent Bit[1] or OpenTelemetry Collector [2]. After collection, the data can be transformed by a number of processors, which involve masking and formatting fields, filtering out redundant entries and so on. Simultaneously, it is common to enrich the arriving data, meaning giving additional context like geolocation or timestamps. The target locations can range from different storage types to fully-fledged observability solutions like New Relic [3]. The capabilities of telemetry pipelines are visualized in Figure 2.2.

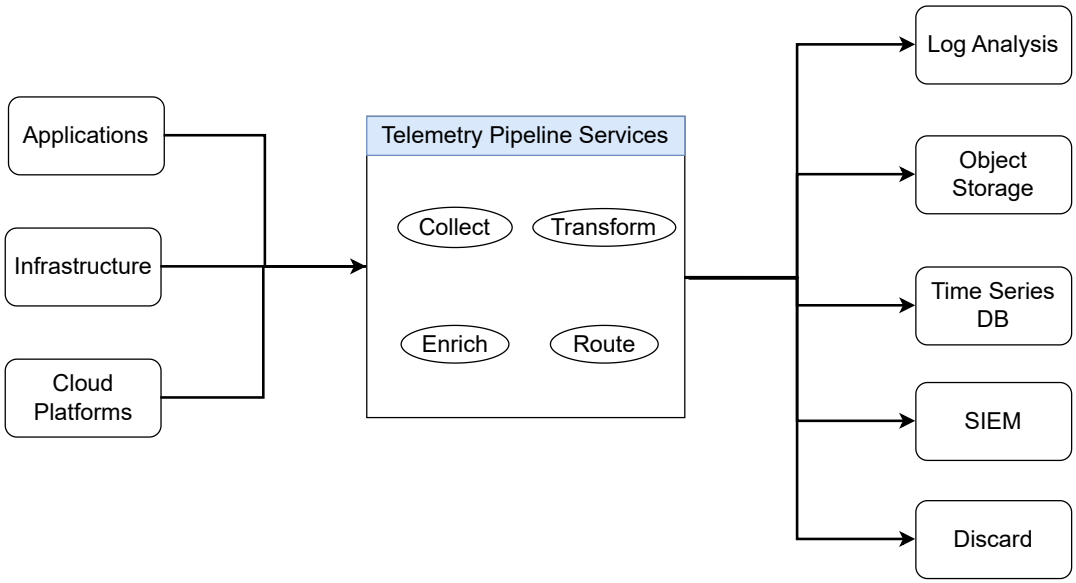

Figure 2.2: Components of telemetry pipelines. Adapted from [3].

For example, the collected data in Figure 2.2 may contain following information:

- application data such as events or traces spanning across microservices and databases
- low-level infrastructure-related metrics such as memory and CPU utilization of virtual machines, bandwith, device uptime
- logs omitted by cloud platforms such as Kubernetes

[1] Fluent Bit, `https://fluentbit.io/`
[2] OpenTelemetry Collector, `https://opentelemetry.io/docs/collector/`

Consequently, a telemetry pipeline is not a synonym of a telemetry system, but rather part of it. This thesis uses the word *pipeline* as synonym for telemetry systems (since it can be seen as a chain of tools). Whenever *telemetry pipelines* become a subject of interest over the course of the chapters, they will be phrased as such.

2.2 Consumers of Telemetry

As telemetry is essential for decision-making about production systems, refactoring telemetry systems first starts by understanding the way consumers require telemetry to make those informed decisions. A consumer often falls into one of the categories which are described in the following.

One group of the consumers are IT Operations, DevOps and SRE. IT Operations' job revolves around maintaining the health of systems in which production code runs - meaning cloud or hardware. Monitoring the systems towards performance supports them in having good uptime. The DevOps philosophy arose to overcome the barriers between IT and software development by continuously ensuring code quality, rapid deployment and regular feedback loops. Naturally, this required them to use observability tooling to drill down and troubleshoot their services. While IT Operations usually consumes metrics and logs, DevOps teams may look at all types of telemetry data, depending on their infrastructure. The latter may be true for Site Reliability Engineers too. They mostly care about error budgets and service level objectives [2].

Another target group are software engineers. They take charge in developing the software that is executed in the production environment. They are interested in how their code is performing, and expect extensive troubleshooting capabilities of their tool-chain. They especially find distributed tracing helpful to debug errors step-by-step [2].

The third consumer group are people working in business intelligence. Understanding how they make use of telemetry can be a bit tricky as they primarily focus on the business's telemetry, and not the one of the technical organization [2]. Their telemetry encompasses data like marketing conversion rates, account upgrade or downgrade rates. However, especially product managers may depend on more technical data to make strategic decisions about their products. For example, by knowing when customers use certain features more than others, product managers can better prioritize and allocate resources during roadmap planning.

2.3 Problem Statement

Over the years, the telemetry systems of Leica Geosystems' R&D department has grown into a complex mix of different chain of tools for different products and services.

The main reason for this situation was the missing observability strategy. Now, software directors fail to keep the operative and maintenance costs of these systems low, while some software engineers lack necessary tools to debug effectively. Product engineers and product managers want to ask questions about user behavior. Thus, the goal of the study is to **develop a observability strategy to consolidate the existing telemetry systems so that these target groups can take action based on the insights retrieved from telemetry data**.

3 State of the Art

Through an extensive literature review this chapter summaries the relevant existing work that relates to this master thesis.

3.1 Observability Engineering

When searching for the terms *observability* or *telemetry*, one of the top hits is the book "Observability Engineering" [4] as it uncovers the various facets of observability. Starting from the historical background of the term, it provides a clear definition of observability by separating it from monitoring and shedding light on misconceptions. The book suggests practical ways to boost production with observability, which involves not only technical solutions for the building blocks of telemetry systems but also cultural aspects. Although the authors do not really mention the phrase "refactoring", some of their technical and social considerations can be transferred to refactoring strategies. Apart from that one of the latest literature is the book "Learning OpenTelemetry" [5], which advises on how to effectively integrate OpenTelemetry in production.

There is also a report by Miles [6] describing a real-world example where a regulated bank was drowning in the flood of telemetry data. His team had adapted to known DevOps principles, such as independently scaling microservices, but their dashboards and telemetry data could not keep up with the constantly changing environment. When a critical incident occurred, they could not respond properly, which deteriorated their reputation. It turned out that the root cause was not the sheer amount of volume, but the data silos. Putting everything together on demand needed a lot of time and insight into particular areas. The key learning was to set up proper telemetry pipelines, whereby the proposed methods are very similar as in [4], but the author focuses only on high-level solutions. For example, one method is to carefully route and filter the data. The motivation of the report is similar to the one of this thesis, but the initial situation is quite different.

While describing practices on how to build telemetry systems, the report [6] mainly refers to the solutions provided by Mezmo [1]. Mezmo is a tool that supports users to optimize, streamline and better understand their telemetry data. It

[1] Mezmo, `https://www.mezmo.com/`

achieves this by offering services such as data profiler to profile new data sources, recipes to select pre-configured pipeline scenarios. It focuses on configuring telemetry pipelines by connecting sources and destinations via various processors that can aggregate or transform the data. As part of its documentation [7], Mezmo also suggests techniques on how to optimize an existing telemetry data pipeline.

Lastly, the paper [8] performed a Systematic Mapping Study about observability in Cloud-native apps. Studies between 2018 and 2022 were analyzed to discover the motivations to adopt observability as well as implementation strategies.

3.2 Cost-Visibility Trade-Off

In the course of the last ten to fifteen years, the standard approach to monitor code in production has been to track application metrics and make them visible to engineering teams. To fully diagnose issues in system behavior, teams have to dig through the vast amounts of telemetry data. But the sheer volume is not the only reason that spiral the costs. Many of the leading observability tools such as Datadog and New Relic attract customers with immediate data collection and seamless integrations. To maintain this set of features, they often request the customers to pay per host, service, pod, which makes cost estimations for consumers difficult. They also use custom format for metrics, and additionally index the same data in multiple formats. Consequently, the consumers either over-invest into observability tooling or decrease visibility to stay within budget [9].

To escape this cost-visibility trade-off, observability vendors reassure with different strategies. For example, Honeycomb [9] advertises having a true price-value ratio thanks to their rich data model, powerful processing engine and fair pricing. On the other hand, Logz.io [10] concluded that only approximately two thirds of the overall data indeed provides helpful information. Eliminating unused telemetry data can be easily achieved with their Data Optimization Hub. To minimize the computation footprint, they also recommend data transformation techniques and a cold storage as a cost-effective storage without sacrificing search performance. Moreover, circonus [11] advises against multiple open-source solutions as with growing volume of data, they can become resource-heavy and lead to the difficulty of correlating insights. Circonus votes for unified platforms built upon open standards together with the concept of dynamic observability, which enables automatic collection of more telemetry data during high traffic, and less data when not needed [12]. In addition, Gartner [13] published a market research about the most popular observability tools. Ultimately, the sources [4], [5], [13] were used for technical refactoring tactics. The source [4] was considered when applying social an organizational strategies.

4 Method

The research employs a study approach, focusing on four distinct telemetry systems within the company Leica Geosystems and is depicted in Figure 4.1.

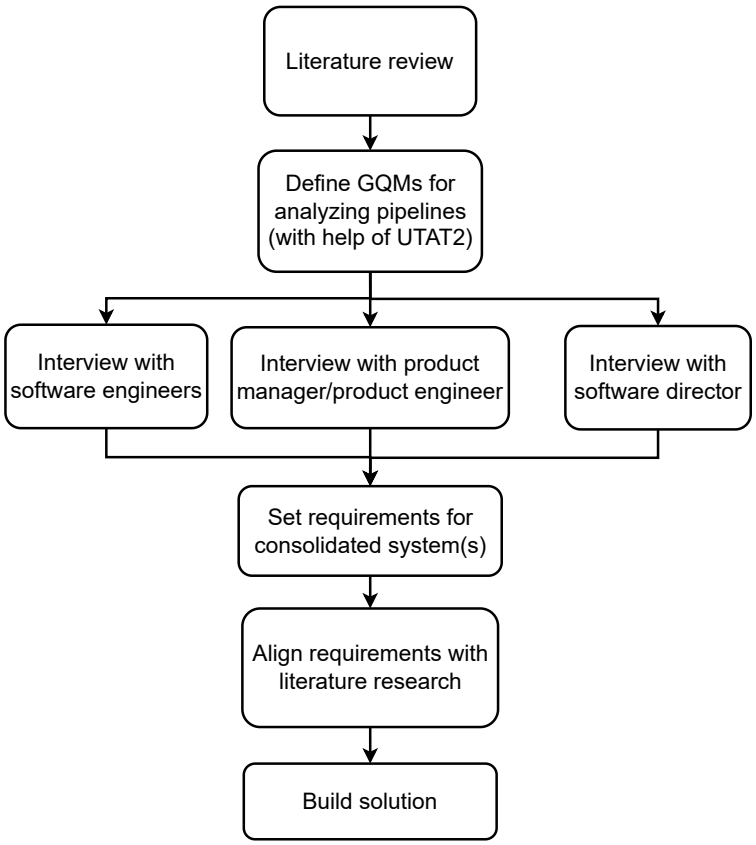

Figure 4.1: Steps of the research methodology.

For each system, I conducted interviews with two software engineers and one product manager or product engineer. A software director, who oversees the software development process of Leica's products, participated in the study as well.

To analyze and evaluate the existing telemetry systems, a set of dimensions (criteria) had to be defined. This was mainly done using the Goal-Question-Metric framework.

The semi-structured interviews were conducted around GQM questions to ensure consistency and depth in the data collected. The GQMs helped to assess the usage and challenges of the telemetry systems. To set the focus of the GQMs, the UTAT2 framework provided an understanding of the motivations and reasons behind the need for refactoring.

After gathering the necessary data from the interviews, I identified common patterns and themes. Then, the literature research concluded meaningful technical and organizational strategies that can generally help to build or simplify telemetry systems (RQ1). These tactics were then mapped to the existing telemetry systems of the study (RQ2).

5 Solution

The solution chapter is composed of four parts. The first part (chapter 5.1) gives an overview of the terms observability, monitoring and data analytics. The second part (chapter 5.2 - chapter 5.6) dives deep into the usage and challenges of this study's four different telemetry systems. The next part (chapter 5.7 - chapter 5.9) brings up key best practices that led to the necessary observability strategy. The section also includes a proof of concept where some use cases were implemented for a desktop app in .NET. The last part (chapter 5.10) summarizes the results by

5.1 Understanding the Versatility of Telemetry Systems

One of the immediate findings of the study is the confusion and lack of awareness for terminologies in conjunction with telemetry. Out of all the interview participants, only one engineer could properly tell the difference between observability and monitoring. Therefore, this section describes the terms monitoring, observability, and business analytics, which should shed light on how different consumers have different needs in regard to telemetry.

5.1.1 Modern State of Observability

In 1960, engineer Rudolf Kálmán first introduced the term *observability* for describing control systems in mathematics. Observability in the scope of control theory "is defined as a measure of how well internal states of a system can be inferred from knowledge of its external outputs"[4]. Since then the term was used in various communities to define different subjects. To map the original concept of observability to software, some further thoughts related to the software engineering realm had to be made. Engineers try to achieve the following points [4]:

- Grasp the functioning of your application's core mechanisms
- Grasp any potential system conditions your application might encounter, even unforeseen ones
- Attain insight into the inner workings and system conditions merely through examining external tools

- Comprehend the internal state without deploying additional custom code to address it, as it implies pre-existing knowledge for explanation

In other words, observability "for software systems is a measure of how well you can understand and explain any state your system can get into, no matter how novel or bizarre. You must be able to comparatively debug that bizarre or novel state across all dimensions of system state data, and combinations of dimensions, in an ad hoc iterative investigation, without being required to define or predict those debugging needs in advance. If you can understand any bizarre or novel state without needing to ship new code, you have observability" [4].

At first glance, this immediately raises the question of what systems does observability even observe. Usually, these are distributed systems consisting of multiple components that run on different machines or environments, and coordinate and communicate over the network through messages. This does not only include applications operating as microservices within a Kubernetes cluster in the cloud. Observability also benefits traditional architectures such as web apps querying a monolithic backend [5]. Engineers always appreciate it when they can trace their source code to reproduce user's problems. However, Majors [4] emphasizes that the capabilities provided by observability tooling becomes a must-have when facing modern distributed architectures. The sub-chapter 5.1.2 explains why this is the case by contrasting the debugging practices between monitoring and observability.

Viewing the systems from bird's eye perspective, they are composed of resources and transactions. On the one hand, resources are physical elements such as containers, CPU, and RAM. On the other hand, they include logical elements such as API endpoints, load balancers, databases. A resource is every kind of component constructing a system. Furthermore, users initiate transactions (requests) that make use of the resources to fulfill the desired functionalities. The only way to comprehend these systems is by looking at the telemetry they emit. A lot of people are confused about the term telemetry. It has become an ambiguous word. The general consensus is that we distinguish between performance and user telemetry. User telemetry data helps to understand users' interactions with the overall system. This may include session durations, number of button clicks or distribution of app versions. Performance telemetry, however, tells how well the multiple system components are behaving and performing. Examples include CPU utilization, speed of database queries, average response time of web servers, network throughput. In simpler words, performance telemetry measures the time to load a webpage, while user telemetry states how many visits that webpage got in the last month [5]. If needed, the performance telemetry can be further split into network telemetry (e.g. open network ports) and application infrastructure telemetry (e.g. error rates of microservice) as defined by Splunk [14].

Regardless of the telemetry type (metric, log or trace), each type undergoes the

same workflow. Firstly, the telemetry type is instrumented as code so data can be emitted. Then the data is transferred over the network via a transmission system to an analysis solution, where the data is observed. This outlines a crucial distinction: The system producing the data is seperate from the system that performs analysis on the data. “Telemetry is the data itself. Analysis is what you do with the data” [5]. Consequently, observability is the equivalent of telemetry with analysis [5].

5.1.2 Differences between Monitoring and Observability

While observability and monitoring are closely related, they have major differences. This sub-chapter compares the two concepts across different dimensions such as focus, data, flexibility and debugging approaches.

The purpose of application monitoring is primarily to track the health and performance of systems over a period of time. It focuses on noticing known issues and expected system behavior by utilizing predefined metrics displayed on static dashboards together with alerts to highlight anomalies. People can actively follow one metric for changes. For instance, an engineer may have decided to receive E-Mails whenever the CPU usage rises above 90% for more than five minutes. Observability, on the other hand, aims to deliver a wide understanding of a system's state based on its external outcomes. The focus lies in answering questions about problems you could not have predicted in advance. Furthermore, monitoring's scope is narrower as it concentrates on only specific components or metrics of a system, and relies on structured data in regular intervals, such as disk I/O or request rates. In contrast, observability provides a holistic view of the entire system's operation by integrating granular, context-rich form of data such as events and traces [15].

For many years, monitoring was effective because systems were straightforward enough for engineers to determine precisely where to look for issues and how these issues might manifest. A large percentage of the problems were variations of previously known failures. This made metrics-based monitoring approaches an appropriate choice as they offer alerts when certain known thresholds are reached. However, when system grew in complexity and especially became more distributed, the limitations of troubleshooting with monitoring came to light [4].

In the early stages of monitoring, one static dashboard provided only core statistics to capture the state of one service (partly due to the reason that there were not many metrics to care about). Modern architectures emit numerous metrics, making it impossible to depict them on a single dashboard. Yet people still try to fit them together by aggregating and averaging them - for instance, memory usage across a cluster may be 80%. These values do not tell you anymore which systems or processes are causing such conditions. Filters and drill-down options were attached to the dashboard tools by vendors to react to this problem. Nevertheless, the effectiveness of using dashboards as a means to troubleshoot is constrained

by the need to predefine the conditions you are investigating. You must specify in advance the dimensions along which you want to break down values so that the dashboard tool can create the necessary indexes for your analysis. You might realize that it is helpful to group disk usage by instance type, but you cannot since you did not add the required labels beforehand [4].

Another crucial difference in debugging is institutional knowledge. Institutional knowledge refers to unwritten information known by some but not widely known within an organization. In monitoring-based approaches, teams often rely on seniority for knowledge: the longest-serving engineer is usually the top debugger and the last resort for troubleshooting. In contrast, teams using observability adopt a different approach. With observability tools, the most effective debugger is often the most inquisitive engineer. These engineers use observability to explore their systems, asking questions and using the answers to guide further inquiries. Instead of relying on deep familiarity with one specific system, observability values broad investigative skills that can be applied to various systems[4].

The impact of this shift is most noticeable when you have to identify problems deeply hidden within intricate systems. A monitoring approach can hide the true cause of issues due to confirmation bias. Problems are diagnosed based on how their behavior resembles previously known issues, which can result in addressing symptoms rather than the root cause. Engineers speculate on the potential problem, quickly seek confirmation, and alleviate the symptom without thoroughly understanding the underlying reason. Even more problematic can be the fact that teams may end up with two issues than just one when they fix only the symptom, but not the cause. In comparison, instead of relying on expert anticipation, observability enables engineers to see each investigation as if it were brand new. When problems are detected, even if the conditions appear similar to past issues, an engineer can methodically follow the trail of system data. By analyzing the information step by step, they can consistently determine the correct solution. This systematic approach allows any engineer to diagnose issues without needing extensive familiarity with the system's complexities to intuitively guess a solution. Additionally, the objectivity of this method ensures that engineers can pinpoint the exact cause of the problem they are addressing, rather than merely treating symptoms reminiscent of past issues [4].

To round up this chapter, the each of the following two tables (Table 5.1 & Table 5.2) demonstrates an example where observability helped to solve the respective issue, while monitoring could not.

5.1.3 Definition of Data Analytics

In this study, the term data analytics also popped up few times, particularly during conversations with product managers. Data analytics involves the systematic

Table 5.1: Example of observing performance of a mobile application using monitoring and observability. Adapted from [16].

Situation	Mobile users complain that a specific feature in the app is slow, but server-side metrics do not imply any problems.
Monitoring	Server-side monitoring shows all services are operational with low latency, suggesting no issues. However, this does not align with user reports.
Observability	Tracing the requests made by the mobile app, including interactions with the backend API, reveals that the latency is introduced by multiple sequential API calls made by the app, which are not optimized for mobile networks. The backend services are fast, but the cumulative latency of these calls degrades the user experience.

examination of raw data to draw meaningful conclusions that help businesses to improve performance. Numerous techniques and processes within data analytics have been automated, employing algorithms and mechanical methods to process raw data for human interpretation. Manufacturing companies, for example, frequently track metrics such as runtime, downtime, and machine work queues. They analyze this data to optimize workload planning, aiming to ensure that machines function closer to their highest capacity. Any type of information can be fed into a data analytics tool. In the context of software development, one popular use case involves delivering personalized recommendations through understanding the preferences and behavior of users. E-Commerce and social media sites are good examples for that [17]. The Table 5.3 contrasts the different types of data analytics.

Table 5.3: Types of data analytics. Adapted from [18].

descriptive	**predictive**	**prescriptive**
describes what occurred over a period of time, e.g.: were sales higher than last year?	forecasts what is likely going to happen in the near future, e.g.: predicting company's future cash flow	suggests a course of action, e.g.: TikTok's "For You" tab

In conclusion, the terms observability, monitoring and data analytics are related, but serve distinct purposes. Monitoring is best suited for reporting the overall health and performance of systems. It helps to understand system-level interests by tracking key metrics and setting up alerts on known failures modes. However,

Table 5.2: Example of observing efficiency of microservice communication. Adapted from [16].

Situation	A microservices-based application performs well under normal conditions, but under high load, some services start to lag slightly, affecting performance
Monitoring	shows that all microservices are up and running, with no significant failures, and resource utilization is within expected ranges
Observability	Detailed analysis of inter-service communications reveals that the lag is due to inefficient synchronous communication patterns among certain services. For instance, Service A waits for a response from Service B for a non-critical operation before proceeding, which becomes a bottleneck under high load.

observability is a property of a system - similar to testability and usability. It tells you the degree to which you can observe your application's internal workings by its outputs. This concept significantly gained importance when organizations started to switch from monolithic to microservice-architectures. The resulting growth in complexity revealed the limitations of monitoring tools when it comes to debugging. Observability tools are ideal for handling application-level concerns as they combine metrics, logs and traces into a context-rich set of information crucial for detecting the "unkown unkowns". Observability is not a substitute of monitoring, the two concepts rather complement each other. Furthermore, the telemetry that is emitted through observability lays the fundamental stones to perform data analytics methods. Data analytics tries to gain actionable insights out of telemetry data. By comparison, many observability and monitoring tools notoriously have only limited data analytics capabilities.

5.2 GQM as Analysis Approach

The first goal of this thesis (RQ1) is to conceive a migration strategy that consolidates telemetry systems of Leica Geosystems' R&D department. As the initial step, we perform a deep analysis of each system across selected criteria, which should help to better understand the requirements of software engineers and product managers. The telemetry environment of this study consists of four distinct systems. While a few pipelines use different telemetry vendors plus some custom parser logic, others pursue a more open-source system. Consequently, this poses a challenge to define criteria that are applicable to all kinds of pipelines.

The Goal-Question-Metric method (GQM) turns out to be an appropriate framework to overcome that hurdle. In short, it is a structured top-down approach for defining and interpreting (software) metrics in a systematic and goal-oriented manner, building a hierarchy with three levels [19]:

1. **Conceptual level (goal):** This level outlines the objectives and motivations behind a study. It specifies the "object of study," which encompasses the particular products, processes, and resources being examined.

2. **Operational level (question):** At this stage, specific questions are formulated to determine not only important elements of the object of study, but also which attributes of these elements are used to evaluate or achieve the related goal. These attributes, known as the focus of the study, help identify which particular aspect of the object is being scrutinized to assess goal attainment. Questions serve as measurable links between the object of study and the focus. For instance, if examining a plane and focusing on its environmental impact, a relevant question might be: "What are the carbon dioxide emissions of the plane?"

3. **Quantitative level (metric):** This level establishes the collection of software measurements required to objectively (quantitatively) respond to the questions posed at the operational level.

Using this framework, the idea is now to derive reasonable metrics from the overall goal of refactoring telemetry systems. These metrics were then part of the analysis of the different pipelines. Since the GQMs view the object of study with a specific focus under a certain target group, multiple GQMs were created.

To set the focus elements of the GQMs for software engineers, it was important to understand why they desired to change their tooling. To do so, the Unified Theory of Acceptance and Use of Technology (UTAUT2) is used [20]. This model provides a robust and holistic understanding of user intentions to use a particular technology. Out of the seven key factors introduced in that model, two factors were successfully considered as the basis for two GQMs: performance expectancy and effort expectancy. Performance expectancy refers to the degree to which a person thinks that using a particular technology will provide benefits in their job performance. For example, a salesperson might adopt a customer relationship management system if he or she believes it will support them manage client details better and close more sales. Hence, the GQM was drawn in Figure 5.1.

The goal is to improve a given telemetry system in order to improve developers' efficiency in their daily work. Mainly, the developers' frequency of interaction with the data is evaluated. The below Table 5.4 explains why the questions were asked.

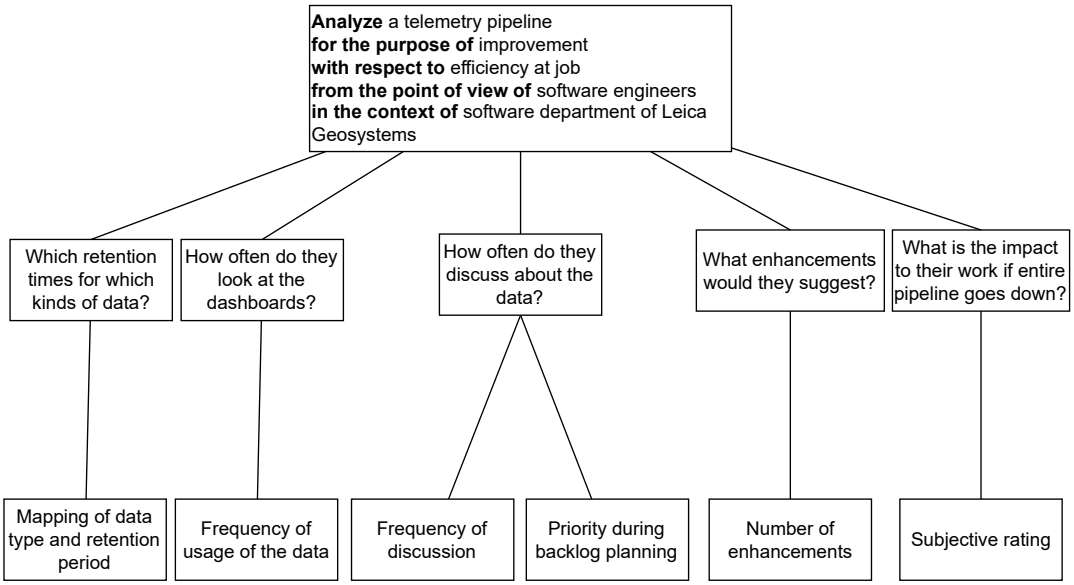

Figure 5.1: GQM 1. Focus is to improve work efficiency from the perspective of software engineers.

Table 5.4: Explains the relevance for the questions of GQM 1.

Nr.	**Questions**	**Reason for relevance**
1.1	Which retention times for which kinds of data?	Understand how long data is stored and how long it is actually needed
1.2	How often do they look at the dashboards?	Get a sense of frequency of usage and reliance on telemetry
1.3	How often do they discuss the data?	Frequent discussions might indicate that data is crucial
1.4	What enhancements would they suggest?	Obtain developer's personal point of view of where improvements should happen
1.5	What is the impact on their work if the entire pipeline goes down?	Highlights the criticality of pipeline availability; significant impact suggests improving robustness of pipeline

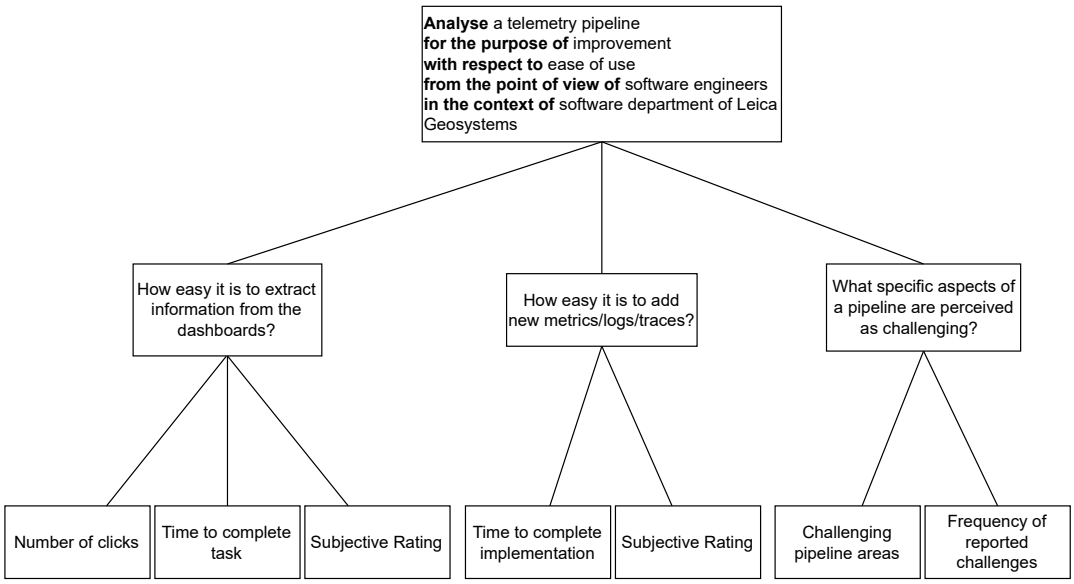

Figure 5.2: GQM 2. Focus is to analyze how easy it use the given telemetry system.

The basis for the second GQM (Figure 5.2) is the effort expectancy. In the UTAUT model, this factor tells how effortless or difficult it is to use of a particular technology.

Table 5.5: Explains the relevance for the questions of GQM 2.

Nr.	**Questions**	**Reason for relevance**
2.1	How easy it is to extract information from the dashboards?	Difficult extraction might indicate inefficient user interfaces
2.2	How easy it is to add new metrics/logs/traces?	If this is cumbersome, it can hinder pipeline's ability to evolve with changing requirements and thus impact its long-term efficiency
2.3	What specific aspects of a pipeline are perceived as challenging?	Find out targeted areas for improvement

While the two GQMs for software engineers deal with a single system, the ambition of software directors is to agree upon strategies to consolidate telemetry systems, primarily with the purpose of reducing operative and maintenance costs (see Figure 5.3 Table 5.6).

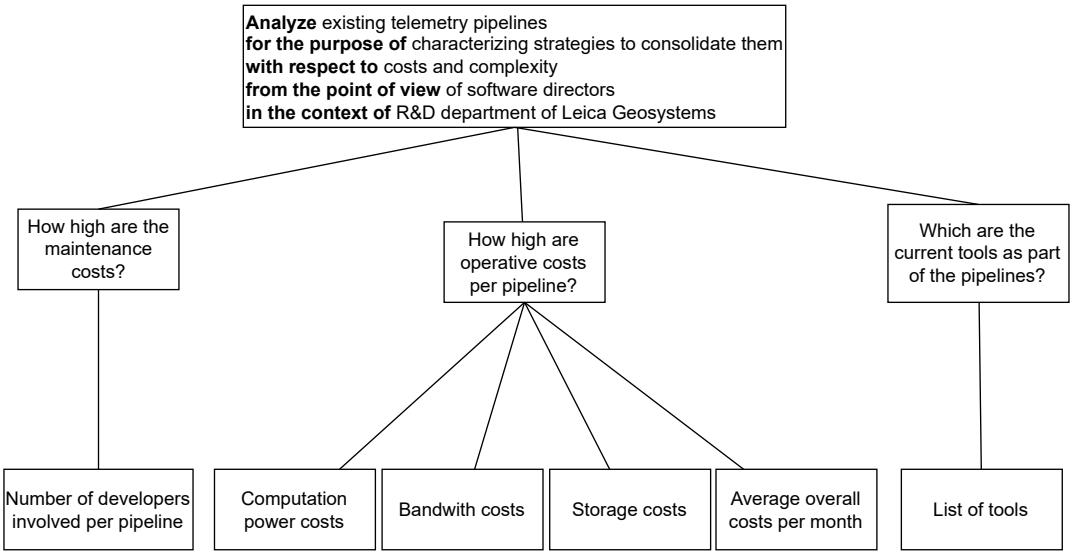

Figure 5.3: GQM 3. Focus of software directors is to reduce costs of their telemetry landscape.

Table 5.6: Explains the relevance for the questions of GQM 3.

Nr.	**Questions**	**Reason for relevance**
3.1	How high are the maintenance costs?	Understand how many developers are involved in keeping the pipeline alive
3.2	How high are operative costs per pipeline?	Needed to determine how much is actually “too much”
3.3	Which are the current tools as part of the pipelines?	Get a holistic view of telemetry landscape

In summary, the metrics of the GQMs can be grouped into multiple categories that form the main criteria (Table 5.7) across which the telemetry systems are analyzed in the next four chapters. Each analysis starts with a description and background details of the respective pipeline. Then it moves on with applying the analysis criteria, highlighting the benefits and challenges.

Table 5.7: The chosen analysis criteria for telemetry systems

Criteria	**Explanation**
Costs	measures operative (financial) and maintenance costs; unit for financial costs is not currency, but average "points (p)" per month; points do not include maintenance costs; nr. of developers is metric for maintenance costs.
Stability	evaluates the robustness of pipeline through e.g. number of outages and reported challenges
Complexity	degree of how well a pipeline can be maintained and understood; ranking: low - medium - high
Data interaction	measures frequency of data usage, especially retention times; although people might not exactly tell how long data is needed for debugging, idea is to get a rough estimation.

5.3 Analysis of Application Insights Pipelines

5.3.1 Description

As seen in Figure 5.4, the first pipeline is a chain of commercial tools and custom implementations. Two concrete products part of this pipeline were selected for the analysis. One of the main telemetry producers is the product *Infinity*, an office software helping to easily manage and process data points originating from different surveying devices and sites. The product idea is to have all the data at one place for the purpose of better project collaboration. Infinity has been delivered as a desktop application to the customers for more than ten years now. It has a monolithic architecture built with .NET. Developers instrumented the application with the Application Insights SDK. Application Insights [1] is a feature by Azure Monitor that provides users a extensive application performance management capabilities via failure views, alerts, live metrics etc. The emitted data arrive at the Application Insights' telemetry ingestion endpoint, which then writes metrics into

[1] Application Insights, `https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview`

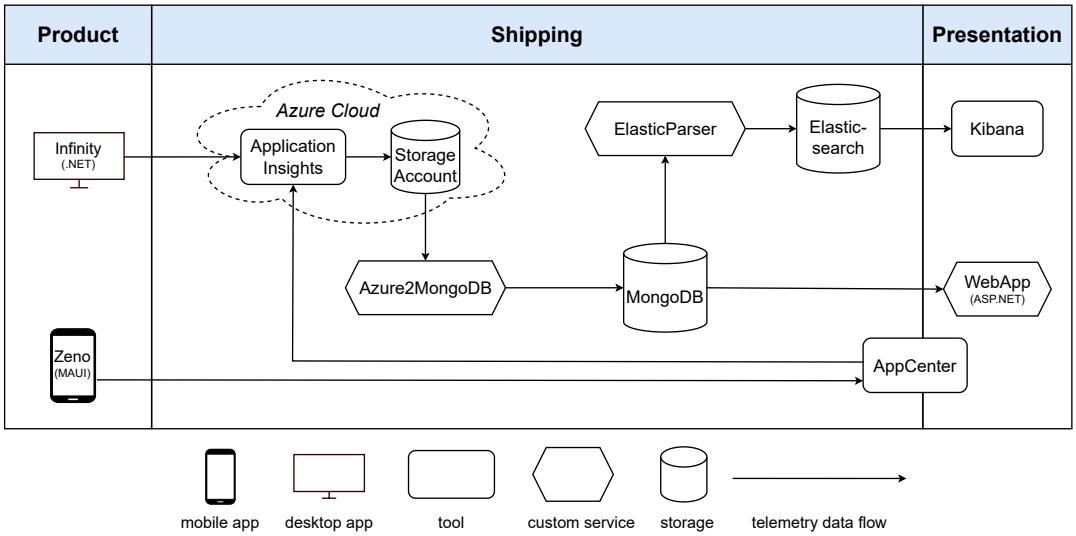

Figure 5.4: Pipeline using Application Insights and AppCenter for instrumentation. Words within brackets indicate used technology.

a internal metric store as well as logs and traces into a log analytics store. Examples for metrics include *project creation time, data points count and vertices count*. The developers also collect various types of exceptions through logging. For instance, *ArgumentNullException* can occur when generating a report. The monolith also interacts with other services - either product interfaces or third-party services. Traces show the activity flow in some of these communications. To save costs on Azure cloud storage, it was decided to not only set the retention period to 30 days, but also export the data to self-hosted databases (to keep data for long-term). To do so, developers configured Application Insights to automatically stream the data to a blob storage (storage account). Every day a custom downloader service (Azure2MongoDB) requests this data, formatted in JSON, deserializes it, does a little processing and eventually persist it in a MongoDB instance. In this instance, there is one database per product, each of them having almost the same collection structure for storing the telemetry types. The JSON extract in Listing 5.1 shows how a *ArgumentNullException* as a log entry looks like.

```
{
    "_id": "67167168176781",
    "time": "2024-02-11T10:41:59.5492629Z",
    "resourceId": "/SUBSCRIPTIONS/8dc22b67-ca12-4da1-
      -80d0-23556bf2b298/...",
    "ResourceGUID": "edvaeed32-ad23-4dc0-9777-7-
```

```
        k8114224d58",
    "Type": "AppExceptions",
    "AppRoleInstance": "Role C",
    "AppVersion": "1.2.3",
    "ClientCity": "Santiago",
    "ClientCountryOrRegion": "Chile",
    "ClientIP": "0.0.0.0",
    "ClientModel": "HP Laptop 15",
    "ClientOS": "Microsoft Windows 11 Home",
    "ClientStateOrProvince": "Region Metropolitana",
    "ClientType": "PC",
    "ProblemId": "System.ArgumentNullException at \
      ...",
    "ExceptionType": "System.ArgumentNullException",
    "Method": "System operation method ...",
    "processed": true
}
```

Listing 5.1: Extract of a sample MongoDB document representing an exception as log.

Many of the fields (e.g. time, resourceId) were added as part of the automatic instrumentation of Application Insights SDK. Since the developers consider this type of document as raw data, they implemented additional parsers to convert and post-process it. There is one parser that condenses the entries by converting them to a different schema through a couple of mapping functions. In addition, an *ElasticParser* mainly sends logs and few metrics to a Elasticsearch instance. With Kibana, developers query these logs as well as create charts about crash rates per version. They created few charts about license usage for product management as well. Specifically for product management (not only the ones from Infinity), however, a web application in .NET was created. The web app maps the MongoDB entries to its domain model and additionally uses XML configuration files to build up the respective diagrams. By mostly glancing at pie and bar charts, product managers get to know how many customers use which app version in which country.

Apart from that Figure 5.4 depicts how telemetry data produced by the mobile app “Zeno” is sent to Microsoft’s AppCenter. It is a DevOps cloud solution to build, test and distribute applications. It offers analytics and diagnostics services to view usage and status of an app. Every Friday a developer uses this tool to review the current status of the app by looking at potential crash rates and API calls. They especially consider AppCenter to capture immediate failures after releases. The

mobile app is implemented in .NET MAUI.

5.3.2 Observations and Challenges

Furthermore, two developers of Infinity were interviewed about the state and challenges of this telemetry system, primarily applying the questions posed in the GQMs (Figure 5.1 & Figure 5.2). The meeting revealed a couple of observations and issues.

First of all, the MongoDB still has been poorly maintained. It significantly grows in size since no cleanup is in place at all. As soon as disk space of the server hits its limit, it is just extended by more gigabytes. In the long-term, this will result in a serious scalability problem. The fact that some telemetry is saved twice (due to the conversion) exacerbates the issue. Although the data is backed up by IT operations team, developers are not sure if the recovery will be useful since they don't know what exactly is backed up. Besides, when trying to view the MongoDB databases for this study, it was difficult to find a compatible GUI because the instance is running on a version that is four stable version older than the latest one.

The second problematic lies in the sporadic data losses across the different steps of the pipeline. The reason is that the custom implementations are error-prone as they heavily rely on the format of the data stored in the Azure storage account. As a consequence, this recently led to a adaption of the downloader and the parsers as Azure changed data layout and folder structure in the storage account. Even in the past there were cases where Microsoft dropped support for some telemetry fields. Few misconversion of the parsers caused data not being propagated to Kibana dashboards as well. Another worrying trend is that such problems are recognized only when developers need certain data fields and not beforehand. Due to these incidents, developers and product managers have a more reserved attitude towards the displayed telemetry data as they do no fully trust it. Moreover, the following points were further observed about the general usage of the pipeline:

- adding new telemetry data to the pipeline takes one or two days.
- building charts in Kibana is very easy.
- product managers stopped looking at dashbboards in both frontends as they currently do not see any valuable information.
- developers usually dig through the logs rather than looking at dashboards.
- developers do not use the "Discover"-tab, a user friendly GUI to search and filter data. Rather they write cURL-like requests in the "Dev Tools"-tab, whereby building nested queries is cumbersome.

- developers do not use the pipeline as primary source of truth in case of incident response. They always request the log files from the customers first. If they want to find out whether a particular error is occurring for multiple users, they consider querying in Kibana.
- retaining exceptions for one year is enough.
- no huge impact to the daily work of developers if the pipeline were to go down for a couple of days. This is because they rarely need to generate reports.
- despite the low usage of the pipeline, tickets about fixing failure of pipeline get high priority, justified by the argument: "in case we need to generate some reports in future".

For the mobile app developers and product managers, the problems and usage are very similar to the ones from Infinity. They additionally have the problem that Microsoft decided to retire AppCenter[2] on March 31, 2025. Therefore, they are waiting for a telemetry tooling strategy from software directors.

For the purpose of completeness, the Table 5.8 lists the concrete values of the GQM measurements for Application Insights pipelines. For AppCenter, GQM was not applied due to its retirment.

Table 5.8: GQM measurement values for analysis of Application Insights pipelines. Ratings are always from 1 - 10; higher numbers mean "good" or "easy". Related questions are in chapter 5.2.

Measure	**Value**	**Question**
Current retention M.E.L.T	forever	Q 1.1
Needed retention M.E.L.T	6 months	Q 1.1
Frequency charts usage	none	Q 1.2
Frequency data discussion	very rarely	Q 1.3
Priority in planning telemetry	low	Q 1.3
Nr. of enhancements	4	Q 1.4
Impact of pipeline outage	low	Q 1.5
Data exploration with charts rating	7	Q 2.1
Data exploration with querying rating	5	Q 2.1

Continues on next page.

[2]AppCenter, `https://learn.microsoft.com/en-us/appcenter/retirement`

Table 5.8, continued from previous page.

Measure	Value	Question
Ease of telemetry addition rating	8	Q 2.2
Time to add telemetry	2 days	Q 2.2
Challenging areas	MongoDB; parsers	Q 2.3
Frequency reported challenges	4	Q 2.3
Nr. of maintenance developers	2	Q 3.1
Average operative costs per month	300p	Q 3.2
Nr. of tools in pipeline	8	Q 3.3
Complexity ranking	high	Q 3.3

5.4 Analysis of Sentry Pipelines

5.4.1 Description

The next pipeline, which is part of Leica's complex telemetry environment, is Sentry.io[3]. The Figure 5.5 illustrates that the web application *GeoCloud-Drive*, written in Angular, is currently the only product connected to Sentry.

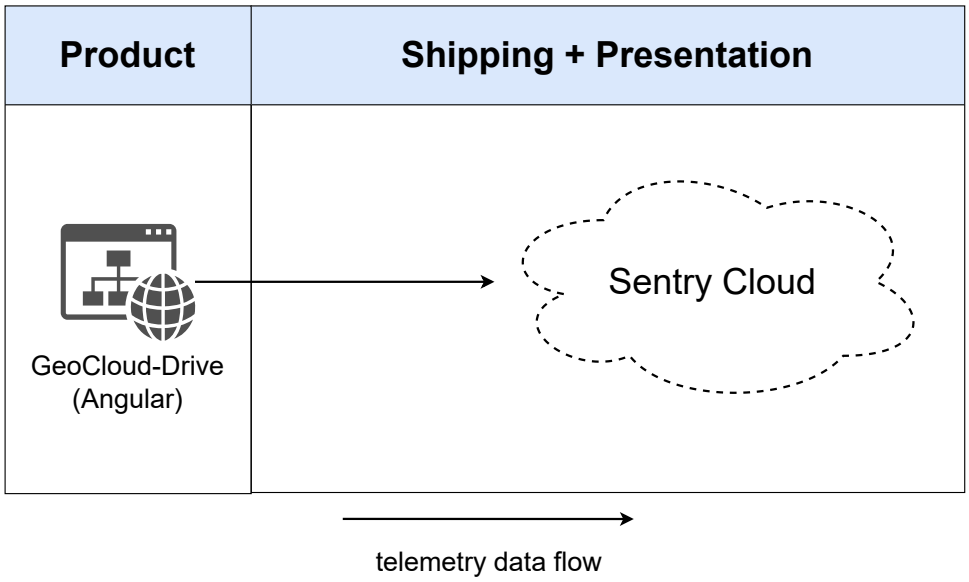

Figure 5.5: Only one web application reports telemetry data to Sentry.

[3]Sentry.io, `https://sentry.io/`

The web app provides a space where users can create projects and exchange files with each other. Sentry.io labels itself as a “developer-first error tracking and performance monitoring platform” allowing users to develop software faster. Since it is a full-blown solution, developers just had to instrument GeoCloud-Drive with the Sentry’s Angular SDK, and Sentry Cloud took care of collection, storage and analysis parts. In comparison to other platforms, Sentry excels in providing extensive support for frontend applications by tracking regressions, long running queries, performance bottlenecks and page speed.

As an example in this pipeline, developers created widgets to visualize Core Web Vitals as well as the frequently or rarely used GraphQL operations. Simultaneously, they look at the User Misery score, which is a performance metric weighted by user impact to evaluate the relative significance of application performance issues. User Misery focuses on the number of unique users experiencing frustration due to response times exceeding four times the satisfactory threshold (in milliseconds). This metric is sentry-specific and identifies transactions that most significantly affect users. The calculation takes into account both the volume and proportion of users experiencing slow transactions; for instance, 6 out of 6 miserable users users will result in a lower User Misery score than 750 of 1000 miserable users. The Table 5.9 contains an example on the User Misery scores per transaction.

Table 5.9: Top five transactions with the highest User Misery score. Each transactions lasted four times longer than the threshold of 300ms. Scores range from 0 - 1.

Transaction	**User Misery (300)**
/items/	0.612
/items/:itemId/upload	0.585
/items/:itemId/get	0.408
/folders/:folderId/contributors/remove	0.321
/folders/:folderId/contributors/list	0.297

5.4.2 Observations and Challenges

When interviewing a software engineer and product engineer about Sentry, its usage greatly varies depending on the state of their product GeoCloud-Drive. According to the software engineer, they conduct Sentry in two scenarios: bug reports or alerts. In case of a bug, developers try to determine whether it only happened to one particular customer or is affecting a broader set of customers. To do so, they scroll through the *issues* tab of Sentry, whereby an issue can be seen as a single bug or problem. Sentry automatically arranges similar events into an issue to make

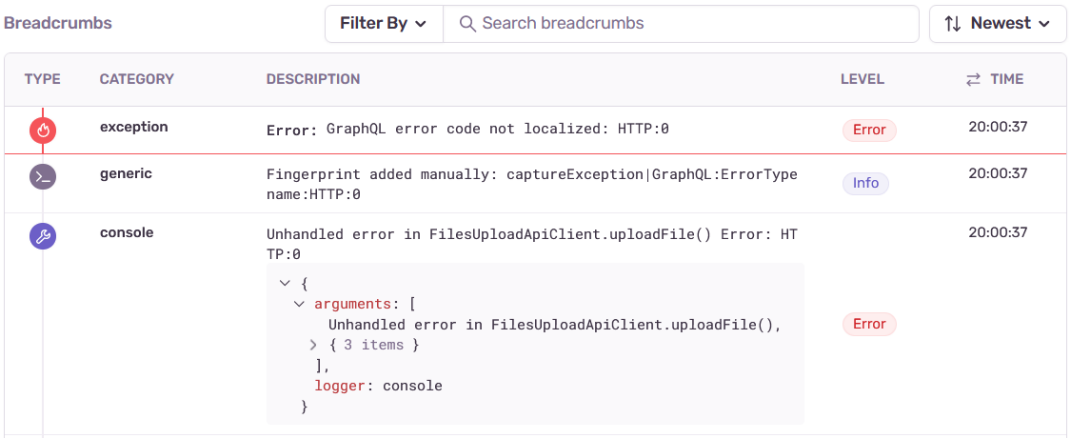

Figure 5.6: An extract of breadcrumbs leading up to an error. The categories vary between HTTP request, user action, navigation, transaction, error.

them more manageable. The issues can be sorted by number of events, number of affected users, trends and latest seen. An issue is categorized as either error or performance issue. When viewing its details, you see not only highlighted context items and tags (browser, user-id etc.) but also the time when the problem was first seen together with the full stack trace. The interviewed developer emphasized that the breadcrumbs of an error are valuable because they draw a timeline and history listing all the events that induced the error (see Figure 5.6). Moreover, the software engineers profit from custom alerts. For instance, Sentry notifies them via Microsoft Teams when new types of errors were captured. They receive warning or critical alerts, as soon as 75% of the transactions take more than certain thresholds within an hour.

In regard to current usage by product engineers, they only briefly check the report that Sentry automatically generates once per week. It contains summary statistics such as number of errors and transactions per project (application). The point in time they discussed more intensively about collecting telemetry was the initial release of GeoCloud-Drive. Major reason was that the web app showed slow performance back then. The engineers wanted to pinpoint whether the issues lied on the frontend or in the response times of the API calls to the backend. Thus, they set up key performance indicators, and used them as proof during discussions between the two teams. Despite some friction in the beginning, both teams agreed on the areas for improvement and introduced service level objectives. However, until this day there is still another issue. The backend consists of many microservices that provide crucial interfaces for a variety of products. Naturally, the teams established different telemetry collection and visualization strategies over time. As a consequence, the frontend developers do not have access to the

entire traces spanning from user interaction in the frontend to database calls in the backend. This limits their visibility on a global level. Apart from that the analysis of the pipeline revealed the following points:

- company bought Sentry because it was strongly recommended by two frontend developers who already gathered decent experience with it during their previous work.
- retention of traces can be short (approx. 2-3 weeks) as mostly recent data is needed for troubleshooting.
- if Sentry were to go down or not properly work, it would have a high impact on developer's work, and get medium priority in backlog planning.
- product engineers do not have enough data to recognize patterns in user behavior.
- product engineers have the feeling that Sentry has limited capabilities regarding advanced statistics (probabilistic distributions etc.).

For the purpose of completeness, the Table 5.10 lists the concrete values of the GQM measurements for Sentry.

Table 5.10: GQM measurement values for analysis of Sentry.io. Ratings are always from 1 - 10; higher numbers mean "good" or "easy". Related questions are in chapter 5.2.

Measure	**Value**	**Question**
Current retention M.E.L.T	3 months	Q 1.1
Needed retention M.E.L.T	1 month	Q 1.1
Frequency charts usage	only during investigation	Q 1.2
Frequency data discussion	rarely	Q 1.3
Priority in planning telemetry	medium	Q 1.3
Nr. of enhancements	0	Q 1.4
Impact of pipeline outage	7	Q 1.5
Data exploration with charts rating	8	Q 2.1
Data exploration with querying rating	8	Q 2.1
Ease of telemetry addition rating	9	Q 2.2
Time to add telemetry	half day	Q 2.2

Continues on next page.

Table 5.10, continued from previous page.

Measure	Value	Question
Challenging areas	custom charts	Q 2.3
Frequency reported challenges	1	Q 2.3
Nr. of maintenance developers	0	Q 3.1
Average operative costs per month	80p	Q 3.2
Nr. of tools in pipeline	1	Q 3.3
Complexity ranking	low	Q 3.3

5.5 Analysis of Grafana stack Pipelines

5.5.1 Description

Next to Leica's core web services and frontend applications, engineers track telemetry from supplementary services as well (e.g. mail-service, RabbitMQ). The underlying Kubernetes infrastructure is monitored as well. They use Thanos to scale up their Prometheus Cluster for storing time series data, while a self-hosted Grafana Loki centralizes the logs emitted by the pods. Afterwards, engineers can search and filter these logs in Grafana. They apply not only the USE (Utilization, Saturation, Errors) method to monitor hardware resources (CPU, memory), also the RED (Rate, Errors, Duration) method to build meaningful alerts and measure SLAs.

End of last year a small group of developers required a quick telemetry backend as part of their investigations and experiments with OpenTelemetry, which is an open-source framework for generating and exporting all types of telemetry data. Instead of adding Grafana Tempo to existing infrastructure, they used a free tier Grafana Cloud instance. As seen in the Figure 5.7, so far, only one product - a monolithic web service - is part of the OpenTelemtry (OTel) flow. The application transmits telemetry data to a OTel-Collector, which simply redirects it to Grafana Cloud after short buffering.

5.5.2 Observations and Challenges

Despite the fact that the Grafana Cloud workflow remains experimental, some usage patterns and challenges emerged. The software engineers have created minimal charts to show health and performance of their monolith on a very high level. Most notably, they are satisfied with Grafana's functionality to filter and dig deep into logs and traces. According to the interviewed software engineer, they struggle in the set-up and usage of the OTel-Collector because it currently is

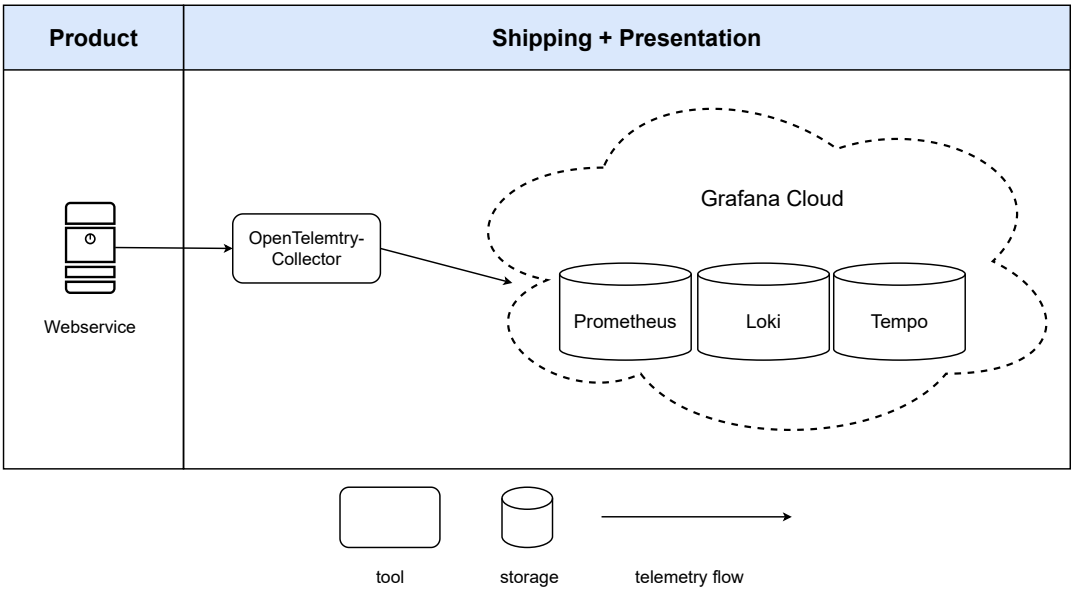

Figure 5.7: One webservice sends metrics, logs and traces via OTel-Collector to a free-tier Grafana instance.

not propagating the data properly. He is neither comfortable nor able to adapt the collector as it is connected to, for him, unknown parts of the deployment process. The root cause is that they mostly don't have time to work on such topics as management excepts them to concentrate on delivering core business features. The same reason applied for the maintenance of the on-prem Grafana environment as well, where the respective people have other kind of problems. For the purpose of completeness, the Table 5.11 lists the concrete values of the GQM measurements for Grafana Cloud pipeline.

Table 5.11: GQM measurement values for analysis of Grafana Cloud (free tier). Ratings are always from 1 - 10; higher numbers mean "good" or "easy". Related questions are in chapter 5.2.

Measure	**Value**	**Question**
Current retention M.E.L.T	14 days	Q 1.1
Needed retention M.E.L.T	3 months	Q 1.1
Frequency charts usage	only during investigation	Q 1.2
Frequency data discussion	often	Q 1.3
Priority in planning telemetry	low	Q 1.3
Nr. of enhancements	1	Q 1.4

Continues on next page.

Table 5.11, continued from previous page.

Measure	Value	Question
Impact of pipeline outage	9	Q 1.5
Data exploration with charts rating	9	Q 2.1
Data exploration with querying rating	8	Q 2.1
Ease of telemetry addition rating	10	Q 2.2
Time to add telemetry	< 10 min	Q 2.2
Challenging areas	OTel-Collector set-up	Q 2.3
Frequency reported challenges	N/A	Q 2.3
Nr. of maintenance developers	1	Q 3.1
Average operative costs per month	none	Q 3.2
Nr. of tools in pipeline	2	Q 3.3
Complexity ranking	low	Q 3.3

5.6 Analysis of IoT Pipelines

5.6.1 Description

These pipelines are not fully comparable to the previous ones since they contain a mix of IoT and monitoring tools. Yet this chapter was included to show the parts relevant to observability.

Many of Leica's surveying devices (e.g. total stations) sends data that to an IoT platform is relevant for obtaining business insights in a later stage. The IoT platform collects and stores such data. More precisely, the platform maintainer label the data as "device telemetry". This type of telemetry often contains the same attributes (e.g. geolocation, timestamp, device version) among the devices, but depending on the product, can have domain-specific fields as well. In case of Captivate, a software product to view and share collect data points of total stations, additional values are gathered such as device type, OS architecture, uptime. Another example includes flight time of the BLK2FLY, an autonomous flying laser scanner. Developers created different contracts with different teams to define the appropriate data formats, so the platform can process the incoming data accordingly.

As illustrated in Figure 5.8, devices send telemetry data first to Azure IoT Hub, with which multiple microservices communicate to receive and further process the data. The microservices are hosted within a Kubernetes cluster. While one service constantly validates the schema and translates the data into an internal

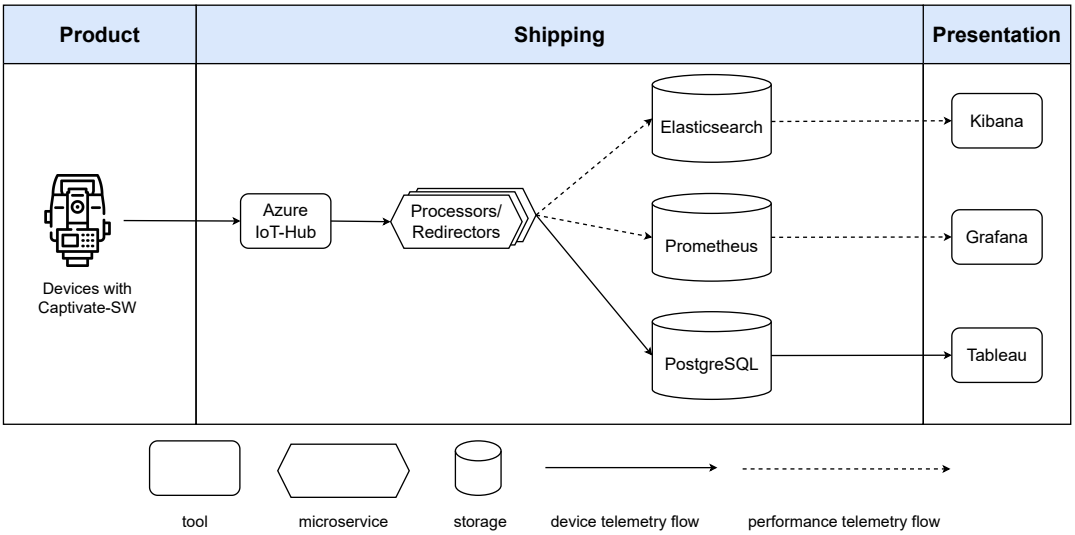

Figure 5.8: Pipeline for collecting and visualizing data emitted by Captivate software installed on Leica's devices. Device telemetry refers to details about the device usage, while performance telemetry refers to the metrics and logs originating from the microservices.

event hub, another one fetches and moves it from the hub to a SQL database. Then, Tableau performs SQL queries to generate charts, such as average number of connected devices per month or version distribution trends. Tableau is well-known for its advanced visualization and analysis capabilities in the realm of business intelligence. It does not deal with application performance monitoring as DataDog, but focuses more on extracting business insights from various data sources (e.g. Microsoft Excel, JSON file, Google Analytics, MySQL). As a means to collect metrics from microservices (e.g. number of ingested events, average time to process an event), developers use Prometheus, and on-prem Grafana to depict them. As a means to store and visualize logs, ELK stack is used.

5.6.2 Observations and Challenges

Software engineers mainly explore logs and metrics on Grafana and Kibana to track the health and performance of microservices. One of the conspicuous results of the analysis were that they value alerting a lot. When asked about the impact in case of pipeline outage, they explained that there would be hardly any data loss thanks to their set of alerts. For instance, assuming that some microservices would be unavailable for a couple of hours, the events that could not be processed in this period still remain on the Azure IoT Hub for several days - ready to be picked up again. One reason to introduce Tableau next to Grafana was that Tableau had a wider

range of source connectors back then. The interviewee pointed out that Tableau offers the powerful feature called “Data Extract”. It creates a compressed subset or snapshot of the original data source in memory to allow high-performance queries.

People involved in this telemetry system see three major challenges. Particularly the software directors speak of Tableau’s high prices per user and month. So far, they have only made use of a small fraction of Tableau’s capabilities. One of the most crucial reports for them is the “world map” displaying the number of connected devices per country. The second challenge the consumers face is on-boarding new products to the pipeline due the custom data format. Changes to this format always need alignment between the teams, which could result in a overhead when integrating a broader set of devices. In addition, they experienced that the queries in Tableau took longer (or were interrupted), whenever data volume increased.

5.7 Requirements of new Telemetry System

The last four chapters unveiled the usage and challenges of this case studies’ telemetry environments. In summary, the employees have a observability and data analytics problem. On the one hand, some software engineers struggle to properly maintain their observability solutions due to complexity and time constraints. On the other hand, product managers do not feel satisfied with their current tooling in terms of tracking user behavior. Based on these analysis results, the next crucial step in developing a observability strategy covers defining high-level needs on the new telemetry system. The functional requirements specify what the system should do:

- standardized method(s) to collect, store and visualize metrics, logs, events, traces emitted by web services, desktop and mobile apps. Software engineers demand this type of data to debug and troubleshoot their applications. This should also simplify the on-boarding process of new apps to the consolidated system.
- data enrichment: automatically adding context to collected data.
- meta-monitoring: observing pipeline itself. Not only basic aspects (health, uptime), but also smarter metrics such as amount of ingested and dropped data.
- frontend observability: tooling should deliver out-of-the box support for gaining end-to-end insights into users of web applications (similar to Sentry in chapter 5.4).

- ability to easily archive logs; they can be queried later on if needed (e.g. for compliance reasons).

Moreover, the we set non-functional goals:

- interoperability: ensuring that a system can work together with other systems or components, preferably regardless of the vendor. Specially tech leads and software directors want flexibility, and to avoid vendor lock-in when introducing new technologies.
- high confidence: people expect a certain degree of stability of the pipeline, so they can consider it as the "single source of truth" when responding to major incidents.
- cost-effectiveness: especially management should have clear visibility into the price-value ratio for the chosen consolidated telemetry system.

5.8 Observability Strategy of this Study

This chapter describes the crucial considerations that led to the refactored telemetry system design for Leica's product and infrastructure landscape (RQ2). Beginning with OpenTelemetry's relevance for building telemetry systems, the section also contrasts the trade-offs between chaining open-source software and buying an all-in-one observability solution. The last sub-chapter portrays the chosen telemetry architecture.

5.8.1 Three Pillars is a troublesome Design

First of all, to grasp the current limitations of telemetry types, it is worth looking at how telemetry types evolved at different points in time. Logging, the earliest form, involves text-based messages that describe the state of a system or service for human understanding. Over time, the methods for storing and searching logs were enhanced with specialized databases for full-text search. While logging provided information about specific events and moments within a system, comprehending the system's changes over time required additional data. Consequently, metrics were introduced, which gave more detailed data on system changes over time. These metrics, representing statistical summaries of system state and resource usage, allowed for alerting based on data beyond mere errors. Finally, with the advent of the modern internet and increasing system complexity, distributed tracing emerged. This form of telemetry tracks entire operations and their interactions across multiple machines, identifying start and end times and operation locations to pinpoint the sources of latency and performance issues.

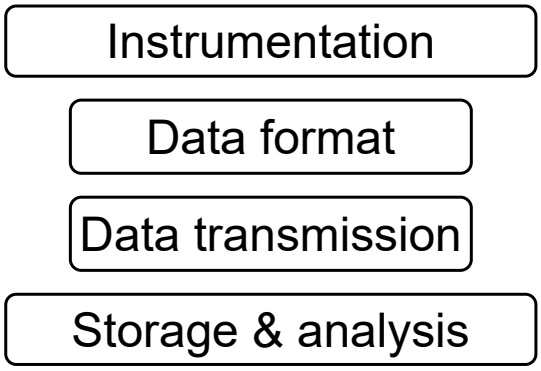

Figure 5.9: One pillar of observability. For example, a tracing system consists of a tracing instrumentation, tracing transmission method and tracing analysis tool. Adapted from [5].

As a result, logs, metrics and traces are labeled today as “three pillars of observability”. However, this is a very bad way to design telemetry systems. Traditionally, each type of observability was developed as an entirely separate siloed system as shown in Figure 5.9.

As new telemetry forms emerged, more vertical implementations (pillars) were built. In this study, patterns of such an architecture were noticed in chapter 5.6, where Prometheus and Grafana collected and visualized metrics, and ELK stack took care of centralized logging. Understanding emergent behavior requires more than examining a single data point or type. Logs or metrics alone rarely reveal the root cause of an issue. Instead, insights are gained by identifying correlations across various data streams. Since the telemetry types remain isolated, the automatic correlation between metrics and transaction logs is prevented. Storing these types in the same data store will not resolve the issue because key identifiers making the correlations reliable and consistent would be missing [5].

The OpenTelemetry framework promises to bring all forms of telemetry together by providing rich context across them. The idea is that standardized, high-quality telemetry should enable unified analysis required for diagnosing problems in live production systems. This marks one of the reasons why adopting to OpenTelemetry is a crucial refactoring strategy in the study.

5.8.2 Benefits of OpenTelemetry

The previous chapter already mentioned one motivation this study proposed to use OpenTelemetry: standardized telemetry allows unified analysis. This sub-chapter lists further reasons on why OpenTelemetry is beneficial.

Applying OpenTelemetry means you own the generated data because it offers

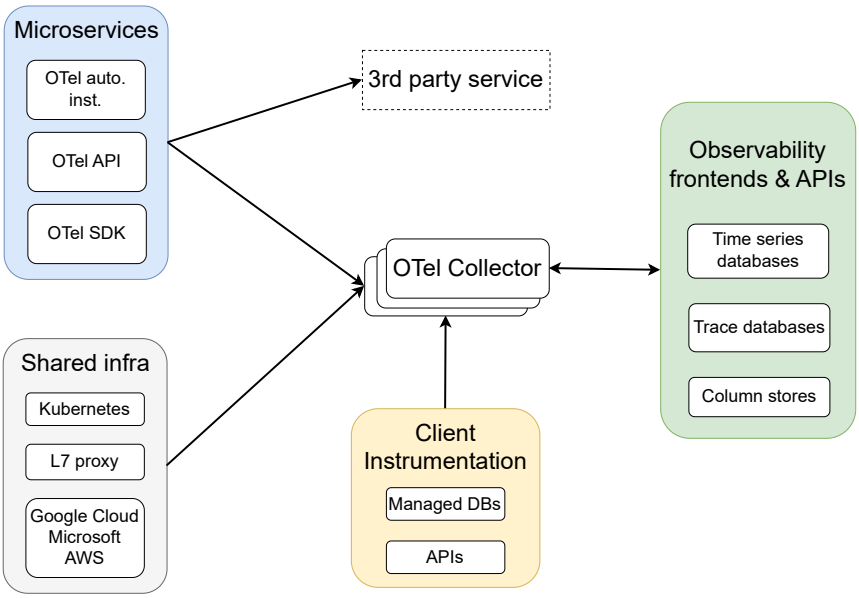

Figure 5.10: Relationship between OTel and observability frontends. Adapted from [5].

a universal way to represent and transmit it to any observability backend. This gives users the possibility to easily swap observability vendors if they need to. This form of "no vendor lock-in" was exactly one requirement of the study. Furthermore, the setup of OpenTelemetry is fairly straight-forward thanks to its automatic instrument, language-specifc SDKs. Developers have to learn only a single set of APIs. Thereby, OpenTelemetry should simplify the adoption of new products to the pipeline too. Table 5.12 summarizes the types of products that will profit from OpenTelemetry instrumentation.

Table 5.12: OpenTelemetry support for .NET and JavaScript, the two major programming language in Leica's product groups. Only support for JavaScript logs is experimental.

Language	**Products**	**Metrics**	**Logs**	**Traces**
.NET	web services, desktop & mobile apps	stable	stable	stable
JavaScript	web applications	stable	development	stable

Another benefit revolves around flexible data handling through the different components (see Figure 5.10). The Collector, in particular, facilitates aggregation, filtering and enrichment of telemetry data before routing it to different backends.

This versatility lets the company tailor their observability systems to meet specific needs, enhancing data flow and minimizing the resource impact on observed applications.

5.8.3 Build vs. Buy

OpenTelemetry does not dictate any storage or analysis tools to route ingest telemetry into. Meaning that the choice of the observability backend depends on the specific desires of the consumers. In essence, this boils down to answering the question whether to buy a complete observability solution or build an own pipeline using open-source tools. This section argues the pros and cons for each approach, and gives a direction for the study at hand.

When you start exploring observability vendor solutions, the overall cost can be quite surprising. This shock often stems from comparing these prices to open source and "free" alternatives that seem feasible to build independently. However, individuals often overlook the value of their own time. If setting up infrastructure and configuring software takes one or two hours, a do-it-yourself approach may seem almost cost-free. As humans, our instincts are overly optimistic about the true costs of maintenance. In reality, we underestimate the time lost due to context switching, the cost of engineers focusing on tasks unrelated to core business functions, and hours drained by that "free" solution. Signs of such pain points were noticed during the analysis of self-hosted Grafana stack in chapter 5.5. To better exhibit the operation costs, it is worth looking at an example. According to Honeycomb [5], a potential customer was thinking about switching to a commercial stack after building an in-house ELK cluster. Honeycomb offered him a price of $80,000/month, totaling nearly $1 million annually. Initially, the customer was shocked by the cost and compared it to the free ELK stack his team was managing. However, he overlooked the expenses of the dedicated hardware needed to run the ELK cluster, which also cost $80,000 per month. Additionally, he did not account for the salaries of three extra engineers required to manage the stack, each costing between $250,000 to $300,000 annually, plus the one-time costs such as recruiter fees (15% to 25% of annual earnings, around $75,000 each). There were also less concrete but still significant costs related to the time spent hiring and training these engineers. Altogether, the "free" solution ended up costing his company over $2 million per year, more than twice the cost of the commercial option. These expenses were less visible to the organization but still substantial. Engineers are expensive, and recruiting them is challenging. Though, the probably biggest benefit of building an own solution is that it strengthens in-house knowledge too. As you need to fulfill the functional requirements of different stakeholders, you can better see what kind of technology will move the business forward. Simultaneously, you will realize what implicit assumptions (or limitations) open source tools have, but

also how they can enable customization.

On the other hand, when going with a vendor solution, the first criteria to look at are financial costs. While it is reasonable for vendors to set prices that cover their costs for delivering a market-ready solution, people [4] say that it is unfair when they obscure the true costs, lack transparency, or include so many pricing dimensions that consumers cannot accurately predict future expenses based on their usage patterns. A common strategy to watch out for involves pricing methods that hide the total cost of ownership (TCO), such as charges per seat, per host, per query. Pay-as-you-go pricing might seem manageable at the beginning, but it can quickly become over-expensive if your usage of a particular tool increases rapidly compared to the revenue it generates. As more and more products become part of the observability system, the harder it gets to predict the costs. Secondly, consumers should consider that buying software means committing to feature set the vendors provide (vendor lock-in). They should be aware of the time and effort required to migrate to a different solution in case they perceive the chosen one as mediocre. To speed up the instrumentation and collection time of telemetry data, many vendors build their own agents libraries to export data in a schema compatible with their observability tool. As already mentioned, OpenTelemetry standard prevents this issue.

Apart from that, introducing an all-in-one observability solution brings along multiple advantages:

- observability can be started within a day thanks to streamlined experience
- maintenance effort is passed on to vendor, plus you have continuous support
- establish a relationship with a partner with years of experience in the observability space, from which one can greatly profit.

Regarding the choice for Leica's case, these pros and cons were considered when trying to fulfill the requirements listed in chapter 5.7. Ultimately, the decision was to go with an hybrid approach: buying a cost-effective vendor solution, accompanied by open-source tools if necessary. The reason behind it was the attitude that the Return on Investment (ROI) would be higher in this way as engineers would put more time for innovation than babysitting the observability infrastructure. Before portraying the concrete consolidated telemetry architecture, the next chapter briefly underlines the differences between observability and BI tools.

5.8.4 Observability vs. BI Tools

Observability enables you to analyze data across multiple dimensions, which helps comprehending the behaviors of individual users, user groups, or the entire system.

According to Majors [4], these insights can be compared, contrasted, or further explored to address a variety of questions that are highly relevant to non-engineering business units within an organization. For instance, one of the interviewed product engineers wants to know which features users like the most. Another question he would like answered: Do they spend more time in the desktop or web version of the application? This type of use cases raise the question whether a business intelligence (BI) tool could perform the same task (instead of an observability tool). The Table 5.13 gives an overview of the key differences.

Table 5.13: Contrasting observability and BI tools

Dimension	**Observability**	**Business Intelligence**
Query exec. time	very fast; subseconds - low seconds	longer due to complex queries
Accuracy	fast results instead of 100% correctness	100% accuracy indisputable
Recency	often fresh data relevant; old data may be interesting only in aggregation	designed to keep data for years
Structure	arbitrarily wide event with lots of context details, which can change over time	consistent schema for repetitive queries

Observability tools need to be fast, supporting quick explorability for real-time analysis, as delays can disrupt the investigative process. In contrast, BI tools are designed for generating detailed reports and complex queries used for long-term strategic decisions, where longer processing times are acceptable. They apply caching or indexing to enhance search performance. While sampling data (only keeping parts of it) is common practice in observability, it is a no-go in BI tools and data warehouses. In terms of recency, observability prioritizes recent data, which is often needed for debugging purposes. BI tools are built to keep data for many, many years. For instance, you might require an exact transaction or billing record from one year ago. However, it is improbable that you would need to determine if the latency between service C and service D was elevated for a specific user request from that same time period. Before storing business data, it needs processing into a predefined schema that simplify queries. In comparison, observability data consists of arbitrary wide events with as many context fields as needed. These details often change as software engineers learn over time which kind of data they require to debug more effectively. Creating a data schema in advance would undermine that goal [4]. In summary, BI tools allow you to identify

the most frequently used features in the last year. Meanwhile, an observability tool offers detailed insights, right down to individual requests. Just as observability and monitoring complement each other, business intelligence and observability work together to ensure you have detailed data that can be aggregated to create comprehensive, macro-level perspectives.

5.8.5 Consolidated Telemetry System

This chapter describes this study's refactored telemetry system. With so many observability solutions on the market, we got quickly overwhelmed when trying to find the right tool (or set of tools). We used the Gartner's [13] detailed study about the best tools for Application Performance Monitoring (APM) in 2023 in order to pick out the most popular solutions, and then contrast these in respect to the requirements. Gartner put each examined APM in the appropriate fields of their Magic Quadrant, which has two axis: ability to execute and completeness of vision. Four tools of the leaders quadrant (high execution, complete vision) were selected: Dynatrace, Datadog, New Relic, Honeycomb. Additionally, Grafana Labs was chosen, which lies in the visionary category (lower execution, complete vision) - almost making it to the leaders quadrant.

Table 5.14: High-level feature comparison of most popular observability vendors. "++" = very good, "+" = good, but with limitations, "-" = not available.

Feature	**Grafana Cloud**	**New Relic**	**Datadog**	**Dynatrace**	**Honeycomb**
Performance monitoring	++	++	++	++	++
Log management	++	++	++	++	++
Infrastructure monitoring	+	++	++	++	+
Real user monitoring	+	++	++	++	+
OTel support	++	++	++	++	++
Free tier	+	+	-	-	+

As seen in the Table 5.14, all of the top APM tools deliver very good results. Dynatrace, New Relic and Datadog are known to provide premium end-to-end

observability covering all kinds of infrastructure and application types (web, mobile etc.). Dynatrace and Datadog have no “forever” free tier option, but a free trial option. New Relic offers a completely free plan until 100GB/month. Honeycomb is the first “OpenTelemetry-native” observability tool with efficient support for high cardinality and high dimensionality data. High cardinality means that a single attribute of an entry (metric or event) can have many unique values, while high dimensionality refers to the high number of attributes. They especially stand out with their “BubbleUp” functionality, which accelerates troubleshooting by automatically correlating logs, metrics and traces to detect hidden patterns. Furthermore, Grafana provides a rich set of customizable visualizations as well as data sources. Honeycomb and Grafana have less out-of-the box solutions for real user monitoring in contrast to the other ones. The next significant factor are obviously the financial costs. A short cost analysis showed that Datadog and Dynatrace had the most complicated pricing structure. For example, Datadog was even billing for storing custom metrics. Honeycomb presents the most straight-forward pricing strategy by just calculating the price per event volume. It stores every telemetry type as an event.

Eventually, the we proposed to go with Grafana Cloud for this study, primarily due to its cost-effectiveness and wide set of data sources accompanied with flexible dashboards. It has a “pay as you go” model, which in Grafana’s case, scales better than other vendors. Although the you cannot determine the exact price per month for such tools, a rough estimation was made by appyling a calculation schema similar to the one from SigNoz [21]. When simulating the usage, Grafana Cloud cost three times less per month than Datadog. Further reasons for adopting to Grafana Cloud include:

- open source ecosystem: very large community contributing to Grafana’s improvement. The involved people in the study value open-source.
- interoperability: supports pulling data from more than 100 data sources (community & enterprise plugins). This philosophy of “single pane of glass” would allow to use Grafana more than just observability. Sample data sources include PostgreSQL, InfluxDB, Amazon Redshift

The next task was about defining the path of OpenTelemetry data to Grafana Cloud. Grafana allows applications to directly send data to their backend via OTLP protocol. This might be fine for testing purposes, but is not recommended in production environments due to scalability and security reasons. Therefore, the designed pipeline applies a mix of agent and gateway Collectors. Grafana Alloy[4] - released on May 2024 - is a distribution of the OTel Collector, which is the best

[4] Grafana Alloy, `https://grafana.com/docs/alloy/latest/`

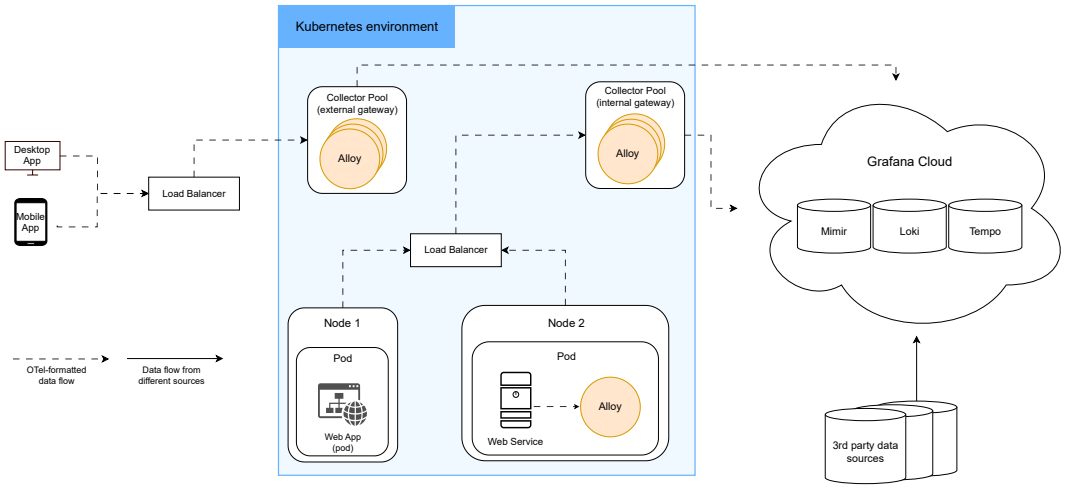

Figure 5.11: Telemetry system concept for this study. Dashed lines indicate flow of OTel data produced by the different applications.

way to transmit telemetry to Grafana Cloud. Figure 5.11 illustrates the topology consisting of a pool of gateway Collectors that receive the data from the load balancer exposed to the public Internet, to which data is pushed by desktop and mobile apps. Besides, a second (private) Collector pool gathers data from web services and web applications hosted on Kubernetes clusters. One big advantages of the gateway pattern is the ability to centralize policy management such as dropping certain labels or only keeping 1 out of 10 traces. Meta-monitoring also becomes easier. Trade-off is higher resource consumption and complexity. If teams wish to introduce their own configuration (e.g. filtering fields or including host-level metrics), they can deploy an Alloy instance as an extra container on the pod (sidecar model).

Furthermore, Grafana's Faro SDK will be used in combination with OpenTelemtry for monitoring applications executed in the browser. This is because OTel supports for browser instrumentation is still experimental and may not provide all necessary features. With just a few lines of code, Faro automatically captures not only unhandled errors, but also the entire user activity (sessions, user journey etc.). The frontend telemetry can be correlated with infrastructure or backend data to achieve full-stack observability.

Infrastructure monitoring is also possible with Grafana. It particularly simplifies monitoring Kubernetes clusters through the provided Helm chart (not depicted in Figure 5.11). By configuring this chart (Figure 5.12, it deploys multiple Alloy instances that collect Pod logs and Kubernetes events. To calculate the Kubernetes

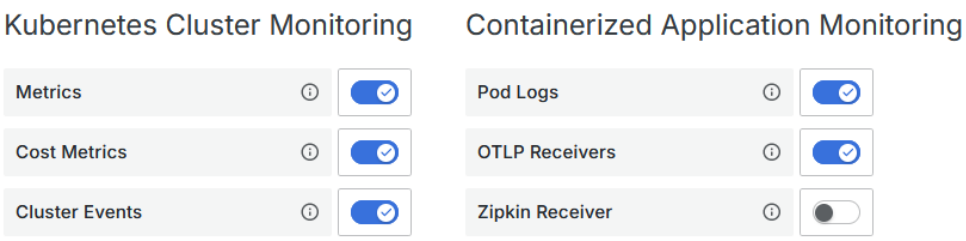

Figure 5.12: Configuration settings to automatically send telemetry data about Kubernetes clusters.

costs, Grafana uses OpenCosts[5].

Apart from that Figure 5.11 shows that Grafana can connect to other data sources as well. Since PostgreSQL connector exists, it raises the questions whether the data in the PostgreDB of the IoT pipeline (chapter 5.6) can be queried with Grafana instead of Tableau. Per definition, Grafana labels itself not as a "BI tool". Although Grafana originated from developers' needs to visualize metrics and logs, it evolved to a powerful platform to visualize any data. This motivated people to apply Grafana also for business-related use case. For instance, engineers at Grafana Labs [22] set up BigQuery as a data warehouse solution, which is fed with events from web and mobile apps as well as data from Salesforce. Then, they leveraged Grafana to generate charts with the BigQuery-plugin. Similarly, we suggested to make Grafana available for business intelligence requirements too. In the specific case of Tableau in chapter 5.6, it turned out that its sole purpose lies in querying the PostgresDB and nothing else. Sales and customer details are visualized by other BI tools. Thus, we replaced Tableau with Grafana. Lastly, we want to mention that the Captivate devices in the IoT pipelines were not added to the final architecture because they emit data that is quite different than the type of telemetry data regarded so far.

5.9 Observing .NET Desktop Apps with OpenTelemetry

The previous chapter gave a deep dive into crucial factors that led choosing Grafana Cloud ecosystem as the observability strategy. In form of a proof of concept, this chapter demonstrates the necessary steps to onboard the desktop version of the GeoCloud-Drive app (chapter 5.4) to the pipeline.

[5] OpenCosts, `https://www.opencost.io/`

5.9.1 Tracing a System Operation

The desktop app allows users to create projects and add members to it. Developers wanted to record the trail of core system operations in order to use them for debugging purposes. As an example, the “Project Creation” operation was instrumented using OpenTelemetry Traces. The app is based on WinUI 3. For confidentiality reasons, the code examples and images are simplified, and contain fake names and values.

OpenTelemetry labels a telemetry type as a signal. The trace signal is basically a collection of correlated spans. A span represents a single unit of work in form of a log. These spans can have several so-called *attributes*. While span attributes only tell details about a specific operation, resource attributes provide (immutable) metadata about the entire service instance. In this case, the desktop app produces the resources name, version and environment (Listing 5.2).

```
// initialize ILoggingBuilder & HostBuilder
// load otlp options from config file

var resource = ResourceBuilder.CreateDefault()
    .AddService(
        serviceName: "CloudDrive",
        serviceVersion: "3.1.1")
    .AddAttributes(
        [new KeyValuePair<string, object>
        ("environment", "development")]);

// ConsoleExporter only for debugging purposes
loggingBuilder
    .AddOpenTelemetry(logging =>
    {
        logging.SetResourceBuilder(resource);
        logging.IncludeFormattedMessage = true;
        logging.IncludeScopes = true;
        logging.AddConsoleExporter();
        logging.AddOtlpExporter(opt =>
        {
            opt.Endpoint = otlpOptions.↘
               LoggingEndpoint;
            opt.Protocol = otlpOptions.Protocol;
        });
    });
```

```
hostBuilder.Services.AddOpenTelemetry()
    .WithTracing(tracing =>
    {
        tracing.SetResourceBuilder(resource)
            // "SourceName = operation-tracer"
            .AddSource(CustomInstrum.SourceName)
            .AddHttpClientInstrumentation()
            .AddConsoleExporter()
            .AddOtlpExporter(opt =>
            {
                opt.Endpoint = otlpOptions.
                    TracingEndpoint;
                opt.Protocol = otlpOptions.Protocol;
            });
    });
```

Listing 5.2: C#-Code for defining resource attributes as well as exporters for tracing and logging. HttpClientInstrumentation does automatic collection of HTTP requests to the backend.

The .NET instrumentation library bases on the *Logger* and *Activity* classes to pull together logs and traces. So, an *Activity Source* called “operation-tracer” was added to start an Activity within the appropriate method for “Create Project”-transaction. Depending on the outcome of the operation, the status of the hierarchically upmost span is set to *ok* or *error* (Listing 5.3).

```
// business logic ...

// Track "Create Project" operation
using (var activity = this.ActivitySource.
   StartActivity("Create Project"))
{
    try
    {
        // validate input params ...
        // write to backend ...
        activity?.SetStatus(ActivityStatusCode.Ok);
    }
    catch (Exception ex)
    {
        activity?.SetStatus(ActivityStatusCode.Error,
            ex.Message);
```

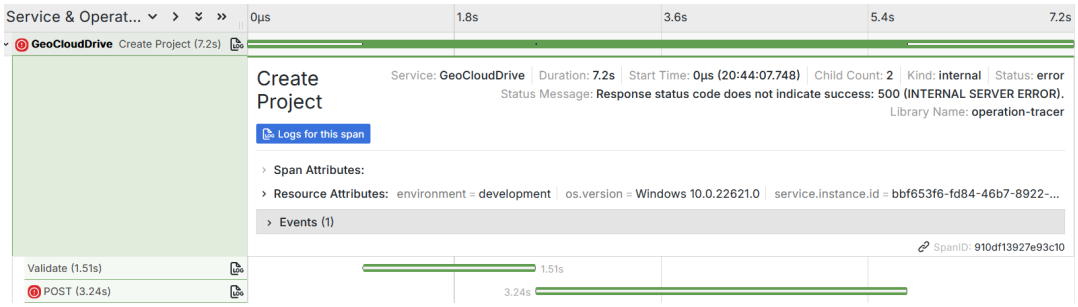

Figure 5.13: Example of a failed trace discovered in Grafana. Attributes and duration of the individual spans are visible.

```
15            activity?.RecordException(ex);
16        }
17    }
```

Listing 5.3: Excerpt of the project creation trace. After validating the input fields, the project details are communicated with the server.

These traces are then exported to a Grafana Alloy instance (on Kubernetes), which redirects them to Grafana Cloud. Figure 5.13 depicts an example of a trace which seems to have failed due the POST request to backend. The span details reveal the respective resource values as well as an event that can be expanded to view the error stack trace. Alternatively, users can click on the blue buttton to navigate to the concrete logs of the particular span.

5.9.2 Metrics Summary

Another request by software engineers revolved around displaying KPIs on a dashboard. The two major use cases include: the average duration of core system operations, and error rate of failed project creations in the last 24h. There are three ways to calculate the latter:

- introduce counters via OTel Metrics API.
- constantly query number of total and failed operations to put them in relation.
- leverage Grafana Alloy's pre-built "spanmetrics"-function, which determines Request, Error and Duration metrics (R.E.D) from spans

The second option could lead to slow loading times once multiple users start to query larger time intervals. The third option basically makes the first option

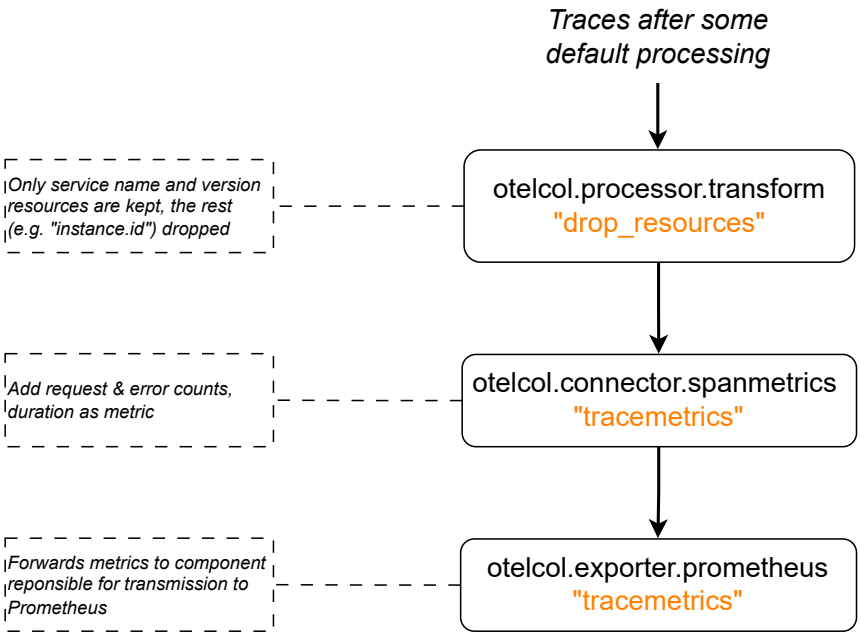

Figure 5.14: Part of Alloy's component pipeline showing the predecessor and successor of the spanmetrics-component.

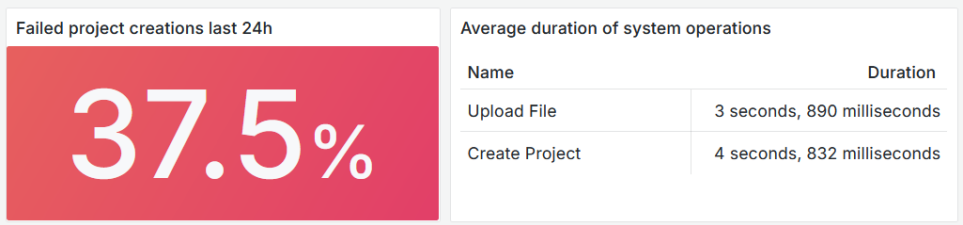

Figure 5.15: Two sample charts that are part of the metrics summary dashboard.

redundant. Hence, the config file of Grafana Alloy wad adapted accordingly. A so-called *component* represents the building block of Alloy. Each component has a single responsibility, and is composed of arguments (settings) and exports (values passed on to other components). Figure 5.14 draws the part of the component-chain where metrics are calculated from spans. Before doing that, unnecessary resource attributes are dropped because, otherwise, they have been retained in the generated metrics. After the metrics are connected through the spanmetrics-component, the last component writes them to Prometheus. Afterwards, the data is presented in form of two charts as seen in Figure 5.15. The table uses the Grafana Tempo data source to obtain the durations based on the globally selected time interval. Prometheus stores metrics in a cumulative way (meaning it always adds

up). The Stat panel uses the following PromQL query (Listing 5.4) to calculate the number of failed “Create Project”-operations within the last 24h. The number of spans with status “error” are divided by the total number of spans in the time interval. Here, it is important to note that an “increase”-call returns an instant vector containing the labels (with their values) and a value for the actual difference. While the first increase-call returns multiple labels, the second one only delivers the span name label in the vector, which hinders valid binary operations between vectors. The “on”-modifier reduces the labels so vectors can match.

```
floor(increase(traces_spanmetrics_calls_total{
   status_code="STATUS_CODE_ERROR", span_name="Create
    Project"}[24h])) / on(span_name)
floor(sum by (span_name) (increase(
   traces_spanmetrics_calls_total{span_name="Create
   Project"}[24h]))) * 100
```

Listing 5.4: PromQL for calculating error rate of “Create Project”-operations within the last 24h.

5.10 Process for Developing an Observability Strategy

The study followed various steps to conclude a observability strategy as described in chapter 5.8. This chapter summarizes the process by going through the key stages, and refactoring tactics that the study considered along the way.

5.10.1 Collection and Instrumentation

While the previous chapters implicitly applied some strategies to control costs, additional (technical) tactics exist to effectively roll out observability. This chapter gives a short overview of:

- filtering and sampling
- transformations
- iterative instrumentation

Filtering means removing any data fields that are not needed. Engineers can drop certain log entries or spans. The PoC in the previous chapter showed how resource attributes were dropped before generating metrics from spans. Engineers can implement filtering both on level of SDK and Collector. However, authors of OpenTelemetry [5] recommend to implement it on the Collector because it not

only cleanly separates concerns between platform engineers and developers, but also allows adapting the pipeline without re-deploying code.

On the other hand, sampling involves selecting a statistically ideal subset of data while discarding the remaining portion. Three sampling types are available [5]:

- Head-based: deciding to sample at start of trace, e.g. keeping 1 of 20 traces. This is not recommended as important info could get lost.
- Tail-based: deciding to sample after trace finished. Enables to only route traces with errors to backends.
- Storage-based: sampling is done by observability tool, e.g. deleting majority of data after certain time period, and a small number of samples stored for historical purposes.

Whether to sample or not is a difficult decision at the beginning due to potential cause side effects when implemented frivolously. That is why no general sample rule was introduced in the study. Nevertheless it will be subject to discussion if the costs become unmanageable.

Transformations form another way to manipulate telemetry data. For example, converting a log into a metric can be an economical way to capture key statistics from web servers or databases. When processing telemetry, two key aspects are crucial. First, the more transformations you perform, the greater the resource consumption. Complex transformations can impact memory and CPU performance, potentially necessitating an increase in your Collector pools. Second, extensive transformations can delay the time it takes for your telemetry to become actionable. It's generally preferable to get things right initially and use attribute transformations to normalize telemetry that cannot be corrected at the source [5].

The last tactic to keep an eye on is iterative instrumentation and collection. As services and conditions differ from organization to organization, no general guidelines to all circumstances can be suggested. Though, what can be said is that avoid monitoring components to which people do not give attention. Often, best practices is to auto-instrument a given service (if possible), and then continue manually instrumenting. It might be difficult to exactly know what kind of data is needed at the start, but clearer over time. That is why instrumentation should happen iteratively.

5.10.2 Process Flow

The below diagram (Figure 5.16 draws up the necessary steps to develop an observability strategy. The first three are about defining the business goals. Establishing clear objectives — such as improving system performance, recognizing patterns

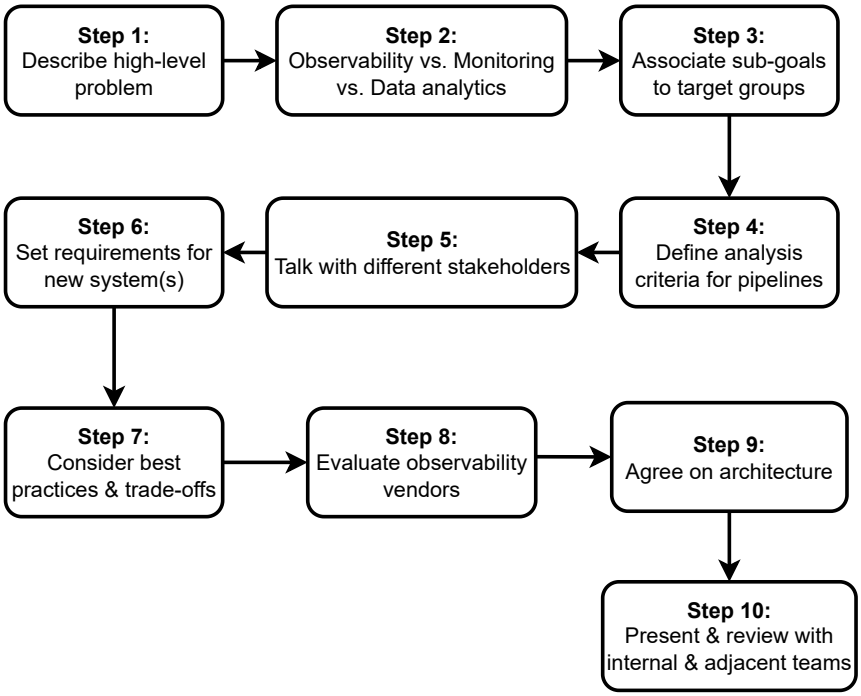

Figure 5.16: Key steps taken by the study to conceive a telemetry system.

in user behavior, or identifying security threats — ensures that the telemetry data gathered is in line with the organization's strategic goals. In this particular case, the problem was first described at a very abstract level: simplifying telemetry environment to optimize operative and maintenance costs. After initial talks with product engineers, software engineers and software directors, the sub-goals were derived and associated with the respective target groups.

The second phase (steps 4-6) covers an evaluation and analysis of the current telemetry systems. Identify any shortcomings or limitations in data collection, analysis, or visualization. In the study, the GQM framework helped to identify analysis criteria for the pipelines. When talking to different stakeholders, asking questions about why and what kind of data is required was important. Going through some selected use cases was pivotal. Based on the recognized challenges and needs, the sixth step was about setting requirements for the "new" telemetry system.

The last phase (steps 7-10) deals with considering different telemetry system topologies. We considered several best practices when building the new system - OpenTelemetry being the most crucial one. OpenTelemetry serves as the primary instrumentation framework to avoid vendor lock-in. In the design phase of a telemetry system, engineers should consider collection and instrumentation strate-

gies as well, such as filters and transformations (chapter 5.10.1). Subsequently, organizations should evaluate observability vendors with respect to the specified requirements. They have to contrast building their own telemetry system with open source tools or opt for a bundled observability solution. Considering the trade-offs of each approach, the study decided to go with an observability tool, which streamlines the troubleshooting practices and reduces maintenance effort. Since nowadays many vendors offer "observability" the one or the other way, the choice may become challenging. It makes sense to narrow down the selection to a handful of tools, and implement short proof of concepts with each of the selected tools. Generally, it is advisable to start "small" by trying out the cost-effective solutions first. In the study, this was done because the requirements were rather on a high-level, and the telemetry usage of certain pipelines was low.

There is rarely one solution that fits it all. Engineers should regularly review their telemetry system and present the positive and negative aspects to not only their peers, but also adjacent teams. By demonstrating how customer support, sales and product teams can profit from observability data, people will appreciate the value of observability. Finally, this should encourage the organization to recognize observability as a core practice.

6 Evaluation and Discussion

This chapter evaluates how well the proposed telemetry system architecture consolidated the different pipelines. Hereby, the evaluation also considers the usefulness and relevance of the more technical refactoring strategies. The thesis assesses the Grafana ecosystem across the following dimensions:

- number of tools
- how often people discuss about the data
- ability to explore telemetry types in an ad-hoc manner
- support for web apps, web servers, mobile and desktop apps

The answers (or values) to these criteria were retrieved through not only the feedback by product managers and developers, but also the experience from the proof of concept. It is important to highlight that, at this point in time, the results and limitations may change over the course of the next weeks or months because the inclusion of all the products were not fully evaluated yet.

6.1 Observations

Despite a massive number of tools left the telemetry landscape, the current solution comes with a few limitations and challenges. First of all, if people would like to add mobile apps based on different technologies, it will not be possible as OpenTelemetry has no mature support for mobile clients yet. As of this writing, it has official support for Android SDK and Swift, and unofficial support for Dart. As the mobile app framework in this case was only .NET Maui, OTel can cover a decent amount of use cases thanks to its stable .NET instrumentation library. However, it requires a lot of manual instrumentation as well as additional effort in building the charts in Grafana. For instance, the OTel Android SDK provide features such as crash reporting, network change detection and offline buffering of telemetry via storage. All of these functionalities are not available when instrumenting a .NET Maui app. The same circumstance applies generally for desktop apps as well.

Besides, two software engineers reported that they experienced a steep learning curve when digging into the query language of Prometheus. Querying metrics was

http_requests_total { method = "GET", status = "200", user_id = "617b1..." }

http_requests_total { method = "GET", status = "400", user_id = "617b1..." }

http_requests_total { method = "POST", status = "500", user_id = "617b1..." }

http_requests_total { method = "DELETE", status = "500", user_id = "617b1..." }

Figure 6.1: Example of how Prometheus stores time series data. Especially adding unique identifiers causes generations of thousands or even millions of time series.

not straight-forward as it required comprehending Prometheus concepts/functions such as instance vector vs. range vector, rate vs. increase function. Obviously, adopting to observability tools will always require learning such query languages. Yet PromQL seems a bit difficult to to become accustomed to.

Another concern is running into a cardinality explosion when storing metrics in Prometheus. For every unique combination of metric name and label values, Prometheus creates a distinct time series. For example, a metric named *http_requests_total* with labels method="GET" and status="200" will be stored separately from the same metric name with labels method="POST" and status="500" (see Figure 6.1). When OpenTelemetry starts to capture a large number of labels, especially ones with high number of unique values, the number of time series grows exponentially, potentially causing crashes or resource starvation on Prometheus. If you were to add a user-id to *http_requests_total* metric with thousands of variations, the cardinality would explode. This implies that using metrics for debugging user-specific data becomes impossible.

The most beneficial outcome is that newly introduced Grafana Cloud covers the majority of the requirements, consolidating the four telemetry systems into one system. Out of the 19 tools - partly custom ones - spread over four pipelines, the new system consists of Grafana Cloud, combination of agent and gateway Grafana Alloy collectors, resulting in a approx. 85% reduction of tools. Meaning that the complexity and maintenance effort significantly went down. As mentioned in the beginning, at this moment, this remains an optimistic result. Certain limitations of Grafana may make it necessary to add the one or other tool or technology. Moreover, if we see that the costs spiral too quickly, we could switch back to a self-hosted Grafana stack to store the data needed for infrastructure monitoring (e.g. Kubernetes). Nevertheless the Table 6.1 lists the names of the tools which were removed from the telemetry landscape.

Table 6.1: Removed tools from this study.

Tool category	Tool
Custom services/DBs	2x Elasticsearch, 2x Kibana, 2x Prometheus, 1x Loki, 2x Grafana, 1x MongoDB, 1x custom frontend, 2x Parsers
Commercial services	1x Application Insights, 1x AppCenter, 1x Sentry, 1x Tableau

Furthermore, developers appreciated the functionality that they could slice and dice their logs and traces in an ad-hoc and intuitive manner. Seamlessly navigating between correlated traces and logs was particularly emphasized. Learning LogQL and TraceQL was a bit challenging at the beginning, but got easier with time. Since OpenTelemetry and Grafana established well as standards in the industry, there is a high chance that a lot of people have experience with them, making the on-boarding process of new employees simpler.

6.2 Threats to Validity

As only a handful of participants were selected when evaluating the existing telemetry landscape, it may seem as a sampling bias. Obviously, the values for a few analysis criteria were based on developer's and product manager's personal experiences and preferences. However, the investigation showed that people using the tools & technologies were not significantly more than the participant group.

Due to the nature of this type of study, not all its findings can be generalized to other settings. This study chose the Grafana ecosystem because it has great price-value in combination with OpenTelemetry. Thanks to its broad set of data sources and rich visualization capabilities, it aids in displaying non-observability data as well. The relevant factors that led to the decision were no vendor lock-in, cost-effectiveness and devotion to open source. Other organizations might opt for a different observability backend due to a variety of reasons and conditions, though. Yet engineers can make use of some of the key results. For example, the process flow described in chapter 5.10.2 serves as an orientation when starting to refactor telemetry systems.

Besides, the study focused on four distinct telemetry systems, but there are some further telemetry systems - managed by other departments/teams - that should be included in future discussions. For example, the web app, which is connected to Sentry (chapter 5.4), makes calls to a backend, which consists of a huge set of microservices maintained by teams part of Hexagon (Leica's parent company). To

achieve full end-to-end observability, it is crucial to align observability strategies with them as well.

6.3 Final thoughts

The impact of the potential limitations as presented in the chapter 6 will become clearer when people start migrating to OpenTelemetry and using Grafana Cloud. If the real-user monitoring for mobile and desktop apps (in .NET) turns out challenging with OTel, Leica could switch to New Relic. A mixed set-up with Grafana for web frontend and backends, and Sentry for mobile and desktop apps could also serve as a good alternative. In addition, the thesis touched upon the fact that observability data can help non-engineering teams, such as product engineers interested in tracking feature usage. The concrete implementation will vary depending on the use case. With the current architecture, if you were to capture "user behavior" via OTel events, they would land in Grafana Loki as logs. Considering that Grafana deletes logs after the a few weeks (or months), doing analytics on very old logs will not be possible (as already hinted in chapter 5.8.4). Maybe engineers should store event data, which they know will be needed for a long time, in a data warehouse or data lake. For instance, Grafana would even provide a data source plugin for BigQuery, Snowflake and ClickHouse - three popular data warehouse/lake platforms.

Personally, what surprised me the most was the fact that software engineers do not get enough time to work on telemetry-related tasks because management expects them to deliver features. I think some managers are not aware of the value that collecting details about user behavior will give them. It helps them to review their products, better prioritize features and assign resources. Simultaneously, a good observability solution will significantly speed up developer's way of debugging and troubleshooting bugs, giving them more time to work on new features.

7 Conclusion

The aim of this thesis was to show helpful ways on how organizations can refactor their telemetry systems. Hereby, the complex telemetry environment of Leica Geosystems represented the basis of the conducted case study. The main motiviations to refactor their environment revolved around not only reducing operative and maintenance costs, but also providing developers platforms to debug effectively. The first research question (RQ1) dealt with finding out refactoring strategies for a given telemetry system, while RQ2 required to conceive a plan on consolidating the telemetry systems of the case study.

The case study investigated four distinct telemetry systems, mainly by applying the Goal-Question-Metric framework to define dimensions across which the systems were evaluated and analyzed. In each pipeline, two software engineers and one product manager/engineer participated in the semi-structured interviews. A software director, who carries the responsibility for overseeing the software development process and technological progress of Leica's products, took part in the study too. Based on the inputs, positive and challenging aspects of each telemetry system were identified. Then, the thesis set requirements for the consolidated telemetry system. When refactoring the existing telemetry systems, the design decisions incorporated numerous best practices and trade-offs (RQ1). Thereby, OpenTelemetry represented a core pillar because it standardizes the way different services emit telemetry and therefore prevents vendor lock-in. Per definition, observability helps to slice and dice the data in ad-hoc and intuitive way, without needing to define the dimensions in advance. This may not be possible in the beginning as these debugging needs often become clearer over time. That is why developers should follow iterative instrumentation. Other results for RQ1 include configuration of filters, transformations and sampling on Collector-level. The trade-offs between building an own infrastructure with open source tools and buying a complete observability platforms were discussed as well.

Mapping the common practices to the set requirements, the thesis proposed to go with Grafana Cloud as storage and analysis solution for OpenTelemetry data. Datadog, New Relic, Dynatrace and Honeycomb were among the evaluated observability vendors. Grafana was chosen since it fulfilled three important needs: cost-effectiveness, interoperability and rich set of visualization techniques. Apart from Grafana, a mix of agent and gatway Grafana Alloys - Grafana's OTel Collectors - run in Kubernetes and carry the responsibility for collecting metrics, logs and traces.

As the OpenTelemetry libraries are not fully mature for browser instrumentation yet, Grafana Faro SDK was recommended next to OpenTelemetry in order to collect web vitals and errors, which are then automatically grouped into statistics and potential issues on the Grafana dashboard. In terms of desktop and mobile app (WinUI3 and .NET Maui) in this case study, OTel .NET library may be sufficient, but would require a lot of manual instrumentation and complicated querying on Grafana. This is because OTel support for such clients is very limited at the moment.

In conclusion, approx. 85% of the tools could be removed from the telemetry environment. Though, this remains an optimistic result because a more comprehensive evaluation by the developers is still ongoing. By migrating to OpenTelemetry and using Grafana, the limitations and requirements will become clearer. In addition, this study mentioned that observability data can increase productivity of product engineers and managers. Future studies could investigate in a similar direction by delivering more details towards the usage of such data for other target groups, such as sales and executive teams. They could also address concepts that were not applied in case study's context. An example would be dynamic observability, which adjusts the volume of metrics, logs, and traces collected according to environmental cues. This approach ensures that data collection intensifies during critical moments, like incidents and peak traffic, and diminishes during normal operations.

Bibliography

[1] L. Lawson. "Datadog's $65M Bill and Why Developers Should Care". (May 2023), [Online]. Available: https://thenewstack.io/datadogs-65m-bill-and-why-developers-should-care/ (visited on 03/04/2024).

[2] J. Riedesel, *Software Telemetry: Reliable logging and monitoring*, 1st ed. USA: Manning, Aug. 2021, ISBN: 978-1-61729-814-1.

[3] K. Woods. "Gartner IOCS Conference Recap: Monitoring and Observing Environments with Telemetry Pipelines". (Dec. 2023), [Online]. Available: https://www.mezmo.com/blog/my-recap-on-the-gartner-it-infrastructure-operations-cloud-strategies-conference (visited on 04/24/2024).

[4] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*, 1st ed. USA: O'Reilly Media, May 2022, ISBN: 978-1-4920-7644-5.

[5] T. Young and A. Parker, *Learning OpenTelemetry. Setting Up and Operating a Modern Observability System*. O'Reilly Media, Mar. 2024.

[6] R. Miles, *The Fundamentals of Telemetry Pipelines*. USA: O'Reilly Media, Sep. 2023, ISBN: 978-1-098-15384-7.

[7] Mezmo. "Introduction: Telemetry data optimization with mezmo pipelines". (2023), [Online]. Available: https://docs.mezmo.com/practioner-guide-data-optimization/data-optimization-introduction (visited on 03/18/2024).

[8] J. Kosińska, B. Balis, M. Konieczny, M. Malawski, and S. Zieliński, "Toward the Observability of Cloud-Native Applications: The Overview of the State-of-the-Art", *IEEE Access*, vol. PP, Jan. 2023. DOI: 10.1109/ACCESS.2023.3281860.

[9] F. Zaman. "Escaping the Cost/Visibility Tradeoff in Observability Platforms". (Jan. 2024), [Online]. Available: https://www.honeycomb.io/blog/escaping-cost-visibility-tradeoff-observability-platforms (visited on 02/20/2024).

[10] C. Klein. "How to tackle spiraling observability costs". (Jan. 2023), [Online]. Available: https://logz.io/blog/how-to-tackle-spiraling-observability-costs/#how-to-reduce-observability-costs-without-jeopardizing-visibility (visited on 04/22/2024).

[11] B. Conn. “4 strategies to reduce observability costs – without sacrificing visibility”. (Dec. 2023), [Online]. Available: https://www.circonus.com/2023/12/4-strategies-to-reduce-observability-costs-without-sacrificing-visibility/ (visited on 04/22/2024).

[12] Circonus. “Dynamic observability”. (2023), [Online]. Available: https://www.circonus.com/resources/dynamic-observability/ (visited on 04/22/2024).

[13] G. Siegfried, M. Bangera, M. Crossley, and B. Padraig. “Magic quadrant for application performance monitoring and observability”. (Jul. 2023), [Online]. Available: https://www.gartner.com/doc/reprints?id=1-2EF13QIG&ct=230710&st=sb&submissionGuid=ec51803d-12f6-40c3-a509-349b4a55229d (visited on 04/24/2024).

[14] S. Wickramasinghe. “Telemetry 101: An introduction to telemetry”. (2023), [Online]. Available: https://www.splunk.com/en_us/blog/learn/what-is-telemetry (visited on 05/17/2024).

[15] Last9. “Observability vs. telemetry vs. monitoring”. (Aug. 2023), [Online]. Available: https://last9.io/blog/observability-vs-telemetry-vs-monitoring/ (visited on 05/18/2024).

[16] ProtonsAI. “Observability vs. telemetry vs. monitoring”. (Feb. 2024), [Online]. Available: https://blog.protons.ai/monitoring-to-observability-evolution-from-monoliths-to-cloud-native-microservices-a5e26db8d54f (visited on 05/20/2024).

[17] MoldStud. “The role of data analytics in software engineering”. (Feb. 2024), [Online]. Available: https://moldstud.com/articles/p-the-role-of-data-analytics-in-software-engineering (visited on 05/20/2024).

[18] C. Cote. “4 types of data analytics to improve decision-making”. (2021), [Online]. Available: https://online.hbs.edu/blog/post/types-of-data-analysis (visited on 05/25/2024).

[19] V. R. Basili, G. Caldiera, and D. H. Rombach, “The Goal Question Metric Approach”, in John Wiley & Sons, 1994, vol. I.

[20] V. Venkatesh, J. Thong, and X. Xu, “Consumer Acceptance and Use of Information Technology: Extending the Unified Theory of Acceptance and Use of Technology”, *MIS Quarterly*, vol. 36, pp. 157–178, Mar. 2012. DOI: 10.2307/41410412.

[21] A. Anand. “9x more value for money than datadog - signoz”. (May 2023), [Online]. Available: https://signoz.io/blog/pricing-comparison-signoz-vs-datadog-vs-newrelic-vs-grafana/ (visited on 05/19/2024).

[22] C. Shih. “Grafana for business intelligence: How grafana labs uses dashboards for more than observability data”. (May 2022), [Online]. Available: https://grafana.com/blog/2022/05/31/grafana-for-business-intelligence-how-grafana-labs-uses-dashboards-for-more-than-observability-data/.

Statement of Affirmation

I hereby declare that all parts of this thesis were exclusively prepared by me, without using resources other than those stated above. The thoughts taken directly or indirectly from external sources are appropriately annotated. This thesis or parts of it were not previously submitted to any other academic institution and have not yet been published.

Dornbirn, 09 July 2024 Mert Öztürk