FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Monitoring design patterns for cloud applications

Carlos Jorge Direito Albuquerque

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Filipe Figueiredo Correia

July 29, 2022

Monitoring design patterns for cloud applications

Carlos Jorge Direito Albuquerque

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Nuno Flores
Referee: Prof. Alfredo Goldman
Supervisor: Prof. Filipe Figueiredo Correia

July 29, 2022

Abstract

Monitoring is essential to ensure the quality of an application. Recent trends in Cloud Computing and Microservices Architecture demand a change in monitoring processes. Like many parts of computer science, monitoring is guided by a set of best practices. Most companies adopt them, even if with different variations. However, practices may not be apparent to everyone and, most of the time, are not sufficiently explained or documented to be learned quickly and communicated effectively. Moreover, there is a lack of research on formalising monitoring practices to solve the mentioned problems.

To tackle this issue, this dissertation formally describes a set of practices as design patterns. Design patterns provide enough structure and detail to be easily reused by practitioners and have space to accommodate different needs and quirks depending on the usage context.

The proposed patterns are based on existing literature stemming from industry best practices that are further detailed and adapted to design patterns. Relations to existing monitoring patterns are also analysed to point the reader to more patterns that, together with the ones presented in this dissertation, compose a base catalogue for monitoring applications in the Cloud. An empirical study is conducted with an ongoing industry project to validate the proposed monitoring patterns. Patterns are put to practice in the project and validated by its team members to evaluate the adequacy of the patterns' descriptions. The study does not concretely falsify any of the patterns applied to the project. Nonetheless, its results are important insights that are used to improve the patterns and, consequently, the pattern catalogue.

Keywords: Monitoring, Design Patterns, Cloud Computing, Microservices, Empirical Study

Resumo

Monitorizar um aplicação é essencial para garantir a sua qualidade. As tendências atuais na adoção de computação na Cloud e arquitetura de Microserviços exigem uma alteração nos processos de monitorização. Em semelhança a outras partes da ciência de computadores, a monitorização de aplicações segue um conjunto de boas práticas. A maioria das empresas adota-as, ainda que com diferentes variações. Contudo, práticas podem não ser aparentes para toda a gente e, na sua grande maioria, não estão bem explicadas ou documentadas para serem aprendidas rapidamente e comunicada de forma eficaz. Para além disso, há pouca investigação no sentido de formalisar as práticas de monitorização para resolver os problemas anteriormente mencionados.

Para abordar este desafio, a presente dissertação descreve formalmente um conjunto de práticas sob a forma de padrões de desenho. Estes padrões proporcionam estrutura e detalhe suficiente para serem reutilizados por praticantes na área e têm liberdade para acomodar diferentes necessidades e peculiaridades dependendo do seu contexto de utilização.

Os padrões propostos são baseados em literatura atual e resultam de boas práticas da indústria que são posteriormente detalhadas e adaptadas a padrões de desenho. As relações com padrões de monitorização existentes são também analisadas com o intuito de encaminhar o leitor para mais padrões que, em conjunto com aqueles propostos nesta dissertação, compõem um catálogo base para monitorizar aplicações na Cloud. Um estudo empírico é conduzido com um projeto em desenvolvimento da indústria por forma a validar os padrões de monitorização propostos. Os padrões são aplicados na prática nesse projeto e validades pelos membro de equipa daquele com o objetivo de avaliar quão adequadas são as descrições dos padrões face à realidade. Do estudo não resulta a falsificação de nenhum dos padrões aplicados ao project. Ainda assim, do estudo resultam informações relevantes que são usadas para melhorar os padrões e, consequentemente, o catálogo de padrões.

Palavras-chave: Monitorização, Padrões de desenho, Computação na Cloud, Microserviços, Estudo empírico

Acknowledgements

I got to be honest. The dissertation process was hard and frustrating. If it were not for my supervisor, professor Filipe Correia, I would not have delivered such complete and well-thought work. Thank you very much for your advice, guidance and constant feedback, Filipe. More than my professor, you became my mentor and friend, which I greatly value. I hope we can keep working together. Moreover, I was fortunate to have not one but two excellent supervisors. I was also guided by José Silva, who was always very supportive, joyful and concerned with my work and well-being. I learned much from you, José, especially regarding managing people and time. I see a lot of potential in you, and I wish you all the best!

On a more personal level, I can not thank my family enough for the support, concern and help that they all provided me with. Thank you mother, for not judging me even when I was wrong. Thanks father, for teaching me that effort is always rewarded. Thank you brother, for helping me keep my inner child alive and for all the company throughout these years. Last but not least, an extraordinary person accompanied me through this process: my girlfriend. To you, Beatriz, my most sincere gratitude and respect. You are an endless source of joy and energy. You took care of me when I was lowest and always celebrated my achievements more than everybody else. Love is complicated and takes a lot of effort, but it is worth it. Thanks for everything, *coisa boa*!

Of course, I could not end this cheesy rant without thanking my friends. I carry every single one of you with me, and I am very thankful for that. A special thanks to Tito, Bernardo and Cunha for all the car drives, discussions and adventures we went on. The three of you are my best friends from university, and you were crucial to my success during these five years. I also want to thank Carolina, Luís and Gonçalo for being such incredible friends. Even if I do not talk to you every day, you have been my best friends for a very long time, and the love I got for you has only increased as the years went by. Distance does not matter at all with you guys. You are true friends no matter what. Finally, a huge thanks to all my friends from my exchange in Sweden. You are too many to list here, and I am still amazed by how fast one person can become such a great friend. Thank you so much for helping me have one of the best experiences of my life and showing me that I am a citizen of the world.

To conclude, there are many people that I have not mentioned here that were still important. It is unfair to select just a few for these paragraphs but a necessary evil. Therefore, I want to thank everyone who was somewhat part of this journey and helped me grow and experience new things.

It is incredible how five years passed by so fast. My father always told me these would be the best years of my life, and they were definitely great. I hope this is just the start of many great things. People say I am always smiling, but that is easy when you are fortunate to have these amazing people by your side. Let us change the world together!

Carlos Jorge Direito Albuquerque

"Never doubt that a small group of thoughtful, committed people can change the world. Indeed, it is the only thing that ever has."

Margaret Mead

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

ADFS	Active Directory Federation Services
AI	Artificial Intelligence
AOP	Aspect-Oriented Programming
API	Application Programming Interface
CPU	Central Processing Unit
ESRQ	Empirical Study Research Question
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ID	IDentifier
IT	Information Technologies
JSON	JavaScript Object Notation
LRQ	Literature Review Question
MC	Monitoring Challenge
MVC	Model-View-Controller
NIST	National Institute of Standards and Technology
OData	Open Data Protocol
PaaS	Platform as a Service
RAM	Random Access Memory
REST	Representational State Transfer
RQ	Research Question
RUM	Real-User Monitoring
R&D	Research and Development
SaaS	Software as a Service
SLA	Service Level Agreement
SPA	Single page application
SSI	Semi-Structured Interview
UI	User Interface
XML	eXtensible Markup Language

Chapter 1

Introduction

This chapter introduces the topic of the dissertation. In Section 1.1 we explore the context in which it is being developed. Afterwards, in Section 1.2, motivation to why this topic is important is provided and then, in Section 1.3, the goals of this work are discussed. Finally, an overall structure of the document is made available in Section 1.4 to make the chapters easier to navigate.

1.1 Context

Monitoring is:

> *"A process that fully and precisely identifies the root cause of an event by capturing the correct information at the right time and at the lowest cost in order to determine the state of a system and to surface the status in a timely and meaningful manner."* [32, p. 7]

Monitoring evolved along with the computational environment. As distributed systems emerged, new definitions and practices of monitoring were adopted [47]. Then, Grid computing came and monitoring adapted [98]. The point is that monitoring applications is important because it ensures the quality of an application [23]. Therefore, monitoring will adapt to new realities and grow along the way.

Trends in Cloud computing [34] and Microservices architecture [39] demand a change in monitoring processes [96]. In fact, Newman [64] argues that successfully monitoring Microservices requires a complete change of mindset. Thus, monitoring had to evolve to this new reality and new practices have emerged.

Like many parts of computer science, monitoring is guided by a set of best practices. These practices are a result of the evolution of the environment and are usually consolidated solutions for recurring problems based on professional experience of practitioners [64]. Companies widely adopt and tailor them to their specific needs on the Cloud. However, practices may not be apparent

to everyone and, most of the time, are not sufficiently explained or documented to be learned quickly and communicated effectively.

A way to tackle this issue is formalising monitoring practices as design patterns. Each pattern is "a solution to a problem in a context" [38, p. 3], and is expected to result in different concrete solutions depending on the specific context it is being used [6]. These properties resemble the way companies are adopting practices, since different companies follow relatively similar processes, but adapt their tools and solutions to their specific needs.

1.2 Motivation

Monitoring practices are the result of industry changes on monitoring processes to tackle the new computation challenges created by the increasing adoption of the Cloud [34] and Microservices [39]. Studying them is important for a two big of reasons: (1) they can greatly impact the quality of an application and (2) they are hard to communicate and reuse.

First of all, as we have stated in the previous section, monitoring is a very important process for Cloud computing [2]. When implemented correctly, Cloud monitoring solutions should be, among other properties, scalable, robust, customisable and extensible in order to improve capacity planning, configuration management, security, privacy and fault management [32]. Thus, getting monitoring right might be a complex challenge, which forces companies to resort to practices as a way to reduce it. If practices are not clear, documented or properly validated, how can they be of any use to the industry?

Secondly, practices are a result of the work of companies on the topic. They emerge as the industry understands what works better or worse, and they get consolidated over time as best practices that are simply accepted and adopted. They show up mostly in grey literature [64, 72, 71, 44, 31], and, to our best knowledge, when they appear in scientific literature they are usually based on grey literature [96, 54, 95]. Therefore, practices usually lack formalisation and a standard structure that would make all of them equally reusable.

In addition, monitoring practices are very context dependant. Some companies share their practices and guidelines through online posts [43, 44, 30] or through published books [31]. Nonetheless, the fact that monitoring practices are public does not mean they are reusable. Every project has its intricacies and specific needs, so the way monitoring is implemented naturally changes from case to case. Moreover, certain characteristics of the company and the actual project influence greatly the way monitoring is carried out. A larger company with more resources may go an extra mile to guarantee its applications are properly monitored, while a startup may need to restrict monitoring to the bare minimum that brings the greatest value. To summarise, without a formal description of practices that facilitates communication, reusability and implementation, practices by themselves may not be enough to help tackling the monitoring challenges the industry faces.

Furthermore, there is little research targeting the formalisation of monitoring practices. As mentioned in Section 1.1, one way to achieve this formality is the use of design pattern. Some works do provide patterns for the Cloud [14, 83, 88, 89, 85, 87, 84, 55, 28, 1], for Service Oriented

Architectures (SOA) [43] and for Microservices [72, 71, 13], but these topics are very broad so we believe there is not enough focus on monitoring. Therefore, many monitoring practices that are currently adopted by the industry are not yet formalised as design patterns.

As a final note, this dissertation is developed in partnership with *Konkconsulting - Consultoria Informática S.A.*, a "software development company, specialised in the design and implementation of processes in the area of Human Resources" [50]. The company is challenged with monitoring some of its applications deployed in the Cloud. One of its developers declares that it is hard to understand the source of a failure due to the distributed nature of Cloud applications. Even though there exist practices to tackle problem such as this, it is not easy to reuse them in a specific context. This dissertation is the result of addressing the company's needs and the lack of research on practice formalisation.

Concluding, we strongly believe that enhancing existing practices by formally describing them as design patterns and then empirically validating them will greatly benefit the industry. In the following section, these goals are further detailed and more rationale is provided on how we think this benefit will be achieved.

1.3 Goals

This dissertation aims at solving the issues exposed in the previous section through a pattern catalogue for monitoring cloud applications. Design patterns provide enough structure and detail to be easily reused by practitioners and have space to accommodate different needs and quirks depending on the usage context. The proposed patterns are based on existing literature stemming from industry best practices that are further detailed and adapted to design patterns. Not all practices are good pattern candidates, so part of this work consists of critically analysing existing practices and choosing which ones to formalise. In addition, we intend to analyse relations with existing monitoring patterns to build upon existing literature and make the proposed patterns as complete and detailed as possible.

Furthermore, this work seeks to validate the pattern catalogue empirically. To that end, a case study is conducted with an ongoing project of *Konkconsulting*. We characterise the project and report its main monitoring challenges. Afterwards, we select adequate monitoring patterns to tackle these challenges and implement them in the project. Finally, we organise a focus group to get new insights to complement the patterns and validate the work done so far. This way, we intend to understand how accurate the patterns' descriptions are and improve them to provide a better pattern catalogue to practitioners.

By making it easier and more efficient for teams to adopt monitoring processes, this dissertation strives to incentivise monitoring adoption and, as a broader consequence, to increase the overall quality of Cloud applications.

1.4 Document Structure

This document is made of seven chapters. In Chapter 1, Introduction, the topic of this work is introduced and the context, problem and motivation are explained. This chapter helps the reader understand the basic goals of this dissertation and captivate their attention to the problem at hand. In Chapter 2, Background, the key concepts used throughout this dissertation are introduced and described. These concepts are crucial to understand part of this work, so this chapter ensures the reader is aware of them.

In Chapter 3, The state of monitoring in the cloud, the literature review process is explained and conclusions regarding the state of the art on the topic are presented. Starting in this chapter and for the remainder of the thesis, we use SMALL CAPS text to highlight design patterns in the text. This chapter is the backbone of this work because it is where we identify existing monitoring practices and design patterns that will dictate what patterns this dissertation can develop. Afterwards, in Chapter 4, Problem statement, the problem, scope, research question and methodology of this work are thoroughly defined. These guide the remaining of this dissertation towards its goals.

Next, in Chapter 5, A catalogue of monitoring design patterns, the pattern catalogue for monitoring cloud applications is presented. This chapter explains the methodology used to mine the patterns and how the literature review played an important part on this process. Moreover, it provides an overview of the developed patterns and explains their structure and considerations. Next, in Chapter 6, Empirical study, the empirical research goals, methodology and three phases are explained. It picks on the proposed patterns to provide initial corroboration. Additionally, its results provide insight to improve the design patterns, which leads to a revised pattern catalogue of improved quality. Finally, in Chapter 7, Conclusions and future work, the main contributions and conclusions of this dissertation and possible new directions of research are discussed.

Chapter 2

Background

In this chapter we provide a brief explanation of core concepts used throughout this dissertation. In Section 2.1 we write about Cloud computing and its characteristics. Then, in Section 2.2, a definition of monitoring and its main capabilities is provided. Finally, in Section 2.3, the concept of design patterns and their relationship with practices is introduced.

2.1 Cloud computing

Cloud computing is defined by the National Institute of Standards and Technology (NIST) as:

> *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [58, p. 6]

The same work defines the following essential characteristics of Cloud computing: on-demand self-service, broad network access, resource pooling, rapid elasticity, measured service [58]. Because of this, Cloud is a very attractive model that has seen very high numbers of adoption [34].

Depending on what is provided to the consumer, Cloud applications can be characterised into 3 different service models: *Software as a Service (SaaS)*, *Platform as a Service (PaaS)*, and *Infrastructure as a Service (IaaS)*. In the first, SaaS, the whole application is managed by the provider and consumers are only given the capability to use it. In the second, PaaS, the provider manages the whole Cloud infrastructure where the consumer can deploy its applications. In the last, IaaS, the consumer "has control over operating systems, storage, and deployed applications" [58, p. 3] while the provider is in charge of managing the underlying cloud infrastructure. [58].

Finally, the NIST also defines 4 deployment models for the Cloud, which classify clouds according to where they are located and who manages them. They are defined in [58] as:

- Private cloud - a single organisation and its comprising consumers use the cloud;

- Community cloud - the cloud is used by a specific community of organizations that have shared concerns;
- Public cloud - the cloud can be used by the general public;
- Hybrid cloud - the infrastructure is a mix of the 3 above types by bounding them together.

The description we provide of the Cloud is not the most detailed. Nonetheless, we believe it provides enough information to understand this dissertation.

2.2 Monitoring

Monitoring has been an important process for software engineers for a while. It became essential to tackle challenges with distributed systems [47, 57] and has constantly evolved along with the technological landscape [98, 2].

That said, there are more than one monitoring definitions [57, 51]. As this dissertation's focus is monitoring cloud applications, we chose to use a definition of monitoring in the context of the cloud. Thus, we consider monitoring to be:

> *"A process that fully and precisely identifies the root cause of an event by capturing the correct information at the right time and at the lowest cost in order to determine the state of a system and to surface the status in a timely and meaningful manner."* [32, p. 7]

The same work that provides this definition also lists the following capabilities of cloud monitoring tools:

- **Scalability** - "a monitoring tool needs to be scalable to deliver the monitored information in a timely and flexible manner" [32, p. 7]
- **Portability** - "the portability of monitoring tools is indispensable to facilitate efficient management of such environments and to achieve wide penetration across multiple Clouds" [32, p. 7]
- **Non-intrusiveness** - "a monitoring tool should consume as little resource capacity as possible on the monitored systems so as not to hamper the overall performance of the monitored systems" [32, p. 7]
- **Robustness** - "a monitoring tool needs the ability to adapt to a new situation by continuing its operation in the changed environment" [32, p. 7]
- **Multi-tenancy** - "to support multi-tenancy provisioning, the Cloud monitoring tool should maintain concurrency and isolation" [32, p. 7]

- **Interoperability** - "a modern monitoring tool should be capable of sharing monitoring information between heterogeneous Cloud components for managing collaborative operations" [32, p. 7]
- **Customisability** - monitoring tools "must be able to manage the service customisation of each customer" [32, p. 7]
- **Extensibility** - "the monitoring tools need to be extensible and be able to adapt to new environments" [32, p. 7]
- **Shared resource monitoring** - "a Cloud monitoring tool needs the capability of supervising shared resources to manage such an environment" [32, p. 8]
- **Usability** - "to be highly usable a monitoring tool should facilitate deployment, maintenance and human interaction" [32, p. 8]
- **Affordability** - "being open source would also have a positive impact on the prominence of a monitoring tool" [32, p. 8]
- **Archivability** - "a monitoring tool should possess a means of storing historical data" [32, p. 8]

Although these capabilities are specified in the context of monitoring tools, they provide a good rationale behind monitoring main usages. When correctly implemented, monitoring helps both Cloud providers and Cloud consumers in areas such as: accounting and billing, SLA management, service/resource provisioning, capacity planning, configuration management, security and privacy assurance and fault management [32].

Like many parts of software engineering, monitoring is guided by a set of best practices. These practices are a result of the evolution of the environment and are usually consolidated solutions for recurring problems based on professional experience of practitioners [64]. Companies widely adopt and tailor them to their specific needs on the Cloud.

2.3 Design Patterns

Design patterns were first used in the field of Civil Architecture by Cristopher Alexander in his work *A pattern language* [6]. In it, he provides the following definition:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [6]

Patterns were then adopted by Gamma et al. [38], which state that, at their core, patterns are "a solution to a problem in a context" [38, p. 3]. This work popularised design patterns in the software engineering field.

The process of explaining a pattern is called *pattern mining*. This is usually done through induction, i.e. "patterns are derived from practical experience and not deduced from theories" [49]. As Fowler puts it, "a key part of patterns is that they're rooted in practice" [36, p. 10]. Therefore, there is a strong relationship between practices and patterns. The difference lies in the fact that practices exist informally in the wild while design patterns are a creative effort of describing practices in a structured and formal way.

Even though patterns are heavily based on practice, Kohls and Pahke state that:

> *"Because each pattern contains a network of hypotheses (the forces that link the context form to the solution form), we think it is adequate to consider a pattern as a theory and not only as an isolated hypothesis."* [49, p. 4]

We agree with this view of patterns as "theories about practice" [49, p. 4]. They are not inherently true, which means they should be tested empirically in order to increase the level of confidence that practitioners have in them. Only then, after successfully passing many of the tests, can a pattern be reliable and useful to the practitioner [49].

Chapter 3

The state of monitoring in the Cloud

The importance of monitoring for the Cloud is widely accepted [2]. Nonetheless, before contributing to the field we first need to understand what already exists and what contributions are worth making. To that end, in this chapter we conduct a thorough literature review on the state of monitoring in the Cloud.

In Section 3.1, we define the goals of the literature review process, identifying the key questions we seek an answer to. Afterwards, in Section 3.2, the approach used to find said answers is explained, pointing out the search queries and how we carried out the literature review. In Sections 3.4 and 3.5, the existing literature on monitoring practices and design patterns, respectively, is analysed and explained. In addition, some discussion around the existing body of work is provided to point towards a preliminary answer to the literature review questions. Finally, in Section 3.6, the main conclusions of this chapter are presented by answering to the aforementioned questions. Moreover, we explain how these conclusions prove the importance of this dissertation and further validate its goals.

3.1 Goals

As a starting point of this dissertation, a literature review was carried out to better understand the state of the art in monitoring Cloud applications. Its purpose was to answer the following literature review questions (LRQs):

LRQ1. What monitoring practices are mentioned by current literature?

This work proposes design patterns based on existing monitoring practices. As such, an initial understanding of said practices is important to guarantee the relevance of the design patterns to be developed. Furthermore, through this question we also want to conclude on practice adoption and the existence of variations for the same practice.

LRQ2. Which practices are formalised as design patterns?

Existing work on formalising the aforementioned practices as design patterns is important for this research. In order to contribute with new patterns, we need to know which ones are already defined, keeping in mind that not all practices can become patterns because they are either too broad or too narrow to be reusable solutions. The patterns proposed in this dissertation may refer to previously existing ones as part of their *Related Patterns* section. Moreover, existing work can show the adequacy of design patterns to solve Cloud monitoring challenges.

LRQ3. How detailed are existing monitoring design patterns?

Not all existing design patterns are described accurately and in detail. Gamma et al. argue that patterns are "a solution to a problem in a context" [38, p. 3]. If any of these parts is poorly described, then the pattern is harder to reuse and less relevant to the industry. As such, evaluating pattern detail is important to understand which patterns need to be further detailed to be easily communicated and used by professionals.

LRQ4. To what extent are monitoring practices or patterns empirically validated?

One of this dissertation's goals is to create design patterns that are relevant for the industry. Since the proposed patterns will be based on existing monitoring practices and patterns, it is important that these are empirically validated with the industry. Therefore, we want to understand what kind of studies are conducted by existing literature to show the value of existing practices and patterns.

3.2 Methodology

To answer the above LRQs we executed the search queries (SQs) in Table 3.1 in Google Scholar. We decided to use this search engine due to the high amount of indexed scholarly documents. Google Scholar would then redirect us to digital libraries (e.g. IEEE Xplore, Science Direct), paper archives (e.g. arXiv) or directly to the document. The search process was then complemented by following interesting references and related work to the initial results.

Table 3.1: Search queries used in the literature review process.

	Search Queries
SQ1	`cloud monitoring`
SQ2	`"cloud applications" monitoring`
SQ3	`microservice* monitoring`

Regarding the search queries, SQ1 was the broader of the three, providing a good starting point for taxonomy and fundamental concepts of Cloud monitoring. However, most results did not focus on application monitoring practices or patterns, which we achieved through SQ2 and SQ3. Microservices are just one type of Cloud application. Even though this dissertation targets

many kinds of Cloud applications, the trends in Microservices adoption and research make it a valuable source of monitoring challenges and practices.

After having the search results, each work's relevance to the topic was evaluated by reading its title, abstract and keywords, prioritising the ones closer to the literature review goals. Afterwards, we read the documents and took notes of their contents. Finally, a summary of each work's ideas, concepts and conclusions was registered on a spreadsheet, making it easier to compare existing literature.

The analysed literature includes books, conference papers, journal articles, industrial reports, blog posts, web pages and a PhD thesis. The inclusion of grey literature is due to the common practice of companies and professionals sharing solutions to their development and deployment issues through dedicated blogs or forums. This kind of literature was used to complement the formal research process and was carefully considered by the authors. Including it increases the number of industry practices found, making this dissertation more complete. We highlight grey literature in the tables used to summarise the analysed works (cf. Tables 3.2 and 3.4) for transparency reasons.

3.3 Classification for practices and patterns

Before diving into the answers to the literature review questions, we propose a classification for monitoring practices and design patterns. The classification is based on Gupta's [43] article on design patterns for observable services, where he separates patterns in three different categories: *Health checks*, *Real time alerting* and *Troubleshooting*.

Based on these three types of patterns, we propose a division of *Troubleshooting* into *Logging* and *Tracing*. Although tracing is mostly achieved through logging, we believe they are fundamentally different and applicable in different contexts. For instance, a team handling a monolithic architecture or just a few services can troubleshoot their application using only logging practices. In contrast, another team with a project containing hundreds of services deployed may need both logging and tracing to monitor their application effectively. Therefore, the division in *Logging* and *Tracing* patterns would allow the first team to search specifically for the first type without having to sort out patterns of the second type. We consider this an advantage over Gupta's [43] classification.

That said, throughout this chapter, we classify practices and patterns in four categories: *Health check*, *Remediation*, *Logging* and *Tracing*. Each category has a different purpose and context, creating a separation between patterns of each type. The *Health check* category encompasses every practice that provides a view into the application's current state. The *Remediation* category pertains to the practices that are set up to mitigate an issue after it occurs. The *Logging* category includes practices that help the team generate, manage, store and use logs as a troubleshooting artefact. Lastly, the *Tracing* category is used to classify all the practices that provide a way to track the flow of events in the system. We consider these descriptions to be enough to distinguish between patterns and practices found during the literature review, so we do not dive into more detail into each category.

3.4 Monitoring practices

In Table 3.2 an overview of the literature we found on monitoring practices is presented. Before diving into the practices, we want to clarify that not every practice is a design pattern. Some practices we found are either too broad in scope or are too abstract to become a helpful pattern. For example, Newman [64] describes the concept of *Semantic Monitoring*. The main idea behind this practice is that we should define a model of our system to answer the question, "is the system behaving the way that we expect?" [64, p. 333]. Even though this is a useful guideline, the solution is, in our opinion, too abstract to become a usable monitoring pattern. The design patterns we propose are on a lower, more actionable level of abstraction that we believe is best suited for the problem we are tackling. We include these practices in our analysis for completeness but discard them as potential pattern candidates.

That said, Newman's [64] work is the one that introduces more practices and the only one that covers all four categories. This is one of the most comprehensive collection of practices we found during the literature review. However, the author does not conduct any empirical study to validate his contribution. Instead, he proposes practices based on his professional experience as a consultant for the industry, providing further insight on common pitfalls and challenges when adopting said practices.

Table 3.2: Related work on monitoring practices. Note that the second column is the number of practices for each work. The grey literature works are highlighted in bold text.

Work	#	Logging practices	Tracing practices	Remediation practices	Health check practices	Empirical validation
Newman [64]	9	Standard Logging Log Aggregation	Correlation ID Distributed Tracing	Alerting	Metrics Aggregation Testing in Production Semantic Monitoring Synthetic Transactions	
Ewaschuck [30]	3			Symptom-Based Alerting Classifying Alerts Alert Just Enough		
Ewaschuck [31]	1				The Four Golden Signals	
Cito et al. [21]	1				Synthetic Monitoring	
Kubernetes [52]	2				Liveness Health Endpoint Readiness Health Endpoint	
Dinesh [26]	2				Liveness Health Endpoint Readiness Health Endpoint	
Gupta [44]	1				READS Metrics	
Li et al. [54]	4	Log Collection Log Sampling	Correlation ID Tracing Pipeline			10 Interviews
Waseem et al. [96]	7	Logging Management Audit Logging Log deployment	Distributed Tracking Exception Tracking		Health Check API Monitoring Metrics	106 Survey Answers 6 Interviews

Ewaschuk [30] writes about the issues and challenges of real time alerting and, as he provides solutions to them, he explains in detail what we consider to be monitoring practices. Thus, from his work we extracted three practices dedicated to how alerting should be implemented. In another

work, the author also proposes a set of metrics, what he calls *The Four Golden Signals*, that he considers fundamental to any monitoring process: latency, traffic, errors and saturation [31]. Although being a *Health Check* practice, these metrics are used by the author as criteria for alerts, which denotes a relation between categories.

Cito et al. [21] simulate three different levels of performance problems and evaluate how effectively synthetic monitoring and real-user monitoring (RUM) are to tackle them. They start by explaining both techniques and identifying the significant causes of server performance degradation. Afterwards, the authors create mathematical models to be able to simulate these causes in a controlled environment. Then, they apply synthetic monitoring and RUM practices to detect the simulated performance issues. By the end of the study, Cito et al. conclude that synthetic monitoring "is beneficial for maintaining a constant schedule to reliably gauge availability and performance" [21, p. 438], but that it is a limited representation of what a real user would do. These provide further information on the practice's consequences.

Kubernetes' [52] documentation and Dinesh's [26] blog post explain the purpose and discuss implementation details for readiness and liveness probes. Both of these works state that Kubernetes uses the readiness probe to check if an application is ready to serve traffic, while the liveness probe tests the application's ability to respond to requests. They may seem similar at first, but their consequences differ significantly. The readiness probe's sole purpose is to "ensure that traffic does not reach a container that is not ready for it" [52, p. 1], while the liveness probe guarantees "that containers are restarted when they fail" [52, p. 1]. Moreover, the examples provided by the works mentioned above further accentuate this distinction. On the one hand, the readiness probe could fail when the application depends on an external service that fails [52]. On the other hand, the liveness probe is more concerned with the application's internal state. When it ends up in an error state, such as a deadlock that makes the application unable to serve requests, it needs to be restarted [26].

In addition, we have Gupta's article [44] that further explores health check practices, more specifically, guidelines for metrics collection. The author proposes the *READS* acronym, which stands for: request rate, errors, availability, duration/latency and saturation. Gupta goes in further detail explaining the reason to collect each of these metrics and suggests measurement units for them. Gupta's metrics are very similar to Ewaschuk's, which indicates that the metrics are likely a good monitoring practice. Nevertheless, none of the authors provide empirical validation on the covered practices, lacking some evidence of actual industrial validity.

On the contrary, the last two works of Table 3.2 do provide empirical validation for their practices. Li et al. [54] explore the impact of distributed tracing on the observability of Microservice applications. The authors argue that tracing implementations follow a pipeline composed by (1) logging, (2) collection, (3) preprocessing, (4) storage and (5) analysis. From this pipeline we extracted four logging and tracing practices listed in Table 3.2. To assess this hypothesis, they conduct interviews with 25 professionals from 10 companies of different sizes and different domains. The results show that only the two smallest companies do not use distributed tracing as a means to monitor their Microservices, while the rest do follow the tracing pipeline. In addition,

the tracing pipeline is used with variations, that is, companies follow the same steps but implement them differently according to their specific context (e.g. Microservice size, performance requirements, number of Microservices). Furthermore, larger companies tend to create and maintain their own tracing systems while smaller companies prefer to use open source systems to save time.

Finally, Waseem et al. [96] explore how Microservice applications are designed, tested and monitored. To do so, they start by identifying 6 monitoring practices and 12 monitoring metrics from grey literature. We consider the metric collection a practice, independently of the metrics used, so in Table 3.2 we list 7 practices for this work. Moreover, the authors conduct a mix-methods study with a survey with 106 replies and 6 interviews with Microservice professionals. The results indicate that *Log Management*, *Exception Tracking* and *Health Check API* are the most used practices and that *Resource Usage*, *Load Balancing*, and *Availability* are the most used metrics. In addition, the authors found that the collection of metrics and logs from containers, and distributed tracing are the biggest monitoring challenges for practitioners. Lastly, as a research implication they conclude that, even though companies use tools and frameworks to tackle monitoring challenges, "further research can be explored to develop and empirically evaluate the solutions for addressing the challenges, such as having many components to monitor (complexity), performance monitoring, and distributed tracing" [96, p. 32].

Now that it has been explained, we present a brief discussion of the literature analysed so far. It suggests 30 practices in total. However, when comparing practices from different works, we can conclude that some of the practices are used under different names. For instance, we believe that the *Health Check API* practice is an aggregation of *Liveness Health Endpoint* and *Readiness Health Endpoint* because these tackle the same issue but with greater detail, which we believe is important for the design patterns. In addition, we argue that *Log Aggregation* and *Log Collection* refer to the same practice — to collect (or aggregate) logs in a central repository that can be easily queried. We believe that slight variations in the practices' names and descriptions are due to the existing variations in the way companies implement monitoring on their systems that Li et al. [54] verified empirically. Since design patterns can be applied many times over in different ways [6], they seem adequate to tackle the variations between the same core practices.

When considering only distinct practices, we shrink the list to 17 that are portrayed in Table 3.3. The names chosen are mere preference among the nomenclature used in the literature. Because they appear repeatedly among different works, we can also conclude with some confidence that these are established practices, even though not all of them are suitable design pattern candidates.

To conclude this section, it is important to note that only 2 out of the 9 analysed works provide empirical evidence of the practices' relevance. This means that, from the practices in Table 3.3, 10 are properly validated. As stated before, this does not mean the remaining 7 should be neglected. Instead, they accentuate the importance of empirically validating the proposed design patterns to guarantee their adequacy for the industry.

Table 3.3: Distinct monitoring practices identified in the literature review, by category.

Category	Practices	Total
Logging	Log Aggregation, Standard Logging, Log Sampling, Audit Logging, Deployment Logging	5
Tracing	Correlation ID, Distributed Tracing, Exception Tracking	3
Remediation	Symptom-Based Alerting, Classifying Alerts, Alert Just Enough	3
Health Check	READS Metrics, Metrics Aggregation, Semantic Monitoring, Synthetic Testing, Liveness Health Endpoint, Readiness Health Endpoint	6

3.5 Monitoring design patterns

A summary of the analysed literature regarding monitoring design patterns is shown below in Table 3.4. We found 4 different works on the topic that suggest a total of 19 design patterns. Only 1 work provides empirical validation on the proposed design patterns. In this section, we describe each work in the scope of monitoring and, at the end, provide a discussion around them. In addition, we evaluate the level of detail of each pattern. This evaluation is based on (1) the formal structure of the pattern, which must include, at least, context, problem, forces, solution and consequences, and (2) how thorough is the description of each of these parts so that they provide enough information for practitioners.

Table 3.4: Related work on monitoring design patterns. Note that the second column is the number of patterns for each work. The grey literature works are highlighted in bold text.

Work	#	Logging patterns	Tracing patterns	Remediation patterns	Health check patterns	Empirical validation
Richardson [71]	7	Log Aggregation Audit Logging Log Deployments	Distributed Tracing Exception Tracking		Health Check API Application Metrics	
Gupta [43]	5	Logging Error Conditions	Distributed Tracing	Remediating Issues	Outside-In Health Check Inside-Out Health Check	
Brown et al. [14]	3	Log Aggregator	Correlation ID Query Engine			
Sousa et al. [83]	4	Preemptive Logging Log Aggregation		Automated Recovery	External Monitoring	100 Survey Answers 5 Startup SSIs

The first entry on Table 3.4 is Richardson's online wiki [71] that is based on his book *Microservices Patterns* [72]. In the latter, the author presents a pattern language with 44 design patterns to address many Microservices challenges, from adoption to observability. In fact, in the book the identified monitoring patterns fall under the chapter dedicated to observable services. Richardson writes about the patterns mixing the context, forces and problem together. Therefore, the patterns lack a formal structure that we believe is important for their communication and reusability. That issue is tackled by the online wiki [71], where the book's patterns, plus the LOG DEPLOYMENTS pattern which is not part of the original pattern language, are formally described. Every pattern in

the wiki has all necessary sections properly divided and follows a good formal structure, but the level of detail between patterns varies a lot. We consider that none of the patterns proposed by the author provides enough information to be effectively reused and communicated. This is the work that proposes the most design patterns out of the analysed literature. As a final note, neither in the book nor in the wiki the author conducts a study to validate the patterns empirically.

Next, Gupta's [43] article introduces 5 design patterns to enhance a service's observability. This is the only work to cover all 4 pattern categories and to propose a *Remediation* pattern, while the others just mention the possibility of remedying issues inside other patterns' descriptions. However, similarly to Richardson's [72] book, the patterns do not follow a formal structure and it is unclear what in the descriptive text is the context or the consequences. This article is a collection of patterns that are used by the company Gupta represents, so they are based on the professional experience of the author. Apart from that, there's no dedicated empirical validation of the patterns.

The following work by Brown et al. [14] is an online wiki of Cloud adoption patterns based on published pattern papers. The basis for this wiki is Brown and Woolf's [13] paper where they develop a pattern language for implementing Microservices. In the paper, the patterns are presented in a formal structure with all sections well detailed. In the wiki, the authors adopt a less formal structure similar to the one first used in [6], where there is no separation between sections. That said, the LOG AGGREGATOR and CORRELATION ID patterns, both proposed in the paper, contain all minimum required sections and we believe they have enough detail to fully capture the different aspects of these solutions and of the rationales for applying them. Similarly, we argue that the QUERY ENGINE pattern, introduced by the authors in the online wiki, provides a sufficient description and a semi-formal structure that make it just bare minimum accepted as a fully defined pattern. To conclude the analysis on this work, the absence of empirical validation does not allow for conclusions on these patterns adoption.

Lastly, Sousa et al. [83] propose a pattern language for the Cloud, where they include 5 design patterns relevant for monitoring. While PREEMPTIVE LOGGING, LOG AGGREGATION and EXTERNAL MONITORING are labelled by the author as monitoring patterns, AUTOMATED RECOVERY is labelled as a orchestration and supervision patterns. The latter remains important because it suggests that failures should be automatically remedied whenever possible, so we consider it to be a *Remediation* pattern. Moreover, we believe that the suggested patterns are very well detailed and described. This is an example of the pattern maturity we want to achieve with this dissertation. On another note, this is the only work that empirically validates its patterns. The authors conduct a survey with 100 responses and SSIs with 5 startups from the greater Porto region in Portugal. The survey reveals that around 56% of their patterns are used by respondents and that LOG AGGREGATION is the most adopted pattern, with 72% of respondents saying they adopt it. From the SSIs, Sousa et al. conclude that LOG AGGREGATION may be a good starting pattern for Cloud practitioners and that more mature teams tend to adopt more patterns.

Moving on to the discussion about the literature on monitoring design patterns, the lack of empirical validation pops out as the major issue. Only 1 out of 4 analysed works provides studies to measure pattern adoption and usefulness. We consider this a very low proportion of valida-

tion, thus, with this dissertation, we intend to reduce this issue. Furthermore, the results of the only work that conducts studies are very positive towards the usage of design patterns to solve Cloud engineering challenges, more specifically, the monitoring issues we are addressing with this dissertation.

Regarding the patterns, similarly to what is observed in Section 3.4, authors use different names to refer to the same design pattern. This phenomenon is to be expected since design patterns can even include an *Also Known As* section to mention other names to refer to the same pattern [38]. That said, from a total of 19 monitoring design patterns, we boil the list down to 13 distinct ones, portrayed in Table 3.5. We chose the names from the analysed literature simply by what we believe that better fits the pattern description. Note that only 4 of these patterns (i.e. AUTOMATED RECOVERY, EXTERNAL MONITORING, LOG AGGREGATION and PREEMPTIVE LOGGING) are empirically validated, which means the remaining 9 need further research to prove their usefulness for the industry.

Table 3.5: Distinct monitoring design patterns identified in the literature review, by category. The highlighted patterns are the ones with good structure and descriptions.

Category	Patterns	Total
Logging	Log Aggregation, Audit Logging, Preemptive Logging, Deployment Tracking	4
Tracing	Correlation ID, Distributed Tracing, Exception Tracking, Query Engine	4
Remediation	Automated Recovery	1
Health Check	Health Check API, Application Metrics, Infrastructure Metrics, External Monitoring	4

Some of the associations are based on the interpretation we made of the patterns and may not be obvious. We believe that LOG DEPLOYMENTS proposes logs just as a way to track the system deployments. Thus, we renamed the pattern to DEPLOYMENT TRACKING since tracking is the more generic solution. The INSIDE-OUT HEALTH CHECK pattern suggests the collection of metrics for the application and infrastructure to monitor the health of the system. Since there already exists an APPLICATION METRICS pattern, we believe that the INSIDE-OUT HEALTH CHECK contains enough information to write another pattern dedicated to INFRASTRUCTURE METRICS. We believe these should be two different design patterns because they have different consequences and slightly different contexts. The LOGGING ERROR CONDITIONS pattern postulates that every error should be logged to make it easier for develops to troubleshoot their applications. We merged this with PREEMPTIVE LOGGING because the latter suggests that log granularity should be tweaked to catch relevant information right on the first occurrence of an error, so we think it includes the former in its definition.

Another aspect we want to discuss is the lack of formalisation or detail of the patterns in the first two works, which implies that we only consider 6 of the distinct design patterns of Table 3.5 to be fully described and ready to use. Following the order in which they appear in the table, these patterns are: LOG AGGREGATION, PREEMPTIVE LOGGING, CORRELATION ID, QUERY

ENGINE, AUTOMATED RECOVERY and EXTERNAL MONITORING. This issue makes some of the patterns less relevant in comparison to others, which means we may opt to reference more detailed patterns in favour of less detailed ones throughout this dissertation. In addition, to make this dissertation's pattern collection as complete as possible, in Chapter 5 we build upon these less detailed patterns to improve their quality and provide more rationale in their sections.

3.6 Conclusions

In Section 3.1, the LRQs that guide this literature review were introduced. To provide an answer to the questions, they are once again mentioned below:

LRQ1. What monitoring practices are mentioned by current literature?

In Section 3.4, we explore existing literature on practices and end up with a list of 17 different monitoring practices that are represented in Table 3.3. We believe this list provides a complete view of established and mature monitoring practices. We also conclude that the same practice was mentioned by different authors through different names and that practices are used by the industry with variations. In other words, the same practice can result in different concrete solutions, which we argue is a reason to use design patterns to formalise them.

LRQ2. Which practices are formalised as design patterns?

In Section 3.5, we analyse the literature to find out which monitoring design patterns exist. The result is a list of 13 distinct patterns outlined in Table 3.5. A match between the practices and design patterns found is presented in Table 3.6 to better illustrate which practices have a pattern counterpart. We conclude that there are 9 practices formalised as patterns and 8 that still are not. At a first glance, this means that half of the practices have a matching pattern.

However, some of these practices are not suitable pattern candidates as explained in Section 3.1. Taking that into account, the number of suitable pattern candidates drops to 6 because we argue that *Semantic Monitoring* and *Alert Just Enough* are too generic to become design patterns. Additionally, in Section 4.2, an explanation is provided to why alerting is out of the scope of this work. Thus, by taking out *Classifying Alerts* and *Symptom-Based Alerting*, the result are only 4 suitable practices.

One of this dissertation goals is to contribute with new patterns for existing monitoring practices. Through this literature review, we already identified 4 pattern candidates that reinforce the importance of this work: *Synthetic Testing*, *Standard Logging*, *Log Sampling*, and *Readiness Health Endpoint*.

LRQ3. How detailed are existing monitoring design patterns?

In Section 3.5, we also judge pattern quality based on their structure and description, as explained in the start of that section. As portrayed in Table 3.5, we consider that 6 out of the 13 distinct patterns are properly documented in, at least, one of the analysed works. The remaining 7

Table 3.6: Matching of monitoring practices and design patterns. There are 8 practices without a matching pattern and 4 patterns without a practice counterpart.

Practice	Pattern
Alert Just Enough	
Audit Logging	Audit Logging
	Automated Recovery
Correlation ID	Correlation ID
Classifying Alerts	
Deployment Logging	Deployment Tracking
Distributed Tracing	Distributed Tracing
Exception Tracking	Exception Tracking
	External Monitoring
Liveness Health Endpoint	Health Check API
Readiness Health Endpoint	
Log Aggregation	Log Aggregation
Log Sampling	
Metrics Aggregation	Infrastructure Metrics
	Preemptive Logging
	Query Engine
READS Metrics	Application Metrics
Semantic Monitoring	
Synthetic Testing	
Standard Logging	
Symptom-Based Alerting	

patterns are still suitable pattern candidates for this dissertation. These are: AUDIT LOGGING, DEPLOYMENT TRACKING, DISTRIBUTED TRACING, EXCEPTION TRACKING, LIVENESS HEALTH ENDPOINT (we opted for the name of the practice for this one), INFRASTRUCTURE METRICS, and APPLICATION METRICS.

When adding this information to the answer to LRQ2, we conclude that only 2 of the 9 patterns that formalise a practice have good quality. Therefore, there are only 2 out of 17 practices properly formalised as structured and detailed design patterns. They are CORRELATION ID and LOG AGGREGATION. This proportion seems quite low, so it is also important that this dissertation picks up on existing patterns to detail them more thoroughly.

Adding the 7 patterns that need more development to the 4 practices that need to be formalised as patterns, we end up with 11 candidate patterns. In fact, this is the number of patterns that this

dissertation's pattern catalogue has (see Chapter 5.

LRQ4. To what extent are monitoring practices or patterns empirically validated?

This question is twofold, encompassing both practices and design patterns. We answer each one of these parts in Sections 3.4 and 3.5, respectively. By relating both sections we can conclude that only 3 of the 13 analysed works had proper empirical validation, i.e. validated studies with industry professionals or companies. Even though most of the remaining works are based on the authors' professional experience, practices and patterns that are not validated must be taken with a grain of salt. Regarding practices, we conclude that 10 of the 17 distinct ones outlined in Table 3.3 are empirically validated. When it comes to patterns, only 4 out of the 13 distinct design patterns portrayed in Table 3.5 are validated.

Nonetheless, existing studies show that (1) companies do adopt monitoring practices and patterns, (2) companies adopt practices with variations, (3) log management is the most used practice and log aggregation the most used pattern, and (4) there is a need for more research to solve and then evaluate the solutions for monitoring challenges. All of these are important insights for this dissertation since they manifest that monitoring challenges exist and that design patterns are useful to address these challenges.

Finally, in the answer to LRQ3 we conclude that only 2 practices are formalised as quality patterns, from which only LOG AGGREGATION is empirically validated. Since another goal of this dissertation is to validate the proposed patterns through a case study, the fact that there is so little validation in the literature makes this objective even more relevant.

With a better understanding about the state of monitoring in the Cloud and an established importance of this dissertation, we can now move on to the next section where we further detail the problem, introduce the research questions that guide this work and explain how we will find an answer to them.

Chapter 4

Research strategy

In the previous chapter, existing literature on the topic of this dissertation is analysed. We conclude that there is still some work to be done. This chapter analyses the problem and explains how we seek to tackle it.

In Section 4.1, the problem addressed by this dissertation is further explained. In Section 4.2, the scope and assumptions of this work are defined. In Section 4.3, the research questions that guide this document are presented. To provide an answer to them, in Section 4.4 the methodology chosen for this work is outlined, containing an explanation for every task and the main outputs of the process.

4.1 Problem

This dissertation aims to tackle two issues with monitoring practices: (1) the lack of formality and standardisation of these practices, and (2) the lack of empirical validation to evaluate them and prove their usefulness.

As we explain in Section 1.2, practices are a way to tackle monitoring challenges. So, if they are not reusable from case to case, it becomes harder for companies to adopt monitoring processes, affecting application quality. Furthermore, validating technological theories, such as practices, is important [99]. If no studies prove that a practice is good and helpful to the industry, then its usage should be carefully considered. This also hinders the monitoring process and reduces the value of a practice as a way to tackle monitoring challenges. Finally, in Chapter 3 we explore the existing literature on the topic and conclude that there are research gaps on the formalisation and validation of existing monitoring practices.

Most practices are suggested by grey literature. This fact is not a problem by itself, but the lack of formality that characterises this kind of work results in practices that are not empirically validated. Of the 17 distinct practices we identified in existing literature (see Table 3.3), 8 still need validation.

Finally, existing monitoring design patterns cover 9 of the practices found (see Table 3.6). Thus, there are still 8 practices that need to be formalised. Moreover, of the 9 design patterns that have an underlying practice, we argue that only 2 of them are described in sufficient depth to be used. From these, only 1 is empirically validated. Therefore, existing design patterns are not enough to tackle the identified problems, so there is still much work to address them.

4.2 Scope

This dissertation focus solely on monitoring practices and design patterns for Cloud applications. Another way to tackle monitoring challenges is through dedicated tools. There are plenty of monitoring tools in the industry, and there already is research targeting them [32, 96]. That said, while existing tools may be referenced in the description of the proposed design patterns to make them more complete, they are not in the scope of this work.

Logs, metrics and traces are usually considered the three pillars of monitoring and observability [91]. Alerts, on the other hand, rely on these pillars to help developers and engineers know about issues as soon as possible. Nonetheless, alerts are more of a cross-cutting concern to help with monitoring pillars than an actual necessity to monitor a system. Thus, we keep alerting practices, patterns and dedicated tools out of the scope of this dissertation.

This work assumes that the value of monitoring is established and that companies desire to adopt it for its advantages. We believe there is enough research on the importance of monitoring in the Cloud, either through literature surveys [2, 32] or through industrial studies [96, 54, 83]. Therefore, proving the worth of monitoring and its impact on application quality is out of the scope of this dissertation, even though it is fundamental to the latter's relevance.

This document briefly explores the importance of empirically validating practices and design patterns. We assume that validation of theories is established as an essential step of scientific work, as postulated by Zelkowitz and Wallace [99], so we do not dive into much detail to justify the importance of validating the patterns.

4.3 Research questions

One of this dissertation's goals is to provide a pattern catalogue dedicated to monitoring Cloud applications. Even though patterns should be based on practices, as Kohls and Pahke explain:

> *"Because each pattern contains a network of hypotheses (the forces that link the context form to the solution form), we think it is adequate to consider a pattern as a theory and not only as an isolated hypothesis."* [49, p. 4]

Therefore, the pattern catalogue proposed in this document is a collection of theories. It is, in itself, the hypothesis of this dissertation. In order to verify it, this work seeks an answer to the following research questions (RQs):

RQ1. Which patterns can be defined from current monitoring practices?

As Fowler puts it, "a key part of patterns is that they're rooted in practice " [36, p. 10]. Even if we consider patterns as theories, they are "theories about practice " [49, p. 4]. Created patterns should be based on existing practices. Therefore, we seek to build upon these to propose a pattern catalogue that is detailed and based on what is real.

RQ2. How accurate are the proposed patterns' contexts, forces, problems, solutions and consequences?

Since this document proposes new design patterns, their initial descriptions may not be complete or miss crucial points of their usage. Patterns, just like theories, "can be tested, supported or falsified " [49, p. 3]. Thus, it is crucial to experiment with them on a real-world project and iterate over their descriptions to make the final patterns as accurate as possible and corroborate them.

Note that this question exercises many parts of the patterns and the project to study. The project needs to be characterised and thoroughly contextualised to evaluate if the context and problem match what was initially expected. By going through a semi-structured pattern selection process, we can evaluate how well the pattern's solution, problem and context fit with each other and within the project. Lastly, the forces and consequences involve a more creative task, so these parts can be evaluated by brainstorming with others with knowledge about the project. By answering this research question, we provide an initial evaluation of the design patterns that should demonstrate their usefulness for the industry.

4.4 Methodology

To provide an answer to the RQs presented in the previous section, this dissertation follows the process depicted in Figure 4.1. As a starting point, a literature review is conducted to identify and analyse existing monitoring practices and design patterns. Afterwards, we start the development phase of this dissertation by mining new patterns from the identified practices and further developing design patterns that are not sufficiently detailed. By the end of this stage, an initial pattern catalogue will be available with enough detail to be applied in practice, providing an answer to RQ1. Nonetheless, these patterns still need to be validated empirically.

To that end, we conduct a three-phase empirical study at Konkconsulting, which seeks to answer RQ2. The design of our study leans lightly on other works that seek to empirically evaluate patterns [86, 94] as well as on pattern mining methods, namely the method used by Manns and Rising to teach the fearless change patterns [56].

We start by characterising the project to understand its context, including its architecture, production environment, deployment strategy and team composition. The characterisation encompasses interviews with the team leader and team members and an analysis of the project's source code and documentation. This stage of the study results in a description of the project's context and the identification of its main monitoring challenges.

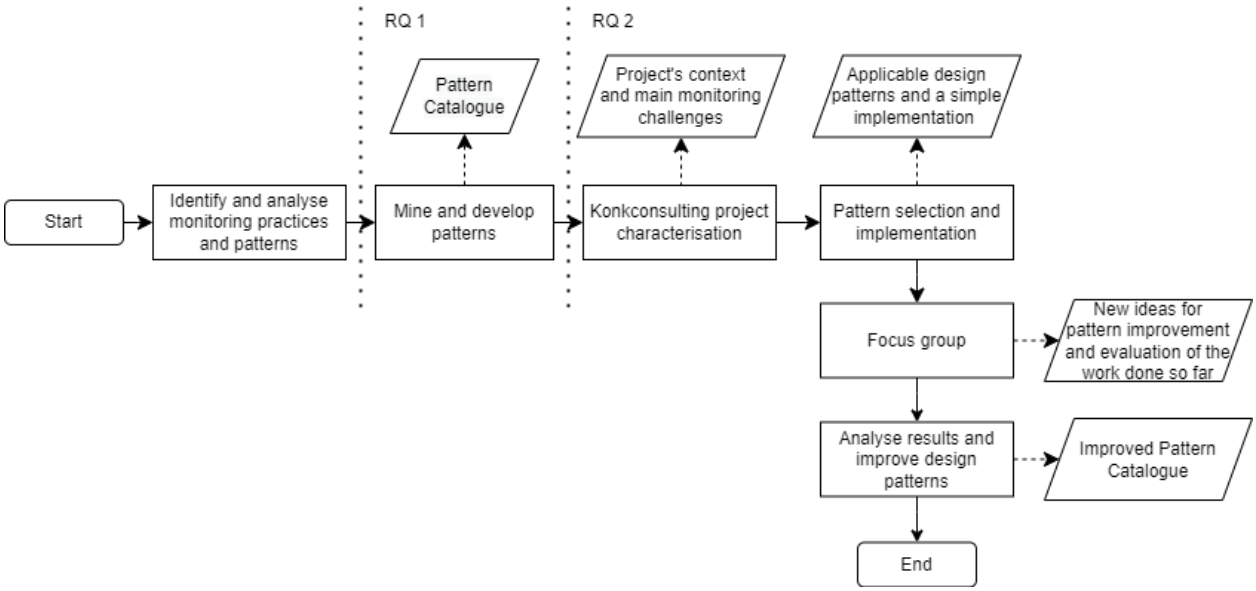


Figure 4.1: Methodology flowchart. Each square is a task of this dissertation and each parallelogram an output of a task.

Afterwards, we analyse these challenges and their underlying problems to identify suitable design patterns to solve them. This is done through a semi-structured pattern selection process in which we compare the patterns' contexts to the project, then verify if the problems we identified match the patterns' problems and check whether the patterns' solutions are feasible and actually solve the problem. This process provides conclusions regarding how the patterns' contexts, problems and solutions fit together and how well described they are. Additionally, we take the selected design patterns and implement them in the project to provide the company with a monitoring stack adequate for the project.

Thirdly, we conduct a focus group with team members to evaluate the work done so far and get new insights into the design patterns' descriptions. These insights result from a brainstorming session around the project's monitoring challenges identified during the previous two phases. We compare the findings from this session with the selected patterns and discuss some implications of their adoption in the project. From this stage, we get new ideas for pattern improvement and evaluate our conclusions from the previous stages of the empirical study.

To conclude our work, we analyse all the data collected throughout this empirical research and improve the pattern catalogue based on this information. The process results in an improved catalogue of what we believe to be helpful and complete design patterns for the industry.

Chapter 5

A catalogue of monitoring design patterns

After analysing existing literature and formulating a research strategy, we are now ready to develop a comprehensive pattern catalogue of monitoring design patterns.

In Section 6.1, we describe this chapter's goals and explain how it seeks an answer to the dissertation's first research question. Afterwards, in Section 6.2, we provide an overall methodology for the pattern mining process and a description of each source used. In Section 5.3, we explain some considerations regarding this dissertation's design patterns. In addition, we expose formatting styles and the pattern structure we use throughout this chapter. From Section 5.4 to 5.14, we describe the patterns with all their sections. Finally, in Section 5.15, we conclude this chapter by commenting on the patterns and providing an answer to the dissertation's first research question.

5.1 Goals

From our work so far, especially the literature review described in Chapter 3, we have concluded that eleven pattern candidates could be written as monitoring design patterns.

In this chapter, we aim to pick the literature review conclusions and elaborate the eleven pattern candidates into full-fledged design patterns. Doing so is not only a relevant contribution to the field but also a way to provide a complete answer to this dissertation's first research question:

RQ1. Which patterns can be defined from current monitoring practices?

The following section explains the methodology used to mine the patterns and answer the above question.

5.2 Methodology

The design patterns proposed in this chapter are based on three different kinds of sources: (1) existing literature, (2) monitoring tools, including libraries and frameworks, and (3) the authors' experience. Our pattern mining process builds upon the conclusions of the literature review process (see Chapter 3), which match our first source of information. The following sections further describe how we utilised each of these sources. We believe that merging them provides a diversified and complete view of each pattern, making the resulting patterns quite detailed.

5.2.1 Literature

The first, and the one from which we mined the most information, is existing literature. We conducted a literature review (see Chapter 3) to understand what monitoring practices exist and which are already formalised as patterns. By comparing different works that describe the same pattern, we get different views over the same solution and become able to mine the underlying pattern.

Therefore, the analysis done in Chapter 3 serves as a consistent base for all the patterns described in the current chapter. To complement this initial research, we analysed other works for each pattern, primarily grey literature. The goal was to fill in some gaps we could not figure out from other sources that were very specific to each pattern. This sources proved especially useful to find real-world examples for the patterns' Known Uses sections.

5.2.2 Monitoring tools

The second pattern source we used was monitoring tools. When practitioners widely adopt the features of some of these tools, they become a sort of best practice. Therefore, we started by identifying commonly utilised features of said tools and then searched for existing literature regarding how those features should be used. We could mine a couple more patterns by compiling this information and understanding how the features work.

5.2.3 Authors' experience

Finally, to complement the two other sources, we resorted to our own experience. Expert knowledge helps identify forces that are hard to extract from existing literature. Thus, this dissertation's supervisor, that has more experience with the topic, actively reviewed the patterns' and helped with the intricacies of each of them.

5.3 About the patterns

In Chapter 3, we identified eleven monitoring practices. They all are suitable pattern candidates that, to our best knowledge, are yet to be formalised as patterns or have incomplete pattern equivalents. After some initial mining, these are their one-line solution statements:

1. AUDIT LOGGING — Record user activity and relevant system changes in a data store “in order to help customer support, ensure compliance and detect suspicious behaviour" [72].

2. STANDARD LOGGING — Implement a consistent logging format across all services and teams, so logs are understood by everyone and consumed by other services independently of their origin

3. LOG SAMPLING — Sample and prioritise logs to reduce the number of logs that need to be stored and processed while maintaining enough information for effective troubleshooting.

4. DISTRIBUTED TRACING — Assign each external request a unique ID and record how it flows through the system from one service to the next in a centralised server that provides visualisation and analysis, making troubleshooting the application faster and less complicated.

5. DEPLOYMENT TRACKING — Track every deployment and change to the production environment [71], making it easy to relate effects observed in the system to the changes that caused them.

6. EXCEPTION TRACKING — Send all exceptions to a centralised exception tracking service that aggregates exceptions, "generates alerts, and manages the resolution of exceptions" [72].

7. LIVENESS HEALTH ENDPOINT — Implement a specialised endpoint that responds to requests without side effects. Then, configure another system (e.g. service, tool, load balancer) to periodically check that endpoint and take action when that fails, providing an automatic way to detect that the instance is unable to respond.

8. READINESS HEALTH ENDPOINT — Implement a specialised endpoint that checks if the service is ready to accept and process traffic. Then, configure another system (e.g. service, tool, load balancer) to periodically check that endpoint and stop routing traffic to the service when the check fails.

9. SYNTHETIC TESTING — Create or pick a subset of existing test cases and periodically run them against the production environment, ensuring the application behaves as expected and detecting issues before they affect end-users.

10. APPLICATION METRICS — Instrument the application to gather business and performance metrics. Collect these metrics in a centralised service that provides aggregation and visualisation, allowing deeper insight into the application’s performance.

11. INFRASTRUCTURE METRICS — Instrument the server and runtimes to capture relevant metrics of the operative system and underlying infrastructure and collect them in a centralised server, allowing the team to get a real-time overview of the application’s environment.

A possible organisation of these patterns and how they can cooperate is portrayed in Figure 5.1. Even though we identify some relationships, we do not think we provide enough guidance on the order in which these patterns should be adopted. Therefore, what we propose in this chapter is not a pattern language; it is a pattern catalogue.

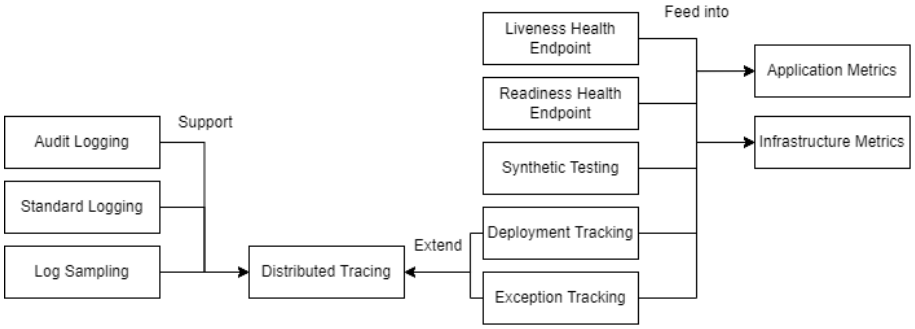


Figure 5.1: Overview of monitoring design pattern candidates.

We believe that writing detailed patterns is important. Patterns are helpful for practitioners of any experience level. While more seasoned professionals can fill in the gaps of detail and extrapolate actionable information, less experienced individuals may find a design pattern hard to grasp if it contains too much domain-specific terminology. Hence, the proposed patterns assume very little about the reader's experience with monitoring and try to explain concepts in detail. Nonetheless, since these are patterns for cloud applications, we expect the reader to be familiar with Cloud computing and its concepts.

Regarding pattern structure, we follow a mix between Gamma et al.'s [38] and Buschman et al.'s [15] pattern descriptions. Thus, we provide the following sections for each pattern:

- **Name** — an intuitive name for the pattern and a summary of its intent.
- **Also Known As** — a list of other names given to the same pattern; in case we have not found any, this section is omitted.
- **Context** — contextualisation of the pattern; provides background on the problem and may also refer to other design patterns that can be considered before the current one.
- **Problem** — a brief description of the problem as a question.
- **Forces** — a list of forces constraining the solution to point in a certain way and not another.
- **Solution** — a complete description of the core solution for the current problem.
- **Consequences** — a bullet-point description of the pattern's main advantages and disadvantages that should be considered when adopting the pattern.
- **Example** — an illustrative example, either real or fictional, of the pattern in action.

- **Known Uses** — succinct description of real-world cases that we know use the pattern; throughout this section, we also briefly mention existing tools that can be used to adopt the pattern.

Lastly, in each of the Solution sections, we highlight certain keywords in **bold**. These represent the central roles of different components or modules that constitute that pattern, similarly to Gamma et al.'s [38] pattern participants. We believe that these roles are effective means of communicating the patterns' solutions. Lastly, by the end of each pattern's Solution, we provide a diagram showing the pattern's different roles and interactions.

5.4 Audit Logging

As more and more people use an application, more problems can arise. Some users will require customer support. Others may try to access the information they are not supposed to see or show suspicious behaviour that the team should be aware of to reinforce existing security measures. Recording the behaviour of users and reproducing it to uncover relevant information is vital in such scenarios. Therefore, the team should record user activity and relevant system changes in a data store "in order to help customer support, ensure compliance and detect suspicious behaviour" [72].

Also known as
Audit Logs [35], Audit Trail [4]

Context
As more and more people use an application, more problems can arise. Some users will require customer support, and a reproducible trace of their actions is helpful to solve the problem. However, sometimes the users cannot provide enough detail for the support to be of any use. Other users may try to access the information they are not supposed to see or show suspicious behaviour that the team should be aware of to reinforce existing security measures.

Being able to reproduce and understand user behaviour can be even more critical when dealing with sensitive data under regulatory policies. Consider, for example, that a care provider in a hospital leaked patient records to the press. In such a case, the hospital administration must be able to discover who did it to take legal action. If the underlying system does not register who did what, then finding out the culprit becomes a nearly impossible task.

Problem
How can the team record the behaviour of users and reproduce it to uncover relevant information?

Forces

- The goal of analysing this information is only known after an incident occurs, so it must be collected preemptively.

- The solution must have minimal temporal and storage overhead in order to avoid user experience degradation.
- If records are to be used as evidence in litigation, they must be stored securely and be immutable.
- Log verbosity must be easily adjustable so the team can somewhat control the number of generated logs.

Solution

Record user activity and relevant system changes in a data store "in order to help customer support, ensure compliance and detect suspicious behaviour" [72].

These records are commonly called **Audit Logs**. They can take many formats, but they are a way to answer "who did what, where and when?" [40]. Therefore, when something important changes, the system should register who caused the change, which action was taken by that agent (either human or not), where was that action applied (i.e. which resource changed) and, finally, when the change took place.

The biggest challenge in adopting this pattern is figuring out the amount of information to register in each log. Logs may be of little to no use in auditing processes if relevant events are missed. On the contrary, if all system events are registered, the system gets slower, and audit logs are much more challenging to process and handle. Therefore, when adopting this pattern, developers should consider which parts of the system need audit logs the most and what events to register. From that point onwards, it is their responsibility to balance the trade-off between data collection and processing cost, aiming for the most comprehensive logs with the most negligible overhead possible.

Adopting this pattern requires a **Log Generator** and a **Data Store** (cf. Figure 5.2). The former is responsible for generating the audit logs and sending them to the latter, which securely stores them for audits, usually following an append-only strategy. There are a few ways to implement this pattern [72] according to how the logs are generated and stored: (1) manually instrumenting the source code, (2) using aspect-oriented programming or (3) using EVENT SOURCING [59].

The simplest option is manually instrumenting the source code to write an actual log. These logs can be stored in a file or folder, which is quite common and straightforward, or saved in a database to be queried later. If developers opt for the former, then a standard format is required. An ASCII string is easily readable by humans and should be enough for a simple structure. For more complex structures, JSON or XML formats can be more convenient. Overall, instrumenting the code is easy and cheap, but Amir-Mohammadian et al. [7] argue that manual instrumentation may miss relevant information.

The second option is aspect-oriented programming (AOP). AOP is a programming paradigm that helps modularise crosscutting concerns (e.g. security and logging) across the application [48]. Audit logs are one of these concerns. When implementing it manually, the logging code must be

replicated in many parts of the program, cluttering the business logic and making it harder to maintain. By implementing audit logs as an aspect of the system, developers can automatically record every method invocation without changing the business logic. However, because AOP usually only provides information about the method name and arguments, it is harder to log business-oriented events [72].

The last option is using the EVENT SOURCING pattern. This pattern "defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store" [71]. Thus, the events become the audit logs, and since they are immutable and the store is append-only, they can be used in legal actions. Developers do not need to do anything else - just by having EVENT SOURCING, they get AUDIT LOGGING. However, implementing it is a significant challenge that requires a change of mindset when dealing with business logic. The EVENT SOURCING pattern has already been explored in existing literature [72, 59], so we do not dive into further detail in this dissertation.

To conclude, every option has its advantages and drawbacks, so the team must choose what is more suitable for their context. Nonetheless, audit logging should be adopted at least in its simplest form as early as possible and not as an afterthought.

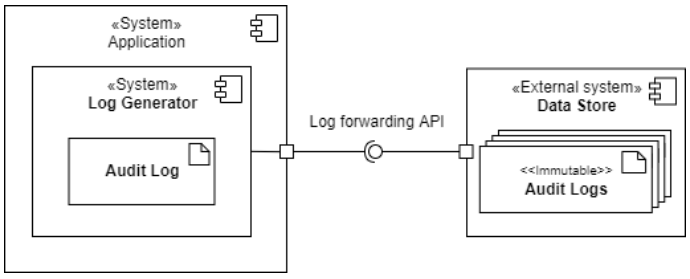


Figure 5.2: Overview of the structure for the AUDIT LOGGING pattern. The stored audit logs should become immutable if they are to be used in auditing processes.

Consequences

This pattern has the following advantages:

- User actions become more accountable than before, so administrators can understand who did what in the case of an incident.
- Audit logs can be used as evidence of regulatory compliance or legal action.
- User activity becomes reproducible, which can be used to fix faults in the code or improve customer support.
- A registry of user actions provides a source for business-driven metrics that may be relevant for managers.
- Audit logs are a straightforward and effective solution for tracking temporal information.

This pattern also has the following drawbacks:

- Audit logs are difficult to process, and tools to do so may require considerable maintenance or infrastructure costs.
- Records must contain enough information *a priori*. Otherwise, when an incident occurs, they are useless. This necessity can lead to records with more information than needed, which leads to massive logs that are harder to analyse and store.
- As time goes by, logs pile up and will occupy more and more space, so they must be carefully managed to control storage overhead.
- Audit logs may contain sensitive information that should be appropriately secured to avoid possible information breaches through the audit data store.
- Since the solution may be a bit complex, its learning curve may create some initial friction with the team.

Example

Amir-Mohammadian et al. [7] provide a real-world example of how audit logs can be helpful in electronic medical record systems. These provide *break the glass policies* so that "care providers can *break the glass* in an emergency situation to temporarily raise their authority and access patient records" [7]. The care providers know that, after they *break the glass*, their "subsequent sensitive operations will be logged and potentially audited" [7].

Suppose that a patient's medical records are leaked. In such a scenario, the hospital must pinpoint the source of the leakage, possibly by knowing who accessed the patient's records during a *break the glass*. "Since it cannot be predicted *a priori* whose information may leak, this goal can be supported by using an audit log that records all reads of sensitive files following glass breaking" [7]. Therefore, these systems must generate audit logs to record "all patient information file reads following a break the glass event, along with the identity of the user that broke the glass" [7]. The actual implementation is not part of the original example. It could be done by instrumenting the code with some logging calls that get activated when the *break the glass* event triggers. These calls would log data into an encrypted file, ensuring that no one would be able to access sensitive information through the logs.

Finally, whenever a leakage would occur, the logged information would be analysed to find all users that read the patient's leaked data. However, "if logging is incomplete then some potential recipients may be missed" [7]. On the other hand, "if logging is overzealous then bloat is possible and audit logs become *write only*" [7], i.e., logs become much harder to process and to store. The authors point out that "these types of errors are common in practice" [7]. Thus, balancing the amount of information in the logs is essential to this pattern.

Known uses

A blog post by Fauna Inc [33] explains how Breezeworks uses audit logs as a feature of their SaaS solution. The post reports that Breezeworks' "end users often want to know when an appointment

or estimate was changed" [33], so Breezeworks needs to store all relevant changes in a record to provide a change history to its users. "They accomplish this by adding an *after_commit* hook to their Ruby on Rails ActiveRecord base model, which maintains a copy of the data in FaunaDB Cloud" [33]. This strategy resembles an AOP approach to implementing the pattern. Whenever a change to a record is committed, the *after_commit* code runs and sends the data to FaunaDB. Hence, the code in the *after_commit* hook acts as the **Log Generator**, and FaunaDB is the **Data Store** where information is kept. This is an example of how audit logs can be used for more than just strict auditing purposes.

Chris Dermody [25], director of product at Flipdish, states that the company uses audit logs to provide customer support. "When a customer calls in and asks why their app seems broken or buggy, being able to track down who changed what, and when, is a powerful tool to have in your arsenal to get the issue resolved as quickly and painlessly as possible" [25]. Unfortunately, the author does not provide more detail on implementing the audit logs.

Finally, a blog post by Ahmed [4], founder of Apptrail, provides some examples of systems that use audit logs, or, as he calls it, audit trails. One of the examples is Google Workspace which the author claims "has a mature audit logs offering" [4]. A screenshot of the platform is shown in Figure 5.3. The dashboard shows the event name, description and time. Some of the events provide information on what changed and which were the old and new values. In addition, the dashboard shows who did the action, i.e. the actor, and the IP address associated with the actor.

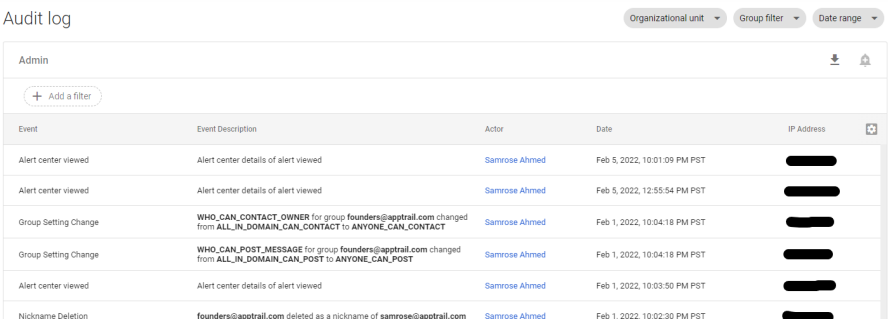


Figure 5.3: Google Workspace's dashboard for audit logs.

Related patterns

As discussed in the solution section, this pattern can be implemented by adopting EVENT SOURCING [59]. Essentially, instead of only saving the current state of resources, the team should keep track of every event that changes them. These events become the audit logs, and actions become more reproducible than before. However, adopting EVENT SOURCING requires a significant change of the business logic and programming mindset, so the overhead might not be worth it for the team.

To query the audit logs, the team should consider adopting QUERY ENGINE [14]. The query system would parse and index the logs' data and make them easier to consume by the team. If the team does not feel the need for such a complex system, adopting the STANDARD LOGGING

pattern may help with the logs' readability without the processing overhead. Lastly, to balance the amount of information and ensure enough detail in the audit logs, the team can consider following the PREEMPTIVE LOGGING pattern.

5.5 Standard Logging

Since different teams and services may generate logs differently, these come in all shapes and sizes. Sorting through a handful of logs may be easy at first, but as the system grows, so does the number of collected logs. The heterogeneity of log formats makes them harder to read and process by humans. In such a scenario, the team needs to improve log readability to make logs easier to read by humans and easier to parse by other services. Therefore, they should implement a consistent logging format across all services and teams, so logs are understood by everyone and consumed by other services independently of their origin.

Also known as
Structured Logging

Context
Virtually every service or application generates logs. These are easy to implement and widely supported [91]. Since different teams and services may generate logs differently, these come in all shapes and sizes. Sorting through a handful of logs may be easy at first, but as the system grows, so does the number of collected logs. The heterogeneity of log formats makes them harder to read and process by humans. Even if the logs are aggregated in a centralised repository (cf. LOG AGGREGATION [83]), it gets to the point that manually reading through the logs is a tiresome and error-prone process.

The team may be considering adopting a QUERY ENGINE [13] to help them extract relevant information from the logs. Even in that scenario, if logs follow different formats depending on the service they are generated from, the query system will take much longer to parse the data. This additional processing effort consumes resources that could be used for other operations more valuable to the system.

Problem
How can the team improve log readability to make logs easier to read by humans and easier to parse by other services?

Forces

- Parsing and consuming the logs must be efficient to avoid impacting the application.
- The solution must be reliable even if the services are written in different programming languages.
- Reformatting logs is a computationally intensive task, so it should be avoided [64].

- Log verbosity must be easily adjustable so the team can somewhat control the number of generated logs.

Solution

Implement a consistent logging format across all services and teams, so logs are understood by everyone and consumed by other services independently of their origin.

As far as we know, there is not yet a common industry standard dedicated to logging [64]. Thus, the team can freely define the logs' structure and format. The **Logging Format** should be configurable in the **Logger**, i.e., the component that generates the **Logs**, be it a method call or a dedicated logging library (cf. Figure 5.4).

Text logs are the simplest to use and work well enough for manual processing. However, processing a JSON or XML formatted log may be easier for other services. Optionally, the logger may be able to dump logs in different formats depending on where they are dumped. For example, it can use text logs when writing logs to the console but a JSON format when dumping them to an endpoint for their consumption by other services.

The information that the logs carry must be defined by the team. They must differentiate mandatory from optional properties and ensure that each property's purpose is easily understandable by everyone. Nonetheless, OpenTelemetry has defined a generic logging format that we believe is a helpful place to start [65]. The following properties are heavily inspired by its format [65]:

- Timestamp — "Time when the event occurred"; ideally follows a standard for date and time such as *ISO-8601* [77].
- TraceId — "Request trace id"; this is an optional field that helps to correlate logs; it is the same for every operation executed during a single request lifetime.
- SpanId — "Request span id"; this is an optional field that helps to correlate logs; it is unique for each operation executed during a request lifetime.
- SeverityText — "The severity text (also known as log level)"; this makes it easier to separate logs according to how severe the event was (e.g. Information, Error, Fatal).
- SeverityNumber — "Numerical value of the severity"; this is an optional field that helps when sorting logs by severity.
- Body — "The body of the log record"; this field can take any format and is the ideal place to insert more details about the event.
- Resource — "Describes the source of the log"; this is an optional field that helps identify the source of the log; it can be, for example, the name of a service, class or machine.
- Attributes — "Additional information about the event"; this is an optional field that, together with the Resource field, helps provide more context to the log.

Finally, the logging format should be adopted across all teams and SERVICES of the system to fully leverage its benefits.

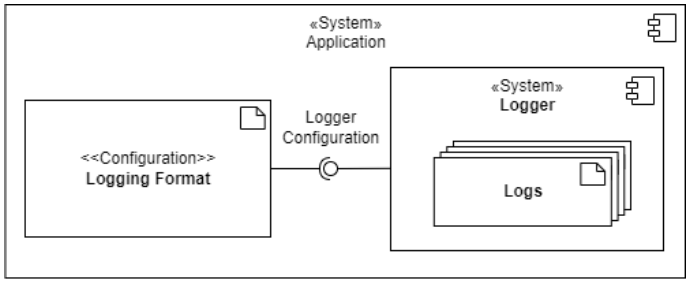


Figure 5.4: Overview of the structure for the STANDARD LOGGING pattern. The logger generates logs based on a configurable logging format.

Consequences

This pattern has the following advantages:

- Logs' readability increases, so the team can extract relevant information from the logs more easily.
- A consistent format ensures that the fundamental properties of logs will be present across all system services.
- Logs become easier to parse and index, reducing the overhead of having other services processing the logs.
- Since logs become faster to process, the system's mean time to resolve (MTTR) is reduced.

This pattern also has the following drawbacks:

- A rigid logging format may not be versatile enough to accommodate the needs of all services and teams. This should not be a problem as long as there are open fields in the format.
- The initial overhead of adopting the standard format may be considerable depending on the number of teams and services that will need to adapt.
- Some logging structures (e.g. binary, XML) are harder to read by humans but better for machines. In that case, the team may consider having two different formats for each use case.

Example

This example is based on another example provided in [64]. There are two services in a system, one for handling orders and the other for payments. They are developed by different teams in different programming languages. On the one hand, the orders team decided to log incoming orders following the format:

```
<TraceID> <Date> <Time> <LogLevel> <ServiceName> <Body>
```

On the other hand, the payments team uses the following format for their service's logs:

```
<ServiceName> <LogLevel> <Date> <Time> <Body> <TraceID>
```

Both teams would like to be able to query their logs, so they decided to implement a query service to ingest the logs from the log files and index their fields for advanced queries. However, they soon realised they needed to configure two different rules to parse each log depending on the service that generated it. Even worse, the name of the service does not appear in the same place in both formats, meaning they would have to come up with some hack to know which parsing rule to use.

Instead, both teams agreed to follow the following logging format:

```
<Date> <Time> <ServiceName> <LogLevel> <TraceID> <Body>
```

Now they can parse and index the logs with a single rule. They do not need to differentiate between the source of logs, and both teams can easily read the logs when they show up together in a log file, as portrayed in Listing 5.1.

```
1 15-02-2020 16:00:58 Order INFO [abc-123] Customer 2112 has placed order 988827
2 15-02-2020 16:01:01 Payment INFO [abc-123] Payment $20.99 for 988827 by cust 2112
```

Listing 5.1: Example of logs from the Orders and Payments services after adopting a common logging format. [64]

So far, finding the main fields of the logs is straightforward — the date is the first column, time is the second and so on. However, it is slightly more challenging to parse the *Customer ID* and *Order ID* from the logs' bodies. The strings used for those fields are easy to read by a person but harder for a machine to parse. In this scenario, both teams could use a JSON object for the body and agree on common property names for the customer and order IDs. It all comes down to the exact needs of both teams regarding query complexity.

Known uses

Waseem et al. conducted an industrial survey with ten different microservices systems from ten different companies [96]. Their study concluded that "the companies widely follow the log content and format defined by the OpenTracing specification" [96]. In the meantime, the OpenTracing specification was archived to make way for OpenTelemetry [22]. Nonetheless, the authors noticed that even the interviewed companies that used customised formats were "migrating to the OpenTracing specification to benefit from its active open-source community" [96]. This tendency reveals that companies adopt a standard logging format and that an industry-standard may be on the horizon.

As part of this dissertation, we analysed and characterised a project from Konkconsulting. Section 6.3 describes the project and our main findings regarding its context and existing monitoring practices. We conclude that the team already adopts STANDARD LOGGING. They use Serilog to keep a consistent format across all their logs. They use text logs that get dumped into separate log files depending on the log's severity level. The logging format is configured in the application's configuration files and is as follows:

```
{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}]
{SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}
```

Finally, in a blog post, Praharaj [69] explains how Grab, a southeast Asia Online to Offline mobile platform, started using structured logging to solve some of its problems. The author states that they used "syslog-style key-value format logs, recognised as the most common format of logs for server-side applications due to its simplicity and ease for reading and writing" [69], for their Golang services. This worked for a long time, but as the system's complexity increased, their "logging bills started mounting to unprecedented levels" [69]. Hence, they changed their logging strategy and bootstrapped a Golang library that, between other features, ensures a consistent log structure. The library output logs in JSON format with a consistent structure that they built "on top of a schema in which [they] can add key-value pairs with a specific key name and type" [69]. Doing so enabled them to use an Elastic stack backend to index and query the system's logs.

Related patterns

This pattern integrates exceptionally well with LOG AGGREGATION. If the logs do not follow a consistent format, aggregating them makes them even harder to read. Thus, the team should consider adopting these two patterns together to help log readability and interchangeability across all other teams.

Moreover, if the team wants to adopt the QUERY ENGINE pattern, they should consider adopting STANDARD LOGGING first. Having a common and well-defined structure across all system services makes implementing a query system much more straightforward, reducing the initial overhead of adopting the QUERY ENGINE pattern.

5.6 Log Sampling

The more services and servers a system has, the more monitoring data it will generate. All this data must be stored and processed to be useful to the team. Logs are part of this data, and the number of logs generated will also increase rapidly as the system grows. The burden to generate, store and process the logs is much bigger in complex systems. The team may need a way to reduce the number of logs generated by the system while maintaining enough information to troubleshoot the application effectively when an issue occurs. Therefore, they should sample and prioritise logs to reduce the number of logs that need to be stored and processed.

Context

Logging is one of the oldest monitoring practices due to its simplicity and broad support. Generating a log is very easy and can be as simple as a piece of text [91]. In addition, "most languages, application frameworks, and libraries come with support for logging" [91]. Therefore, it is safe to assume that almost every service in a system is logging something, one way or another. Logs are indeed quite helpful when troubleshooting an error. The description of the events in the system provides insightful information into its behaviour that is crucial to finding the root cause of failures.

The more services and servers a system has, the more monitoring data it will generate. All this data must be stored and processed to be useful to the team. Logs are part of this data, and the number of logs generated will also increase rapidly as the system grows. Logging excessively can already harm the system's performance [91]. The burden to generate, store and process the logs is much more significant in complex systems. Thus, logging too much introduces an enormous overhead in these systems. On the other hand, too few logs will hinder troubleshooting processes and affect the system's quality. The team must balance the number of logs generated to manage this trade-off.

Problem

How can the team reduce the amount of logs generated by the system while still maintaining enough information to effectively troubleshoot the application when an issue occurs?

Forces

- Collecting every log in a complex system is not feasible due to infrastructure constraints.
- Logs have different severity levels that can be configured in the logger service.
- Collecting an error log without the remaining execution logs usually does not provide enough information for troubleshooting.

Solution

Sample and prioritise logs to reduce the number of logs that need to be stored and processed while maintaining enough information for effective troubleshooting.

"*Sampling* is the technique of picking a small subset of the total population of event logs generated to be processed and stored" [91]. Sampling can be implemented mainly in two ways:

1. Fixed probability sampling — logs are recorded based on fixed probability value [96]; this is easier to implement but does not handle incoming traffic spikes.
2. Adaptive sampling — "dynamically adjusts the probability to ensure that the number of logs produced in a period of time keeps stable" [96]; this is slightly harder to implement but maintains a usual number of logs even when the application is under heavy traffic.

These two methods can be used interchangeably, i.e., the team can use fixed probability for some services and adaptive sampling for others without any problem. Moreover, consider that a **Logger** generates the logs. Afterwards, they are saved on a **Data Store** and are consumed by a **Log Consumer** (e.g. an aggregation service, an indexation service, a human being). In such a scenario, sampling can be classified into two categories: head-based and tail-based sampling. The first implies that sampling is done by the **Logger** as the logs are generated. This cuts on every aspect of the logging overhead but will degrade the correlation between logs (see CORRELATION ID [14]). Tail-based sampling implies that logs are sampled during a preprocessing phase, usually after being aggregated and correlated. This means there is still some overhead with generating and processing most of the logs. However, the sampling process can take into account log correlation and other criteria that were not available when the log was generated. The **Log Preprocessor** service should be the one performing tail-based sampling to prevent data propagation among many services (cf. Figure 5.5). Note that head-based and tail-based sampling can be used together to create a robust sampling strategy.

Finally, sampling can be adjusted based on the log's criteria (e.g. severity level, origin). Sridharan argues that "the efficacy of the sampled dataset is contingent on the chosen keys or features of the dataset based on which the sampling decision is made" [91]. In other words, if the heuristic used to sample the logs is poor, then the resulting log set will also be poor. The team should consider which aspects of the system's logs are more relevant and prioritise the logs based on them. For example, in a tail-based sampling scenario where the logs are correlated with a unique ID, a set of logs for a single request that contains at least one error-level log may bypass sampling altogether. Other possible criteria are the log source, the elapsed time, or custom attributes specific to a service. The team's goal when adopting this pattern is to ensure that the system collects the most important logs and discards the less helpful ones.

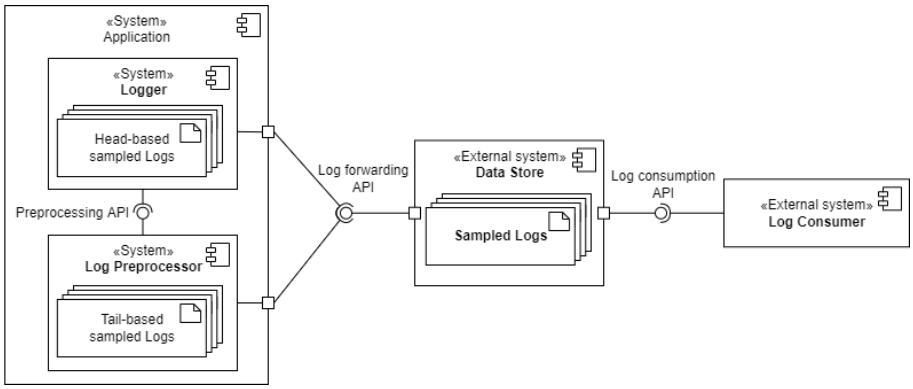


Figure 5.5: Overview of the structure for the LOG SAMPLING pattern. Having both the logger and log preprocessor is not mandatory. The team may use just the one that provides more value to them.

Consequences

This pattern has the following advantages:

- Log overhead is reduced, so generating, storing and processing logs becomes cheaper.
- When using adaptive sampling, the number of logs generated per time unit is maintained somewhat constant.
- Logging remains feasible in complex systems, so the teams do not need to cut on logging practices as the system grows.

This pattern also has the following drawbacks:

- Important logs may be lost during the sampling process, which negatively impacts the team's troubleshooting capabilities.
- Trace analysis usually relies on logs, so sampling them may affect the team's DISTRIBUTED TRACING efforts.
- Implementing adaptive sampling and choosing the appropriate log criteria for sampling may require considerable development effort and costs.

Example

A common usage of log sampling is to control the overhead of DISTRIBUTED TRACING. Since spans are usually generated and saved as logs, sampling the spans mean sampling the logs they are stored in to avoid cluttering the system.

Imagine a system with 1000 microservices and around 1000 requests per second. An incoming request can easily trigger operations in around 100 of those services. In order to trace such a request, there must be 200 generated logs, one for when the request enters a service and another when it leaves. Suppose that the generated logs follow OpenTelemetry's format, like the one in Listing 5.2.

```
{
  "Timestamp": "1586960586000000000",
  "Attributes": {
    "http.status_code": 500,
    "http.url": "http://example.com",
    "my.custom.application.tag": "hello",
  },
  "Resource": {
    "service.name": "donut_shop",
    "service.version": "2.0.0",
    "k8s.pod.uid": "1138528c-c36e-11e9-a1a7-42010a800198",
  },
  "TraceId": "f4dbb3edd765f620",
  "SpanId": "43222c2d51a7abe3",
  "SeverityText": "INFO",
  "SeverityNumber": 9,
  "Body": "20200415T072306-0700 INFO I like donuts"
```

```
18 }
```

Listing 5.2: Example of a trace log that follows the OpenTelemetry standard.

A *.txt* file with the example's content occupies around 0.5 kB. So, for every request, we need to store approximately 100 kB. With 1000 requests per second, the number grows to 100 MB. After one day, the system would have generated 8640 GB of logs. The data provided in this example may sound slightly far-fetched. However, considering that Netflix's API handles around 20000 requests per second [75] and that the three largest companies from the study in [96] had more than 1000 microservices, it is not as exaggerated as it may have looked initially.

The main takeaway is that logging storage becomes an issue very quickly. If we adopt an adaptive sampling strategy by configuring our log library to ensure a sampling rate of around five logged requests per second, at the end of the day, we only have to store 43.2 GB of logs. By adjusting the sampling rate, we can configure how many bytes of data we generate per day. To conclude, note that this is just an issue for complex and high traffic systems, like the one we portrayed in this example. The ideal scenario is to log everything; if the team can afford to do so, they should not use LOG SAMPLING.

Known uses

Waseem et al. conducted an industrial survey with ten different microservices systems from ten different companies [96]. According to their results, four companies used sampling as part of their tracing system. All four companies used head-based sampling and, depending on the service, either fixed probability or adaptive sampling. These companies had the most complex systems in the study, each with thousands of microservices to maintain. The authors concluded that "sampling can alleviate the overhead to services and log storage while losing important traces for trace analysis tasks such as root cause analysis " [96]. Nonetheless, in very complex systems, sampling remains a helpful method to control log overhead.

Sigelman et al. [82] wrote a white paper explaining the design of Dapper, "Google's production distributed systems tracing infrastructure " [82]. The tool was developed internally to tackle Google's production challenges. As one can imagine, problems at Google's scale demand unique solutions to handle the enormous amount of data generated. One of Dapper's design goals was low overhead. The authors concluded that sampling was the best way to achieve that goal [82]. The authors started by using a fixed probability strategy to collect approximately one sampled trace in 1024 candidates. This worked well for high-throughput services, but the number of traces collected per time unit was low for less loaded services. Therefore, the team implemented adaptive sampling in Dapper to ensure that the number of traces sampled per time unit was nearly constant. This tool reveals how large and complex systems like Google's eventually need to sample their data.

Finally, according to a post in Uber's engineering blog [81], the company explains how Jaeger, an open-source tool it created, supports sampling. According to Shkuro, the post's author, "most production tracing systems, especially those that have to deal with the scale of Uber, do not profile

every single trace or record it in storage " [81]. Thus, they were forced to sample the trace data to be able to cope with the storage space required. Moreover, Uber created a feature for Jaeger, a polling feature in the tool's client libraries that allows the to poll for the sampling strategy to use. This allows the system's "backend to dynamically adjust the sampling rates as the traffic patterns change " [81].

Related patterns

This pattern and PREEMPTIVE LOGGING [83] have a slightly different goal, but both strive for having the maximum information possible within the system's resource capacity. Therefore, PREEMPTIVE LOGGING can be viewed as an alternative to the current pattern for less complex systems. However, for more complex systems, LOG SAMPLING can be used to handle any excessive logging that has not been tweaked after the adoption of PREEMPTIVE LOGGING.

If the team is adopting CORRELATION ID [13], they should consider using tail-based sampling to avoid losing the correlation between log entries. Moreover, if they already use LOG AGGREGATION [83], then the aggregation service can take the role of **Log Preprocessor** and be responsible for sampling the aggregated logs.

5.7 Distributed Tracing

Cloud applications are usually a sum of modules that work together to reply to user requests. Therefore, a request may require many nested operations throughout the application's modules, both internal and external. When something breaks during these operations, it can become hard to figure out exactly where the failure is. The team should be able to record and visualise the end-to-end behaviour of the application to identify problems during the request lifetime. Therefore, they can assign each external request a unique ID and record how it flows through the system from one service to the next in a centralised server that provides visualisation and analysis, making troubleshooting the application faster and less complicated.

Context

Cloud applications are usually a sum of modules that work together to reply to user requests. Therefore, a request may require many nested operations throughout the application's modules, both internal and external. When something breaks during these operations, it can become hard to figure out exactly where the failure is. Moreover, the person troubleshooting the error may not even be familiar with all the modules, making the troubleshooting process longer and more complicated.

Sometimes, even if the system did not fail, it might still be underperforming. If a user needs to wait five or more minutes for a webpage to load, there might be an operation (among all executed for that request) that is taking too long. The cause of slowness may be hard to pinpoint when requests are complex and involve many operations.

Problem

How can developers record and visualise the end-to-end behaviour of the application to identify problems during the request lifetime?

Forces

- Monitoring metrics usually provide an aggregate view of the system (e.g. max response time in the last 15 minutes, the sum of request errors in the last hour), so they are not helpful in monitoring a single request.
- Log entries of a specific request, even when correlated through a unique ID, are stored across different locations on many individual logs.
- Logs usually record events or errors. Thus it is harder to visualise how much time each operation took to identify bottlenecks.
- Even when running outside the system's production environment, unit and synthetic tests may miss temporary events that impact the user (e.g. high CPU or RAM usage, network issues in the cloud infrastructure). Hence, developers need information from when the system processes the request to identify potential causes of failures.
- Any solution should impact the system's performance and maintainability as little as possible.

Solution

Assign each external request a unique ID and record how it flows through the system from one service to the next in a centralised server that provides visualisation and analysis, making troubleshooting the application faster and less complicated.

One way to achieve this is by instrumenting the source code as follows:

- An **Unique ID** is assigned to each incoming request (cf. CORRELATION ID [13]);
- The ID is sent to every **Service** that processes the request;
- Local thread activity is recorded in a **Span** of execution and tagged with the unique ID;
- The unique ID is included in all the logs generated while processing the request;
- The span information is sent to a central **Tracing Service** directly from the microservice or through a local **Forwarding Agent** (cf. Figure 5.6).

The collector is then responsible for processing all spans and constructing **Execution Traces** by mashing together the related spans and logs. Usually, the collector also provides a UI to visualise request information and the ability to query the traces. The collector lets developers analyse the whole execution trace and quickly figure out performance bottlenecks, which operation originated a failure and the logs associated with the previous situations.

When recording local thread activity in the spans, developers should consider following the OpenTelemetry API. This standard is based on the existing OpenTracing and OpenConsensus APIs and is widely used by the industry [64]. In addition, many existing libraries and tools already come with support for OpenTelemetry, making it much easier to adopt Distributed Tracing.

The biggest challenge when adopting this pattern is handling the information collected. The system will be impacted if the developing team collects and stores traces for every request. A possible solution to this issue is tail-based sampling, i.e. sampling the execution traces after they have been reconstructed in the collector service. The team can decide to drop some of the traces and only store the remaining ones, effectively reducing the overhead on the system. "The idea here is to capture enough information to understand what our system is doing but not capture so much information that the system itself cannot cope" [64]. The sampling can be random or dynamic. The former consists of randomly selecting which traces to sample, increasing the likelihood of losing relevant information or collecting insufficient information to reach relevant conclusions. The latter has different sampling rates that depend on certain conditions (e.g. if there was an error during execution or if a given operation has been slower than usual). Dynamic sampling is harder to implement and requires a bit more processing but is more likely to catch the information needed to troubleshoot the application. Balancing this trade-off is the responsibility of the team when adopting this pattern.

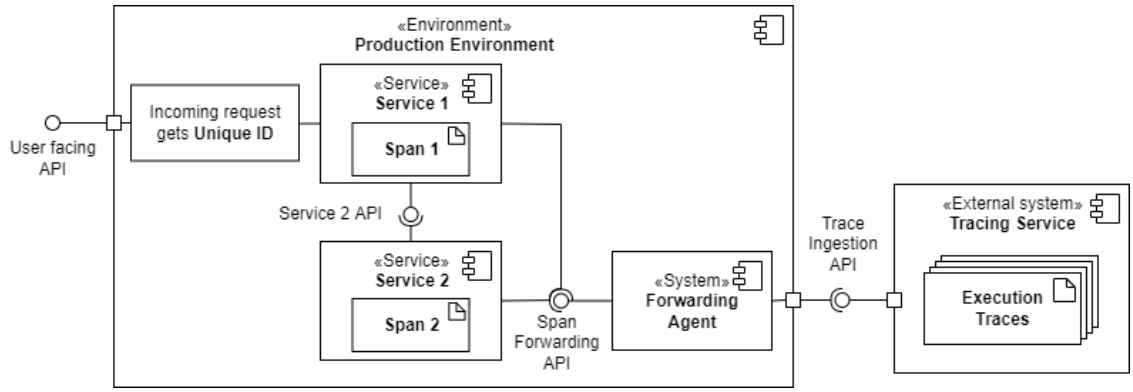


Figure 5.6: Overview of the structure for the DISTRIBUTED TRACING pattern. The request is assigned a unique ID as it enters the system and the generated spans carry that ID.

Consequences

This pattern has the following advantages:

- Mean time to detect (MTTD) becomes shorter, making troubleshooting the system faster.
- User requests become observable from end to end, i.e. every operation performed since the request first arrives in the system until a response is sent to the user is tracked.
- Traces provide insight into individual operations, making it easier to identify system bottlenecks.

- Existing commercial solutions require little source code instrumentation and introduce very little overhead to the system.

This pattern also has the following drawbacks:

- Depending on the programming language used, implementing this pattern may require changes, sometimes complex, in the source code.
- Recording and aggregating traces introduces some overhead in the system and can impact its performance depending on the number of concurrent requests the application serves.
- Traces need to be stored, and as time goes by, they will start to occupy more and more space, demanding considerable infrastructure and increasing the storage costs.
- Existing commercial solutions are usually expensive; adopting an open-source solution and adapting it to the project may be cheaper but also introduce a larger overhead on the system.
- Since the solution is a bit complex, its learning curve may create some initial friction with the team.

Example

Meesho is an online shopping app for India. A blog post by Agarwal [3] explains that, as they transitioned from a monolithic architecture to a microservices-based framework, understanding the path of a single request became much more complicated. Because "a single request passes through multiple services" [3], tracing a request across all services became a challenge. The author explains in further detail their feed system's architecture. Most of their services composing the feed system were "Spring framework-based Java microservices" [3]. It followed a layered architecture, "where each layer has its responsibility and each domain service operates within its boundary context" [3]. The post had images to complement this description, but unfortunately, the post's images are no longer available.

To tackle the abovementioned issue, the team decided to implement distributed tracing in their system through Spring Cloud Sleuth. The author states that the decision was based on the tool's "auto-configuration capability and compatibility with other Spring libraries" [3]. The distributed tracing works as follows:

- The unique ID is assigned by the first service that does not find it in the request headers. They use two unique IDs for each request — the *Trace ID* and the *Span ID* — and the service attaches them to **ThreadLocal**, a Java class that provides thread-local variables.
- The team "implemented an interceptor that passes these IDs downstream as headers" [3].
- To include the unique IDs in the logs, the team used slf4j to fetch the IDs from the *ThreadLocal* variables and include it in the logs.

They bumped into a problem with asynchronous processing that led to the loss of both unique IDs because the thread context changed in the middle of the execution. They wrote their code to copy the variables from one thread's context to the other to solve this issue. With that out of the way, Sleuth handles the aggregation and correlation of spans to construct the execution traces.

The author ends by stating, "we obtained Trace ID from User ID in one of our view layer services, then queried our centralised logging system. This showed us the whole request trace across services and we instantly figured out that the A/B service was returning an unexpected response in one of our domain services which caused that issue" [3].

Known uses

Agarwal's [3] example described in the previous section is already a known use of this pattern that utilised Spring Cloud Sleuth to implement this pattern in a real-world microservices architecture.

FloQast is another company that uses distributed tracing to understand its application. In a blog post, Dinh [27] explains that FloQast is a fast-growing company, so they "add more and more business logic to the codebase every single day" [27]. The team noticed that bottlenecks started to appear in the system, but they did not have enough information to follow a data-driven decision. Thus, they started using AWS X-Ray service, which, as the author explains, "allows users to gain insight into requests that your application serves" [27]. Using that tool, they can "easily trace requests across AWS resources and other microservices" [27].

Finally, and on a much bigger scale, Netflix built its distributed tracing infrastructure to solve user issues better. In Netflix Technology Blog, Pandey [67] goes into some detail about how they did and evolved the infrastructure to cope with emerging standards like Open-Zipkin and Open-Tracing. "Investigating a video streaming failure consists of inspecting all aspects of a member account" [67], so troubleshooting a user complaint can become very complicated on a system as complex as Netflix. They developed Edgar, an internal tool for troubleshooting streaming sessions. They started with a simple tool that, based on unique IDs for each streaming session, was able to "reconstruct session failure by providing service topology, retry and error tags, and latency measurements for all service calls" [67]. For their case in specific, the team had to apply a hybrid head-based sampling approach that enabled them "to record 100% traces in our mission-critical streaming microservices while collecting minimal traces from auxiliary systems like offline batch data processing" [67]. Even though this helped reduce the number of traces stored, the author states that they still had issues escalating their ElasticSearch clusters. So they eventually transitioned to Cassandra clusters to handle the high data ingestion rates.

Related patterns

This pattern somewhat encompasses the CORRELATION ID [13] pattern. In other words, to get DISTRIBUTED TRACING, the team must implement CORRELATION ID. Therefore, if that is not enough, the team can start with the latter to tackle the most basic correlation needs and move on to the former.

Logs are a common way to implement execution spans. In such cases, adopting LOG AGGREGATION becomes almost necessary to get DISTRIBUTED TRACING. In addition, logs can also be correlated and aggregated with the traces of execution. This provides the team with a complete report of the events and the time each operation took across the whole request lifetime.

However, generating too many traces can carry a very high overhead. The team should consider the LOG SAMPLING pattern to implement a sampling strategy that minimises the overhead of logged traces while maintaining the ones that truly matter for troubleshooting purposes.

5.8 Deployment Tracking

Nowadays, code gets released faster and faster to meet customer demands, so system changes can be deployed many times a day. By introducing new features or tweaking existing ones, there is the risk that errors will start showing up and, even with a robust suite of tests, faults can make it into production, possibly degrading some quality attributes (e.g. performance, security). When metrics skew up or down, indicating system anomalies, the team should know where to start looking for a cause. Therefore, the team must track every deployment and change to the production environment, making it easy to relate effects observed in the system to the changes that caused them.

Also known as

Log deployments and changes [71], Release tracking [12]

Context

Nowadays, code gets released faster and faster to meet customer demands, so system changes can be deployed many times a day. Every change that gets released is an opportunity for failure. By introducing new features or tweaking existing ones, there is the risk that errors will start showing up and, even with a robust suite of tests, faults can make it into production, possibly degrading some quality attributes (e.g. performance, security).

By collecting metrics, engineers can notice when there is something wrong with the system. However, metrics represent effects and not root causes. In other words, they denote what failures or warnings are occurring in the system but do not correlate these to possible disruption causes.

Problem

When metrics skew up or down, indicating system anomalies, how the team know where to start looking for a cause?

Forces

- Issues in the system usually occur after changes, software-wise or not, hence the interest in tracking them.
- Changes affect end-users from both the technical (e.g. slow requests) and business (e.g. disappearing purchase history) point of view.

- Software release cycles can vary a lot, so any correlation strategy must be able to cope with just a few releases per year or many releases per day.
- Reusable solutions are more desirable as they help mitigate development and infrastructure costs.

Solution

Track every deployment and change to the production environment [71], making it easy to relate effects observed in the system to the changes that caused them.

"Different companies [should] track change in ways that are reflective of their release cycle" [12]. So companies that release less frequently (e.g. once or twice a year) could make this a manual process, while companies that deploy more frequently (e.g. more than once per day) must automate this task into their **Deployment Tool** or pipeline. Nevertheless, automating deployment tracking handles both cases and eliminates possible human errors. Hence, it can be helpful for companies with infrequent releases as well.

By plotting deployments and changes together with existing metrics graphs, it becomes trivial to understand if a release caused the metrics' to skew up or down and, if so, at what time that happened. With this increase in visualisation, the team can detect the source of a failure and solve it much faster. Tracking Deployments can be achieved in many ways. **Deployment Records** can be logged or represented as events of the system and have an associated event handler to register them. However, to get the deployments shown in the metrics' dashboards, the best option is to directly insert the deployments as a binary metric into the **Metrics Service** (cf. Figure 5.7).

Moreover, tracking deployments and changes and comparing them to business-related metrics can provide insight into how releases affect the end-users. Request rates or daily logins may increase if a major feature is introduced in a new system version. By plotting releases and business-oriented metrics together, managers can also benefit from the increased observability of the system.

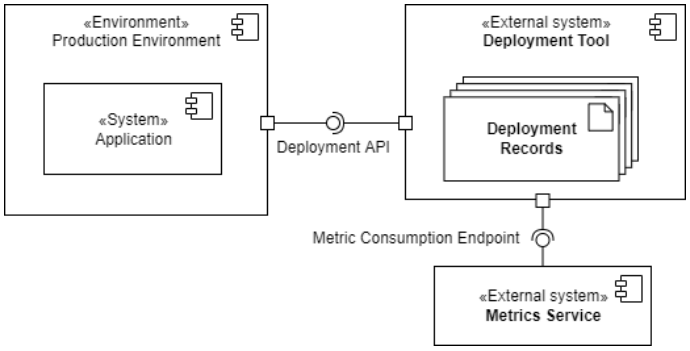


Figure 5.7: Overview of the structure for the DEPLOYMENY TRACKING pattern. The production environment and application are shown just for better context.

Consequences

This pattern has the following advantages:

- Faulty versions of the system become easier and faster to detect, so the meantime to detect (MTTD) gets shorter.
- Releases can be correlated with other metrics, encouraging data-driven decision-making.
- When deployments are plotted with metrics, visualisation becomes richer, and a better system overview is made available to the stakeholders.
- Teams can have more confidence when releasing a piece of software because, even if faults make it into production, they will be noticed quickly.

This pattern also has the following drawbacks:

- To take full advantage of this pattern's benefits, metrics should be already collected and visualised; implementing all of this from the beginning requires considerable time and money.
- Plotting a binary metric, such as deployment logs, may not be supported by all tools; the team should ensure their **Metrics Service** supports drawing the releases as vertical lines.

Example

In an article written in Etsy's *Code as Craft* blog, Mike Brittain [12] explains how the company decided to adopt deployment tracking. Since they were already collecting many system metrics, they could correlate them and understand the effects of a change. Figure 5.8 shows an example of an unusual increase in PHP warnings on the system.

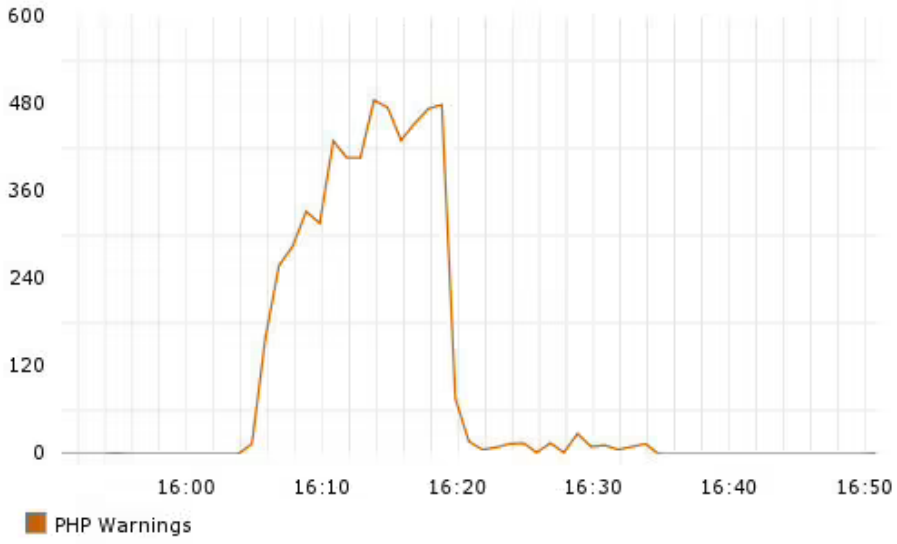


Figure 5.8: Plot of the number of PHP warnings over time for Etsy's system.

However, they were missing the actual cause of the change - the deployments. Since issues in the system usually occur after changes, they started tracking deployments. To do so, they tweaked their code deployment tool to emit a deployment event every time a new release was made. The event consisted simply of a name field (*events.deploy.website*), a placeholder value (*1*,

for example) and the timestamp (in their case, *1287106599* in Unix time). By sending this metric to their monitoring tool, Graphite, they plot the changes together with the metrics, as depicted in Figure 5.9.

Figure 5.9: Plot of the number of PHP warnings over time and code deployments for Etsy's system.

Given the new information, it became clear that the warnings were due to a release at 16h, and they were fixed with the two subsequent releases. The ability to visualise the changes makes it very easy to identify release trends. This example was focused on a technical metric. However, Brittain also provides an example of how the number of new posts in Etsy's forums increased after a product launch (see Figure 5.10), revealing how tracking releases can also be helpful for business managers and customer support.

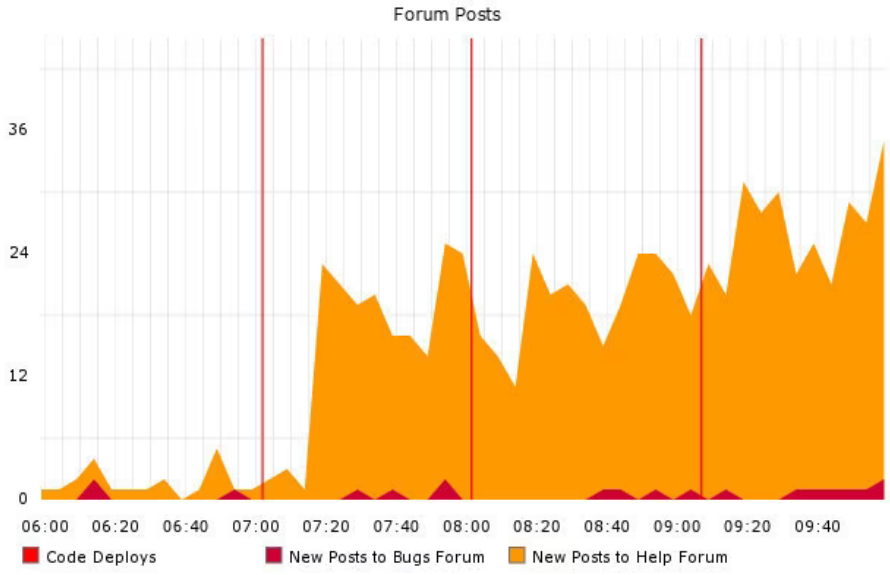


Figure 5.10: Plot of the number of posts in Etsy's forums and code deployments.

Known uses

Brittain's [12] example described in the previous section is already a known use of this pattern that utilised Graphite to implement this pattern in a real-world system.

In a blog post, Will Sewell, a backend engineer at Monzo, explains how they deploy to production more than 100 times a day [80]. The author goes into detail to explain many things that contribute to the high frequency of deployments, including company culture and used tools. We focused on the fact that they track the number of deployments they do each day, which allows the company to plot, for example, the average weekly deployments per engineer. Moreover, their "deployment pipeline records events of what happened, and when " [80], so they can "easily find out why things went wrong and stop it happening again" [80].

Finally, in a blog post about Raygun's Deployments feature, Freyja [37] reports that they interviewed some customers to understand how they use the feature to track deployments and version releases. According to the author, most teams find it "most helpful when investigating which errors have been introduced with the new release, which are still occurring, and which have been fixed " [37]. The tool provides visibility over the deployment pipeline so teams can avoid error propagation across the deployments. The author recommends configuring "Raygun with deployments so that you can correlate error spikes with releases " [37], which greatly resembles this pattern's solution.

Related patterns

Deployments and changes can be treated as events and monitored that way, so this pattern follows an event-based monitoring strategy. Similarly, deployments can be treated as binary metrics that are collected and correlated with other system metrics. Without metrics that illustrate system anomalies, the usefulness of DEPLOYMENT TRACKING is much lower. Hence, the team should consider adopting INFRASTRUCTURE METRICS, APPLICATION METRICS, or both before implementing this pattern.

Nevertheless, if the visualisation or collection of metrics is not yet developed in the project, merging the deployment logs with other system logs by adopting LOG AGGREGATION [83] may already be enough to correlate deployments with system anomalies. Logs are the simplest way to implement this pattern and may provide enough value for teams that do not have time or money to invest in more robust monitoring solutions.

5.9 Exception Tracking

Logging exceptions is the most basic way of recording an error. As the exception occurs in the system, it is sent to the log files just like any other event. This practice is helpful as a long-term historical record of what happened with the system. However, it does not support troubleshooting so well. Logging lacks fundamental capabilities (e.g. de-duplication, issue-tracking) that make exceptions much easier to use for troubleshooting. The team needs a way to efficiently record exceptions to maximise their value for troubleshooting processes. Therefore, they should send

all exceptions to a centralised exception tracking service that aggregates exceptions, "generates alerts, and manages the resolution of exceptions" [72].

Also known as
Error Monitoring [92], Error Tracking

Context
Logging exceptions is the most basic way of recording an error. As the exception occurs in the system, it is sent to the log files just like any other event. Most of the time, the log's severity level is adjusted, so it is easier to identify these erroneous events. This practice is helpful as a long-term historical record of what happened with the system. However, it does not support troubleshooting so well. Consider mobile and desktop users that prefer dedicated applications. These will also generate exceptions, but logging to a local file in such a scenario provides little to no use for the team.

An exception's stack trace is vital to the team for fixing errors. On the one hand, if they use logs as single lines of text, stack traces do not fit in the log's body, and crucial information may be lost. On the other hand, if the stack trace is dumped into the log, it most likely clutters the log files with long lists of method calls. Furthermore, logging lacks fundamental capabilities (e.g. de-duplication, issue-tracking) that make exceptions much easier to use for troubleshooting.

Problem
How can the team efficiently record exceptions to maximise their value for troubleshooting processes?

Forces

- Logging does not handle duplication of exceptions.
- Logging exceptions does not provide an issue-tracking mechanism [72].
- Logging exceptions in the middle of other non-error events makes them harder to read and utilise.

Solution
Send all exceptions to a centralised exception tracking service that aggregates exceptions, "generates alerts, and manages the resolution of exceptions" [72].

The **Exception Tracking Service** handles all major concerns with error tracking. First, it handles all **Exceptions** from production systems, both on the web and native applications. The team should instrument the source code using an **Exception Tracking Library** to forward every exception that gets thrown in the **Application** to the tracking service (cf. Figure 5.11). The library takes care of the details of getting the exception from the running environment, so it works for all platforms alike. Moreover, it is common for the library to be available in many languages to support various sources.

Secondly, the tracking service handles duplication of errors. Since exceptions are usually thrown and sent upstream, they commonly appear duplicated. Thus, the service should either drop duplicate exceptions or aggregate them in a single report that shows many similar occurrences. The latter provides more information for troubleshooting but also requires more storage space and processing effort. Having a single exception case stored may be representative enough for troubleshooting. Hence, the team should start with the simplest scenario and only go for the more complex one if needed.

Additionally, the team should collect as much context as possible when the exception occurs. This is usually done by the instrumentation library because it is running in the production systems. There are no mandatory properties that every exception tracking system should collect, so the team should discuss what things they want to include in an exception report. These may involve information about the source of the exception (e.g. the name of the service, its IP address), some additional specifications of the user's machine, a record of the user actions some moments before the exception occurred, source code snippets from the calls that originated and propagated the exception, and a set of logs generated close to when the exception was thrown. These are merely ideas, but all provide additional views of the system to make the troubleshooting process as straightforward as possible.

Finally, the exception tracking service should provide alerts and automatic issue-tracking. Even though alerts are out of this dissertation's scope, every exception is likely a critical error. Thus, it is vital to have alerts for this level of importance. As is typical with alerts, the way they are delivered, their thresholds and descriptions should be configurable. The issue-tracking is not a must-have in the tracking service, but it does prove to be a valuable feature. The exception tracking service can make the whole troubleshooting process faster and more organised by automatically creating an issue and associating the error report with it. Depending on the team's needs, the issue management can be internal to the service or a third-party tool.

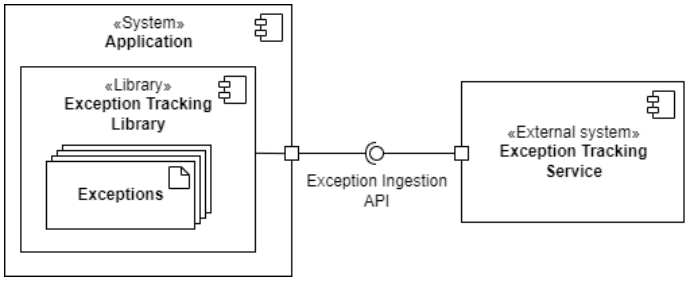


Figure 5.11: Overview of the structure for the EXCEPTION TRACKING pattern. As exceptions occur, the library exports them to the exception tracking service.

Consequences

This pattern has the following advantages:

- Exceptions are aggregated and stored in a single repository, easily accessible by the team.

- Collecting exceptions from different platforms and devices becomes trivial and does not require additional tooling.
- Aggregating similar exceptions reduces overall noise and makes them easier to process.
- Troubleshooting errors becomes faster, reducing the system's mean time to repair (MTTR) and overall number of errors.
- When a record of user actions is included in the error report, the team can troubleshoot the exception without needing to exchange emails with the users.

This pattern also has the following drawbacks:

- Instrumenting the source code to catch all exceptions and call library functions makes it more complex.
- The exception tracking service, either self-hosted or as a managed SaaS, introduces new costs in the system.
- Storing the exceptions and their contexts may require additional infrastructure.

Example

Suppose we are developing a platform available on the web and native mobile applications for Android and iOS. All frontend applications use the same backend services. We naturally start logging exceptions to log files and soon realise that this strategy only works for our backend services. Exceptions that occur solely on the front are not showing up in the log files.

For example, misconfigured navigational routes on mobile applications that receive a null value generate a NullPointerException and crash the user's application. These exceptions dramatically degrade the user experience and should be fixed as soon as possible. However, we cannot notice the problem with our current error tracking strategy.

Therefore, we decided to use Sentry due to its integration with Android and iOS and a JavaScript library for the web application frontend code. "Sentry's SDKs report an error automatically whenever a thrown error or exception goes uncaught" [78] in the application. Even if the tool cannot send the exception report immediately due to application instability or network issues, "the report is guaranteed to send once the application is started again" [78].

With the tool set up, we now have an overview of incoming exceptions from our frontends (see the Figure 5.12 for an example). Not only did we get visibility over a significant section of our user base, but we also improved our ability to troubleshoot exceptions and improve the overall quality of the application.

Known uses

Gerhard Jacobs, in a post on Sentry's blog [46], explains how monday.com uses that tool to monitor their errors. The author puts great emphasis on the importance of customising the tool. He states that monday.com used Breadcrumbs, a Sentry's feature that "[shows] a timeline of actions

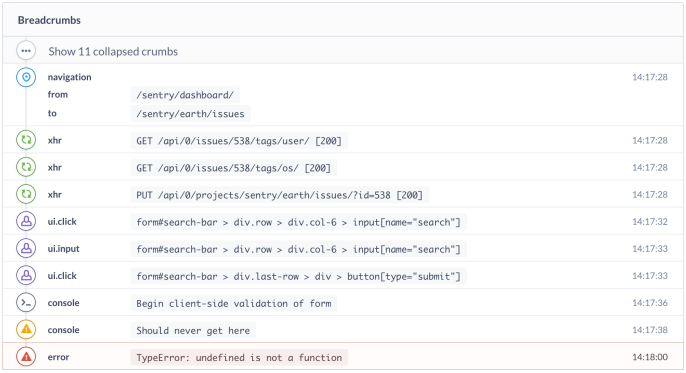


Figure 5.12: Example Sentry dashboard with a visualisation of events before an exception using the tool's *Breadcrumbs* feature. [73]

that led to an error, reducing the time required to resolve it" [46], and "added their own custom events, allowing for more granular investigations" [46]. This feature reduces the need for user feedback since most information is already conveyed in the collected events. According to the author, monday.com's team was able to "reduce the time it takes to resolve an issue from between 30-45 minutes to 10 minutes" [46] using exception tracking.

In a post on Raygun's blog, Penney [68] writes about a presentation he did explaining how Raygun, the company, uses Raygun, the tool, to monitor its deployments. The author reveals that their deployables "[feed] into separate Raygun apps" [68], and developers can subscribe only to the notifications from projects they are involved in. The tool also provides a Slack integration to notify the team of occurring issues. The author claims he uses the error details page to triage and assign issues and a feature called *User Tracking* to prioritise issues [68]. Managing their systems so closely allows Raygun's teams to provide quick customer support and retain customers.

Finally, in another blog post, Borders [9], at that time working at Collage.com, explains some of the company's problems when scaling exception tracking. They were using TrackJS as an exception tracking tool, and the teams would fix some of the tracked exceptions each sprint. However, the author states that it quickly became evident that fixing everything was unfeasible, and the company struggled with keeping track of the priority of each exception. Thus, they "created a system to synchronise exceptions with Jira tickets. This system creates Jira tickets when new exceptions pop up but also synchronises the number of affected users (or total count for back-end exceptions without a user ID) during the past 24 hours and adjusts the ticket priority on a continuous basis" [9]. Additionally, they implemented alerts to trigger when a ticket reached maximum priority, ensuring that one of the on-call engineers would be on top of the occurrence.

Related patterns

Adopting DISTRIBUTED TRACING along with this pattern provides an even more complete view of the events before an exception. The team can follow the execution traces, analyse the information inside each span as contextual information, and have the exception tracking service complement that with the error's stack trace and additional context.

Even though logs make troubleshooting exceptions challenging, it is still relevant to keep logging exceptions as historical long-term storage of system events [79]. Thus, the team may consider adopting LOG AGGREGATION [83] in case they need to keep these events stored (e.g. for audits).

5.10 Liveness Health Endpoint

Once a module or service of an application is deployed and released, the load balancer can start routing traffic to that instance. When a service unexpectedly becomes inoperative or extensively degraded, it can no longer process user requests. The team needs to notice these cases as soon as possible. Therefore, they should implement a specialised endpoint that responds to requests without side effects and configure another system to periodically check that endpoint, providing an automatic way to detect that the instance cannot respond.

Context

Once a module or service of an application is deployed and released, the load balancer can start routing traffic to that instance, which will serve the user requests. Imagine a fault in the code that causes the instance to crash when a specific request is processed or that the host machine, which is not even the team's responsibility, has a very high CPU and memory usage, making the service very slow. These cases are examples of how a module or service of an application may suddenly crash or get extremely slow, unable to respond to user requests.

If the load balancer can not notice when that happens, requests routed to the failed instance will fail or timeout, impacting the correspondent users. In addition, if the team cannot know when something goes "down" before it affects the users, they will eventually spot it because of increasing user complaints. Neither of these scenarios is favourable for users, so there should be a way to preemptively detect the inability of an instance to respond to requests.

Problem

How can the routing service (e.g. a load balancer) and the team know when a service has unexpectedly become inoperative or extensively degraded and needs recovery?

Forces

- It is challenging to detect when a service fails or takes too long to respond.
- When that happens, it is usually too slow to remedy the service by intervening manually.
- Metrics are usually aggregated, so if a service goes down, its metrics stop being collected, and there is no indication that it failed.
- Any solution should introduce little to no overhead.

Solution

Implement a specialised endpoint that responds to requests without side effects. Then, configure another system (e.g. service, tool, load balancer) to periodically check that endpoint and take action when that fails, providing an automatic way to detect that the instance is unable to respond.

The **Liveness Endpoint** can be part of a RESTful API or a particular message topic. The only requirement is that it replies to requests without any side effects since its purpose is to test the ability of the **Application** to respond. Nonetheless, the endpoint can perform some more checks before responding. It can perform some application-specific logic and check the host's status to understand the liveness state of the instance [71]. Note that the more the endpoint is programmed to check, the longer it will take to respond and the more resources it will utilise. Thus, the team should balance the amount of processing required by the endpoint so it does not impact the service's performance.

Additionally, there must be a **Liveness Monitor** (e.g. service, tool, load balancer) that periodically pings that endpoint (cf. Figure 5.13). If the request fails or takes too long to arrive, the monitor can trigger an alert or automatic recovery process to fix the instance. This way, the team can notice the problem before it impacts the application's users. Furthermore, since the application is quickly recovered, its availability increases. Ideally, the liveness monitor runs in a different environment than the application so that problems that affect the users but are not visible from the inside (e.g. a misconfigured firewall that only blocks external traffic) can also be detected.

The monitor should allow the configuration of the interval between liveness checks, the timeout after which they fail if no response was received, the number of consecutive failures after which the application is considered unresponsive and the initial delay that the monitor should wait before starting to ping the endpoint. The last property is handy to avoid an infinite restart cycle. Applications usually take some time to startup, during which they cannot reply to requests. If the monitor restarts the application during startup, it will go back to the beginning of the startup process and eventually be restarted again. Thus, the team should carefully configure each of these parameters to create liveness checks that are strong enough to detect an unresponsive application but not too strong so they contemplate temporary system dynamics and do not cause too many restarts.

Finally, the liveness monitor requests' output can be recorded as a metric and monitored through a **Metrics Service**. This way, the team can keep track of the instance's up-time, user-perceived latency and, consequently, the instance's availability over a period of time [43]. Moreover, they can define acceptable thresholds for these metrics and configure alerts to trigger only in specific situations. For example, suppose the instance repeatedly fails over a couple of minutes. That is most likely a problem that needs human intervention instead of a sporadic instance failure that requires an automatic restart.

Consequences

This pattern has the following advantages:

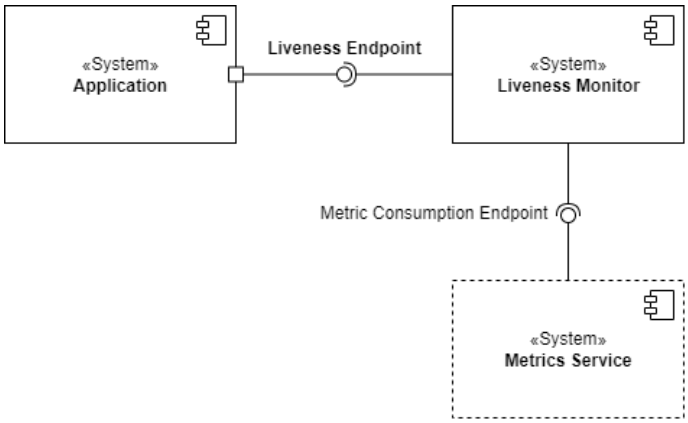


Figure 5.13: Overview of the structure for the LIVENESS HEALTH ENDPOINT pattern. The dashed component is optional, while the remaining are strictly necessary to adopt the pattern.

- Knowing the liveness state of the application, or part of it, allows the team to create automated recovery processes, which will reduce the system's mean time to repair (MTTR).
- The system's reliability increases because traffic is no longer routed to instances that cannot respond on time, causing user requests to fail less often and increasing the mean time between failures (MTBF) of the system.
- Alerts can be configured to set off when the application is "down".
- The system's up-time and user-perceived latency can be derived from the endpoint responses, providing two additional metrics to monitor. [43]
- It provides a detailed health status of the instance when combined with READINESS HEALTH ENDPOINT.

This pattern also has the following drawbacks:

- If the liveness status of an instance is complex to determine, adopting this pattern may require significant development effort, and liveness requests may take too long to process.
- When an instance of the application or service fails between liveness checks, traffic will still be routed to a dead instance; increasing the frequency of checks can mitigate this issue but introduce higher overhead on the system.
- Each liveness check is, at its core, another request that the system needs to answer; when traffic is higher, they represent unnecessary work and occupy resources that could be used to process user requests.

Example

Consider that a web application for image enhancement is deployed and serves user requests. It allows the users to upload an image in various formats and select a specific operation and the

number of iterations it should run. These operations could be, for example, a Gaussian blur or an edge enhancement algorithm.

The application follows a typical layered architecture, having a front-end service dedicated to the user interface that sends HTTP requests to an API exposed by a back-end service. There is also a database to store user accounts, the images they upload and the history of operations performed. The application is deployed using Kubernetes on a public cloud.

Due to a fault in the code, if the user uploads a JPG image and performs a Gaussian Blur with exactly one iteration, the back-end service will enter an infinite loop. This loop increases linearly in memory consumption, which means that the back-end service becomes slow or unresponsive to every user in a couple of minutes.

Without liveness health checks, this situation would go unnoticed until user complaints arise or someone from the team notices an unusually high memory consumption on the back-end's host machine. However, the team had implemented a simple HTTP GET endpoint (e.g. */health/alive*) in the back-end service that checks if the CPU and memory usages are below 80%, failing otherwise. They configured Kubernetes' liveness probe as follows:

```
livenessProbe:
    httpGet:
        path: /health/alive
        port: 8080
    timeoutSeconds: 1
    failureThreshold: 4
    initialDelaySeconds: 5
    periodSeconds: 3
```

Listing 5.3: Excerpt from the Kubernetes deployment configuration file for the LIVENESS HEALTH ENDPOINT example.

This configuration makes kubelet, Kubernetes' node agent, perform a GET request to the liveness health endpoint every three seconds (*periodSeconds*). The liveness checks will only start after an initial delay of five seconds (*initialDelaySeconds*) to give the container enough time to initialise. Otherwise, it could get stuck on a restart loop because the liveness checks would always fail while the application is starting up [11]. If the request fails or takes more than one second (*timeoutSeconds*) to be processed, the liveness probe fails. When this happens four consecutive times (*failureThreshold*), then kubelet will automatically restart the container.

The infinite loop issue renders the application unresponsive, causing the liveness probes to fail consecutively. Considering the utilised configuration and assuming the application was already running for five seconds before the issue, it would take around sixteen seconds for the container to be restarted. Afterwards, the application would return to normal execution and become responsive again.

Known uses

In [97], Yanacek, a Senior Principal Engineer at Amazon, provides some examples of liveness checks that Amazon uses. These are usually simple, such as basic HTTP requests to ensure the server responds and checks to confirm "that a server is listening on its expected port and accepting new TCP connections" [97].

In [76], Scott explains when and how to use Kubernetes' liveness and readiness probes. In addition, the author briefly mentions how liveness health checks helped Spire, his company at the time. Because they "have a number of services that can fail in ways that don't always result in the container crashing" [76], the liveness health endpoint was crucial to identify these cases and automatically restart the container to keep them responsive.

Finally, Owens [66], a senior site reliability engineer at New Relic, explains how the company's container orchestration team monitors the health of their product's containers. The author states, "health checks are critical to make sure an orchestration platform is correctly deploying and managing services at scale, so our developers define a health check for any service they run in a container" [66]. They make their liveness checks as quickly as possible but thorough enough that, if the service responds with success, they are confident that it is working correctly. Unfortunately, Owens does not mention which tool they use to implement the health checks.

Related patterns

This pattern is closely related to READINESS HEALTH ENDPOINT, and they can be adopted together. The main difference relies on the consequences of each pattern. While consecutive failed liveness checks should cause the service to restart, failed readiness checks should inform the load balancer that the service must stop receiving traffic. Suppose the team is looking for a way to automatically detect when a service cannot process requests, even though it is working correctly. In that case, they should consider adopting READINESS HEALTH ENDPOINT.

Nonetheless, both LIVENESS HEALTH ENDPOINT and READINESS HEALTH ENDPOINT suggest the implementation of an endpoint that is periodically checked. Thus, the patterns can be adopted together and merged into a single endpoint instead of having two distinct endpoints. This solution resembles OUTSIDE-IN HEALTH CHECK, proposed in [43], and HEALTH CHECK API, suggested in [72]. However, because of the previously mentioned difference between the endpoints and to keep each of them to a single responsibility, we prefer to split the patterns and adopt them independently.

Additionally, the team may consider adopting EXTERNAL MONITORING [83] to set up the liveness monitor. Doing so results in health checks that simulate end-user conditions better. Consequently, the team can discover more issues than if they perform the health checks in the same environment as the application.

Finally, a natural follow-up to this pattern is AUTOMATED RECOVERY [83]. Most of the value of a liveness check resides in the ability to automatically fix failing instances of the application. That is why many tools already provide an automated process to deal with liveness check failures, usually restarting the application.

5.11 Readiness Health Endpoint

Many factors affect cloud-hosted applications. These include, but are not limited to, network latency, performance and availability of the underlying compute and storage systems, or dependencies. External components may prevent a service from delivering its full functionality, so monitoring them is crucial to maintaining healthy services. The team should know when a running service becomes unable to process requests to stop routing traffic to that service. Therefore, they should implement a specialised endpoint that checks if the service is ready to accept and process traffic. They should also configure another system to periodically check that endpoint and stop routing traffic to the service when it fails.

Context

Many factors affect cloud-hosted applications. These include, but are not limited to, network latency, performance and availability of the underlying compute and storage systems, or external dependencies [60]. The application can fail entirely or partially due to external factors. Moreover, as is the case for cloud applications, different modules or services may run in other places and be replicated differently. "Binary concepts of a system being "up" or "down" start to have less and less meaning as the system becomes more complex" [64].

The team may have already adopted LIVENESS HEALTH ENDPOINT to mitigate this issue. However, that pattern focuses solely on the service's internal state, which can be a very simplified view of its health and, thus, insufficient to cover the team's needs. For example, a business logic component that needs to fetch data from the database to answer user requests may respond to the liveness checks but often fail when processing user requests due to connection problems with the database. External components may prevent a service from delivering its full functionality, so monitoring them is crucial to maintaining healthy services.

Problem

How can the routing service (e.g. load balancer) and the team know when a running service becomes unable to process requests?

Forces

- The health status of an instance of the application is more complex than just its ability to respond to requests. [64]
- User requests should not be routed to a service that is temporarily unable to process them.
- A service does not need to be restarted when only its dependencies fail.

Solution

Implement a specialised endpoint that checks if the service is ready to accept and process traffic. Then, configure another system (e.g. service, tool, load balancer) to periodically check that endpoint and stop routing traffic to the service when the check fails.

The **Readiness Endpoint** can be part of a RESTful API or a particular message topic. How the team implements it does not matter as long as it checks that the **Service** is ready to process user traffic. In other words, even if the service is running and responding to requests, it might not be able to perform the necessary operations to process the requests, which results in frequent failures. The service may end up in such a scenario "when one of its dependencies is unavailable, or while running a large batch job, performing maintenance, or something similar" [11]. The exact way to check that the service is ready varies from case to case, and its implementation is the responsibility of the developing team.

Nonetheless, there are some common situations that the implementation should check. Firstly, it is usually good to check if downstream dependencies are reachable and working. This may help the team find configuration issues, such as misconfigured credentials or SSL certificates, that prevent the system from connecting to its dependencies. However, dependencies may not be crucial for every service's operation. If these are unavailable, the service can still serve a portion of incoming traffic. Therefore, including a dependency on the readiness check should depend on its impact on the service's ability to process requests. Secondly, most services perform startup operations during which they are unavailable to user traffic. Thus, it can be helpful to check if the service has finished starting up in the readiness endpoint. Lastly, services may need to run periodic or complex tasks during which they are very slow to respond. They should stop receiving traffic and route it to other service instances during these periods.

Additionally, a **Readiness Monitor** (e.g. service, tool, load balancer) must periodically call the readiness endpoint (cf. Figure 5.14). If the request fails, the monitor should inform the load balancer to stop routing traffic to the failing service. This way, user requests will only be routed to available instances, likely decreasing the overall number of failures. In addition, the endpoint can generate a report of what is failing so that the monitor can trigger an alert or automatic recovery process to fix the problem. However, the team must consider the overhead that the checks introduce to the system. This overhead can be mitigated mainly in two ways (1) to check the endpoint less often, which increases the chance that something may fail between readiness checks, and (2) to perform more superficial checks. The team should balance these options and determine what works best for their concrete scenario.

The nature of the service's dependencies should also influence the readiness checks. When the endpoint only tests private dependencies, the readiness checks may fail more quickly or more times. On the contrary, if the dependencies are shared between services, the checks should be more forgiving and consider the impact of temporary system dynamics (e.g. increased latency, temporary loss of connection) [11]. The strength of the readiness checks can be adjusted in the endpoint, by checking fewer things or doing so in lower detail, or the monitor, by changing the period between checks, the timeout for each one or the number of consecutive failures that should occur before the monitor considers the service unavailable. This way, the team can prevent the readiness monitor from blocking all application traffic when a single shared dependency is unreachable.

Finally, the readiness monitor requests' output can be recorded as a metric and monitored through a **Metrics Service**. The team can track the service's availability during a specific period.

Moreover, they become able to correlate service unavailability with dependency issues. This service can also provide alerting capabilities so developers can define which scenarios they should be notified to take action, leaving the remaining ones for automated processes.

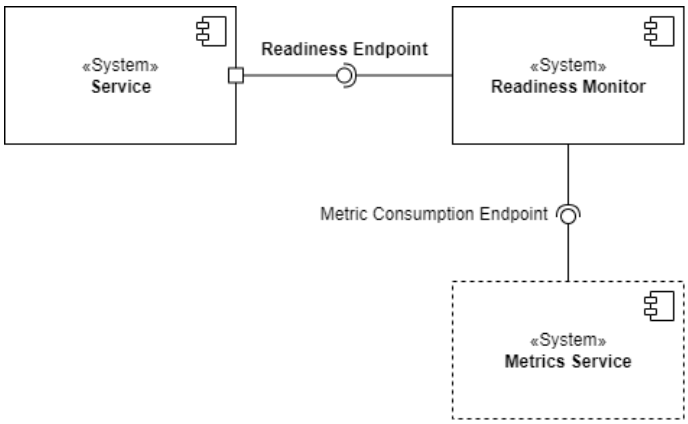


Figure 5.14: Overview of the structure for the READINESS HEALTH ENDPOINT pattern. The dashed component is optional, while the remaining are strictly necessary to adopt the pattern.

Consequences

This pattern has the following advantages:

- Alerts and automatic recovery can be configured to set off when the service is unavailable.
- It provides a more holistic health status of the instance, which is often required for complex systems.
- The system's reliability increases because traffic is no longer routed to instances that are unable to respond, causing user requests to fail less often and increasing the mean time between failures (MTBF) of the system.

This pattern also has the following drawbacks:

- When something fails between readiness checks, traffic will still be routed to unavailable services; increasing the frequency of checks can mitigate this issue but also increases the overhead on the system.
- If the readiness check is a complex and lengthy process, adopting this pattern may require significant development effort and impact the service's performance.
- Each readiness check is, at its core, another request that the system needs to answer; when traffic is higher, they represent unnecessary work and occupy resources that could be used to process user requests.

Example

Suppose a microservice is responsible for an online store's sales. It is essentially a back-end

module that exposes an API to serve other services and the user requests that come through the user interface. The sales service depends on the company's ERP, where the service records every sale for legal compliance. The application is deployed using Kubernetes on a public cloud.

For performance reasons, the service maintains an in-memory cache of records from the last fifteen days. Whenever there is an update, the cache needs to be restored. This process takes around two minutes to finish due to the amount of data transferred from the ERP. This task consumes many resources, making the service slow to process requests during startup. Therefore, the team wants to prevent user traffic from being routed to the service's instances that are restoring their private caches. However, during this time, the service is performing necessary operations, so the team does not want to restart the container while it is starting up. Hence, they adopted the READINESS HEALTH ENDPOINT instead of the LIVENESS HEALTH ENDPOINT.

To achieve their goal, the team implemented an HTTP GET endpoint (e.g. */health/ready*) as part of the service's API that checks if the cache has been successfully restored and if the ERP is accessible, failing otherwise. Afterwards, they configured Kubernetes' readiness probe as follows:

```
readinessProbe:
    httpGet:
        path: /health/ready
        port: 8080
    timeoutSeconds: 5
    initialDelaySeconds: 0
    periodSeconds: 3
```

Listing 5.4: Excerpt from the Kubernetes deployment configuration file for the READINESS HEALTH ENDPOINT example.

This configuration makes kubelet, Kubernetes' node agent, perform a GET request to the readiness health endpoint every three seconds (*periodSeconds*). The readiness checks will begin right after the container starts up (*initialDelaySeconds* is zero), so it does not serve traffic while restoring the cache. Since the service is a bit slow during startup, each health check has a timeout of five seconds (*timeoutSeconds*) to be processed. By default, the probe only considers the service unavailable after three consecutive failures (*failureThreshold*), after which kubelet stops routing traffic to that instance.

As the container starts up, it will be in an unready state, so kubelet will not route traffic to it. Instead, incoming traffic is routed to other instances of the sales service that are ready to process it. The readiness probe will start running right away and succeed after the cache is completely restored and the ERP accessible. The service will then begin serving requests, preventing users from being affected by its slow startup.

Note that the ERP is not essential for the service to work. After startup, the service can use the cache to register incoming transactions. However, the team checked its connection in the readiness endpoint because they had not implemented cache synchronisation with the ERP. In other words, if the ERP is not reachable, the data written in the cache will never make it to the

ERP, even after it becomes accessible again. So, if the ERP is not reachable during the service's lifetime, the readiness probe will fail, and the container will stop receiving traffic. Once the ERP is reachable, the readiness probe will succeed, and kubelet will start routing traffic to the mentioned instance. Once the team implements cache synchronisation, they can opt to remove the ERP from the readiness endpoint, allowing the service to work even if that dependency is temporarily not accessible.

Known uses

In [10], Bradtek explains how his team changed from their platform API to Kubernetes. They had a feature that allowed a service to signal if it was ready to serve user requests. After migrating to Kubernetes, they decided to use its readiness probe feature so the container "won't receive any traffic until a defined request can be successfully fulfilled" [10]. By combining the readiness probe and Kubernetes' init container feature, they were able "to ensure that Pods with dependencies do not get started before their dependencies are ready" [10].

In [76], Scott explains when and how to use Kubernetes' liveness and readiness probes. In addition, the author briefly mentions how readiness health checks helped Spire, his company at the time. One of their services took five to ten seconds between the the time the container started running and the time the service became able to respond to requests. So they implemented a readiness check to stop the load balancer from routing traffic to the service while it was starting up. As Scott states, "without a readiness probe, we'd end up with at least that much downtime every time we updated that deployment" [76].

Finally, Owens [66], a senior site reliability engineer at New Relic, explains how the company's container orchestration team monitors the health of their product's containers. The author states that "developers can define a readiness check, which provides a signal that a deployment is safe to continue" [66]. He explains that readiness checks are particularly useful if services take a long time to start up due to, for example, "migrating a database, establishing partition ownership, or warming caches" [66].

Related patterns

This pattern is closely related to LIVENESS HEALTH ENDPOINT and they can be adopted together. The main difference relies on the consequences of each pattern. While consecutive failed liveness checks should cause the service to restart, failed readiness checks should inform the load balancer that the service must stop receiving traffic. If the team is looking for an automatic way to detect when a service crashes or becomes extremely slow, they should consider adopting LIVENESS HEALTH ENDPOINT.

Nonetheless, both READINESS HEALTH ENDPOINT and LIVENESS HEALTH ENDPOINT suggest the implementation of an endpoint that is periodically checked. Thus, instead of having two distinct endpoints, the patterns can be adopted together and merged into a single endpoint. This solution resembles the one from OUTSIDE-IN HEALTH CHECK, proposed in [43], and HEALTH CHECK API, suggested in [72]. However, because of the previously mentioned difference be-

tween the endpoints and to keep each of them to a single responsibility, we prefer to split the patterns and adopt them independently.

Finally, the team may consider adopting AUTOMATED RECOVERY [83] alongside this pattern. Some tools already provide this feature out-of-the-box (e.g. Kubernetes automatically stops routing traffic when the readiness checks fail). In contrast, others may require manual specification of the recovery procedure. By automating this process, the team can focus on developing new features without interruptions.

5.12 Synthetic Testing

Tests are usually run in a controlled environment before the application is deployed and made available to end-users. However, once it is deployed, many factors can influence the application's behaviour. The team should create or pick a subset of existing test cases and periodically run them against the production environment, ensuring the application behaves as expected and detecting issues before they affect end-users.

Also known as

Synthetic Monitoring [24], Proactive Monitoring [24], Synthetic Transactions [64]

Context

Following CI/CD practices, most teams run tests in multiple stages of their deployment pipeline [62]. These tests are usually run in a controlled environment (e.g. a testing or staging environment) before the application is deployed and made available to end-users [61]. This way, teams validate that the product about to be released is behaving as expected.

However, once an application gets deployed, many factors can influence its behaviour. Some are internal (e.g. a code fault, little disk space), and others are external (e.g. connectivity issues, network configurations). When so many things can go wrong, the system becomes more fragile.

Moreover, the more distributed the application gets, the harder it is to replicate the production environment. Components are usually released multiple times a day, creating further instability [61]. "This rapid rate of change to the production environment tends to make testing during CI/CD stages not hermetic and actually not representative of the end-user experience and how the production system actually behaves" [61].

Problem

How can the team ensure that the application running in production behaves as expected?

Forces

- Automated tests are usually run in a controlled environment, which is not representative of the production environment.

- Traditional monitoring techniques (e.g. metrics, logs and traces) are only available when users interact with the system.
- Health endpoints are helpful to understanding if a service is healthy but do not provide information on how it is behaving.
- Detecting failures only after user complaints are issued should be avoided because it degrades the user experience.

Solution

Create or pick a subset of existing test cases and periodically run them against the production environment, ensuring the application behaves as expected and detecting issues before they affect end-users.

Synthetic tests are essentially a form of testing. Therefore, when adopting this pattern, the team should start by defining the **Test Cases** they want to execute. These can be created specifically for testing the **Production System** or be the same ones used in an existing CI/CD pipeline. There are two main kinds of synthetic tests: "browser tests, which check whether users can complete important transactions such as account sign-up and checkout, and API tests, which allow [the team] to monitor key endpoints at every network layer" [24]. These kinds are comparable to system-level and integration tests, thus having similar advantages and limitations. Moreover, they can be applied to many use cases and be used to check actual behaviour and the performance and availability of the application.

So far, this solution is closely related to already established testing practices. This pattern is set apart because synthetic tests must be run against the **Production System**. "Testing in production [is] arguably a form of "monitoring" activity" [64] because the team monitors the behaviour of our application after it is deployed. They should set up a **Synthetic Testing Platform** to periodically execute the test cases against the application (cf. Figure 5.15). The better the testing platform simulates a real user, the more accurate the test results will be. For the best possible coverage, the team should consider running their tests from different parts of the globe - at least where most of the application's users are located - and simulating different user environments (e.g. browser, OS and device). Obviously, the more the team tests, the more costly it becomes to maintain and execute the test suite. It is a trade-off that should be considered and balanced according to the services' monitoring needs.

The team must consider that they are effectively running tests in production. It is not as daunting as it may sound but still carries some risks. Microsoft's wiki reports the following potential issues with synthetic tests [61]:

- Corrupted or invalid data - Tests use invalid or corrupt test data. A schema may help enforce a proper structure on the data.
- Protected data leakage - Tests accidentally expose sensitive data.

- Overloaded systems - Tests crash or overload the system.
- Side effects - Tests trigger unintended side effects.
- Skewed analytics (traffic funnels, A/B test results, etc.)
- Auth/AuthZ - Tests are required to run in production where access to tokens and secrets may be restricted or more challenging to retrieve.

Finally, the **Test Results** can feed a **Metrics Service** that records the history of executions. The team can keep track of test evolution and correlate test failures with other system metrics. Detecting a failure and finding its cause are two distinct tasks. This pattern only helps with the former. In other words, a test failure will not provide much detail into the actual problem with the service. To find the problem, the team should have other artefacts (e.g. logs, traces and metrics) that they can use to troubleshoot the application.

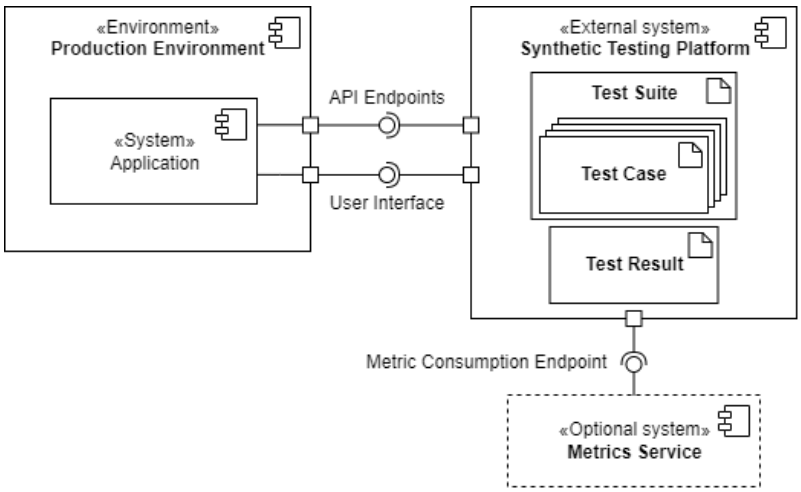


Figure 5.15: Overview of the structure for the SYNTHETIC TESTING pattern. The dashed component is optional, while the remaining are strictly necessary to adopt the pattern.

Consequences

This pattern has the following advantages:

- When a feature is not working as expected, the team can detect the issue before it impacts end users [24], effectively reducing the mean time to recovery (MTTR).
- The team can check whether performance requirements are met.
- It checks the actual system behaviour in production, complementing other monitoring practices more focused on the system's state.
- Synthetic tests can be run anytime, providing a way to monitor the system even when there is no user activity.

- It increases the team's confidence to deploy because any fault that makes it into production can still be detected before impacting the users.
- It provides another metric to monitor to help identify regressions and correlate them to specific events (e.g. deployments, infrastructure anomalies).

This pattern also has the following drawbacks:

- When a test fails, it usually only provides information regarding the expected and actual results, lacking context to identify the cause of the failure [64, 24].
- The testing system needs to be updated every time the application changes, even if there are only slight changes (e.g. renaming a UI component) [24].
- The test cases are a limited representation of what a real user can do, so they might still miss complex user interaction scenarios [21].
- Synthetic tests can be hard to set up, especially when considering real and complex scenarios involving data changes.
- Having an external testing platform requires additional infrastructure and maintenance, increasing the costs of the system.

Example

Newman [64] provides a great example of this pattern's adoption in his book *Building Microservices*. He explains that, some time ago, he was part of a team working on a system for an investment bank. They were responsible for reacting to trading market changes and evaluating the impact on the bank's portfolio. Market changes were constantly arriving in the system through events that it had to finish processing in under 10 seconds to meet the bank's requirements. "The system itself consisted of around five discrete services, at least one of which was running on a computing grid that, among other things, was scavenging unused CPU cycles on around 250 desktop hosts in the bank's disaster recovery centre" [64].

Due to the system's complexity, Newman reports that the low-level metrics the team was gathering were filled with noise. Moreover, they "didn't have the benefit of scaling gradually or having the system run for a few months to understand what «good» looked like in terms of low-level metrics like CPU rate or response time" [64].

Therefore, the team used synthetic tests to simulate events that would only affect a set of testing objects. They ensured the tests would not have any unintended side effects. Newman and his colleagues used a tool called Nagios to periodically run a command-line job that would generate a fake event and insert it into a queue. Their "system picked it up and ran all the various calculations just like any other job, except the results appeared in the «junk» book, which was used only for testing. If a repricing wasn't seen within a given time, Nagios reported this as an issue" [64].

Through synthetic tests, the team could guarantee they were meeting the system's performance requirements and that it was behaving as expected.

Known uses

Newman's [64] example described in the previous section is already a known use of this pattern that utilised Nagios to solve a real-world problem.

In a customer story, Splunk reports how Blue Apron improved its website load time by 30% [90]. Tom Wilson, a principal engineer at Blue Apron, states that "from testing new features to identifying easy performance wins, Splunk helps [them] integrate performance testing into our development life cycle" [90]. The company solved issues with its homepage through synthetic tests, reducing its load time by 30% and meeting its performance requirements.

Finally, a post by Carlbark on Meltwater's blog [18] explains how Meltwater's API was synthetically tested. The system's production environment was so complex that including everything in pipeline tests was slow (or sometimes unfeasible) to achieve. Therefore, Meltwater's team developed an in-house tool to carry test cases against the production environment. As a result, they found faults in their components and 3rd party dependencies, which allowed the team to fix many issues before going into the product's beta stage [18].

Related patterns

Ideally, the synthetic testing platform is running externally from the production environment. In other words, the team can adopt EXTERNAL MONITORING [83] alongside this pattern. Otherwise, the tests may not be affected by factors that influence the users, such as network latency, firewalls and localisation.

The test results are themselves something that should be monitored. Therefore, the team can adopt APPLICATION METRICS [72] and register the test results and metadata (e.g. execution time, service name) as system metrics. This way, alerts can be set when tests fail more than a specified number of times per time frame.

Finally, the LIVENESS HEALTH ENDPOINT pattern requires another service to perform the health checks periodically. Thus, that pattern can be implemented through SYNTHETIC TESTING. Each health check becomes a synthetic test whose expected outcome is just a response with a success code. If so, the team will only need to set up and maintain a single infrastructure, reducing the costs of adopting both patterns.

5.13 Application Metrics

Since the cloud abstracts away the infrastructure on which the code is running, performance becomes more about how well the system can respond to the user than how efficiently it uses the available resources. An overview of how the application is performing and understanding emerging usage patterns is crucial to maintaining a quality system. Therefore, the team should instrument the application to gather business and performance metrics to then collect these metrics in

a centralised service that provides aggregation and visualisation, allowing deeper insight into the application's performance.

Also known as

Inside-out health check [43], Monitoring Metrics [96]

Context

Nowadays, software systems not only need to bring value through appealing features but also to work extremely fast. Users expect things to answer quickly and fail rarely. In such a demanding environment, teams must be able to keep track of their system's performance to ensure it meets certain requirements. These may be informal and more flexible, such as an end-user of a web application that expects a page to load in less than 30 seconds, or formal and strict, such as a service level agreement (SLAs) between an infrastructure as a service (IaaS) provider and its client. Both cases are equally important for a company since it might be losing customers or having to compensate them for a contractual breach, respectively.

Since the cloud abstracts away the infrastructure on which the code is running, performance becomes more about how well the system can respond to the user than how efficiently it uses the available resources. Moreover, performance can be seen from the technical and business lenses. Understanding customer trends and satisfaction is just as important as knowing availability and the number of errors, for example. Both can bring helpful insight to a company and give it a more significant market presence.

Problem

How can the team get an overview of how the application is performing and understand the emerging usage patterns?

Forces

- Processing logs to extract relevant system performance measures is a computationally expensive process and does not provide real-time data.
- Simulating requests and testing in production provide a small amount of data and only for already previously defined scenarios.
- Infrastructure is managed by an external entity that usually provides infrastructure-level metrics.
- Reusable solutions are more desirable as they help mitigate development and infrastructure costs.

Solution

Instrument the application to gather business and performance metrics. Collect these metrics in

a centralised service that provides aggregation and visualisation, allowing deeper insight into the application's performance.

The team can collect **Metrics** on many things, such as request rate, error rate, customer orders, and user posts. Ideally, one would measure everything, but that introduces more overhead and requires considerable infrastructure. That said, the first decision the team must make when adopting this pattern is to define which **Application** metrics they wish to monitor. Consider The Four Golden Signals proposed in [31] as a starting point. They are briefly presented below, but for further detail refer to the original work:

- Latency - the time between a user request arrives in the system, and a response is issued to the user.
- Traffic - a "high-level system-specific metric" [31] that measures the number of incoming requests in your system.
- Errors - the rate of requests that result in a failure, of any kind, by a period of time.
- Saturation - the percentage of system resources being utilised at the moment.

There are other guidelines based on the Golden Signals, like the RED method [45] and the READS metrics [44], so the team should consider which one is more suited to their context.

On top of the Golden Signals, collecting metrics aimed at business objectives, user actions, or high-level application-specific values may be beneficial. In the case of websites, some of these are already provided by systems that collect analytics. In contrast, others can be included manually (e.g. user orders, user turnover, user satisfaction, user posts or wishlists). Remember that the more the team collects, the harder it will be to handle.

Once we know what to measure, we should understand how to do it. Each metric is sampled periodically by a **Metrics Exporter** and is usually composed of a name, value and timestamp [71]. Further context must be added to the metric through tags, i.e. key-value pairs for external properties, that are collected each time step. The higher the cardinality of the added tags (e.g. usernames and emails, which are unbounded sets), the more the metrics system will struggle to keep all the information. When adopting this pattern, the team should consider reducing the cardinality of these tags or using a system that is prepared to receive high cardinality data.

Another thing to consider is the resolution of data, i.e. the number of data points collected by time period. Depending on the current context of the system, some metrics may be collected and stored in a greater resolution than others. As Newman exemplifies:

> *"I might want a CPU sample for my servers at the resolution of one sample every 10 seconds for the last 30 minutes, in order to better react to a situation that is currently unfolding. On the other hand, the CPU samples from my servers from last month are likely needed only for general trend analysis, so I might be happy with calculating an average CPU sample on a per-hour basis."* [64, p. 322]

Therefore, when adopting this pattern, the team should consider having the ability to change the resolution of the data that is stored or being collected. That allows the team to better manage storage space and overhead while still collecting a sufficient number of metric points for troubleshooting.

Finally, metrics generated and spread across many services are not very helpful. To make actual use of them, the team needs a **Metrics Service** - a centralised system to collect and aggregate (e.g. to calculate the average, sum or percentiles) the metrics and present them in a friendly way to the team (cf. Figure 5.16). This can be achieved in two ways [72]:

- push - the application sends the metrics to an API provided by the metrics service (e.g. AWS CloudWatch).
- pull - the metrics service fetches the metrics data from an API provided by the application (e.g. Prometheus).

Both options are viable; it all depends on the context of the system. Usually, the pull model is more transparent to the application. In other words, it does not require as much instrumentation, keeping the business logic more isolated from these monitoring details.

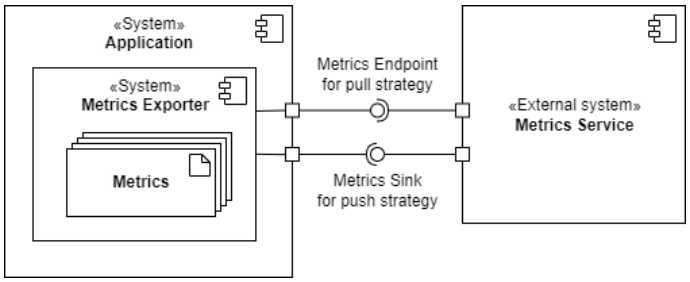


Figure 5.16: Overview of the structure for the APPLICATION METRICS pattern. The solutions usually only follows the pull or the push strategy. We represented both for completeness.

Consequences

This pattern has the following advantages:

- It provides an up-to-date and straightforward overview of the application's state.
- Application metrics can be helpful from both the technical perspective (e.g. capacity planning) and business perspective (e.g. business insights).
- Application metrics are a potential source of information for strategic decisions, motivating data-driven decision-making.
- Metrics enable the team to notice non-evident patterns and problems that would most likely go unnoticed otherwise.
- The collected metrics allow the team to predict possible problems, such as disk space running out, and figure out trends of, for example, capacity planning.

- The team can identify issues before they affect the system's users when application metrics are paired with alerts.

This pattern also has the following drawbacks:

- Aggregating and storing a high number of metrics may require considerable infrastructure, increasing its costs.
- Instrumenting the source code to measure the metrics makes the business logic more complex and harder to understand.
- Collecting application data in the cloud depends heavily on what the team has access to, so this pattern gets harder to adopt the fewer access rights the team has to the deployment environments.

Example

A post by Santiago Campuzano on GumGum Tech Blog [17] explains how the company adopted Prometheus to monitor its systems. Before the company adopted the microservice architecture, the author states that monitoring was quite simple. Their "applications, servers, and services were pretty much fixed and well-known" [17]. However, things got much more complicated once they started containerising their applications and adopting the microservices architecture. As the author states: "the explosion in the quantity and complexity of the systems to be monitored was neither manageable nor sustainable with the legacy monitoring stack that we had in place" [17].

To address the new monitoring needs, they used Prometheus to create a monitoring stack capable of handling the complexity. According to the author, they "chose Prometheus among other systems, mostly because of its flexibility, extensibility and huge open source community backing the project" [17]. They used different Prometheus exporters to instrument the code to generate metrics. The tool follows a *pull* strategy to get the metrics, i.e., it periodically scrapes them from a configurable endpoint. GumGum's monitoring stack collected metrics from the application, containers and servers. Additionally, they used Grafana to visualise the collected metrics on customised dashboards and Alert Manager to configure and manage alerts on top of the Prometheus metrics. Figure 5.17 is part of the post and depicts the overall Prometheus architecture used by GumGum.

Unfortunately, the author does not precisely mention which metrics they were collecting. Nonetheless, Campuzano mentions that they used Prometheus Pushgateway for "ephemeral and batch jobs to expose its metrics to Prometheus" [17]. These jobs send the metrics to the Pushgateway, which Prometheus then scrapes. Thus, they could collect application-level metrics and visualise them in specific dashboards for their teams. With Prometheus and Grafana, GumGum became able to collect metrics from many parts of their system and "create beautiful and meaningful Grafana dashboards with those metrics" [17].

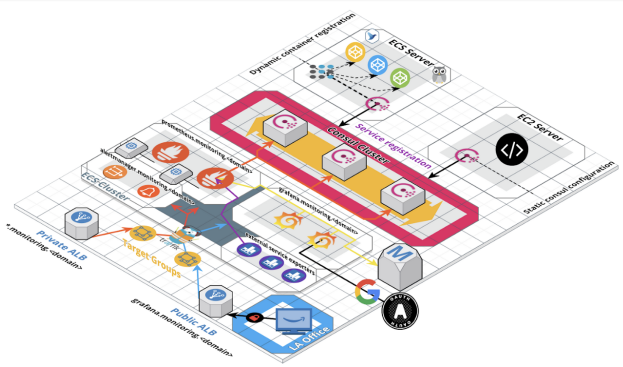


Figure 5.17: Real-world example of the APPLICATION METRICS pattern. [17]

Known uses

Campuzano's [17] example described in the previous section is already a known use of this pattern that used Prometheus and Grafana to collect metrics from a microservice architecture.

A post by Domenico Stragliotto [93], a backend developer for THRON, elaborates how the company adopted metrics on their system. The author explains they were looking to improve their ability to analyse long-term trends, build dashboards, troubleshoot and get alerted. To achieve their goals, they used Prometheus to collect metrics from their services. They followed the RED method [45], so, for each service, they collected the number of requests per second, the number of errors per second and the amount of time to process each request. These metrics were then exposed in Grafana dashboards so the teams could easily view each service's state.

Finally, an infrastructure software engineer post on callstats.io's blog [16] reports how the company uses Prometheus to monitor its services. They chose Prometheus due to its seamless integration with Kubernetes, powerful query language and operational simplicity. The engineer states that the company's developers "use Prometheus to compare performance and resource usage between service releases " [16], while the analytics team uses it "for several artificial intelligence-related metrics " [16].

Related patterns

Although the cloud abstracts the infrastructure where the application runs, understanding the performance of the host machine can be helpful. The underlying machine's performance can impact the application's performance. Suppose the team needs a more comprehensive view of the system or further correlation between the application and its infrastructure. In that case, they should consider adopting INFRASTRUCTURE METRICS to complement the metrics provided by this pattern.

5.14 Infrastructure Metrics

Software needs infrastructure to execute. Even though the cloud abstracts this infrastructure from the developing team, applications still depend on the underlying resources. Scarcity of resources like CPU, memory or disk can bring extreme system slowness or even an outage. The team should

be able to know the state of their application's infrastructure and correlate it with ongoing problems. Therefore, they should instrument the server and runtimes to capture relevant metrics of the operative system and underlying infrastructure and collect them in a centralised server, allowing the team to get a real-time overview of the application's environment.

Also known as

Inside-Out Health Check [43]

Context

Software needs infrastructure to execute. Even though the cloud abstracts this infrastructure from the developing team, applications still depend on the underlying resources. Scarcity of resources like CPU, memory or disk can bring extreme system slowness or even an outage. In addition, these issues can be caused by faults in the code (e.g. memory leaks), so it is the developing team's responsibility to find and fix them. The team should monitor the essential infrastructure resources to identify code-level problems and correlate end-user issues with the infrastructure.

In a pay-as-you-go scenario, which allows for greater flexibility and cost savings, it will drive the costs up if the application starts to utilise more and more resources. Being able to notice the increasing usage of infrastructure is necessary so the team can figure out if that is due to greater demand on the application (e.g. a new product release with a significant influx of users), in which case it is to be expected, or an issue with the code (e.g. poorly optimised loops or memory leaks) that needs to be fixed. Moreover, public clouds are usually under service level agreements (SLAs) that declare the minimum requirements the infrastructure must provide. If a given resource constantly drops below the agreed threshold, the cloud consumer should be able to confront the provider and take action according to what is stated in the SLA. Thus, understanding the performance and state of the infrastructure is relevant for both cloud providers and consumers, so the machines the software is running on should be monitored.

Problem

How can the team know the state of their application's infrastructure and correlate it with ongoing problems?

Forces

- The system needs to scale when a particular resource threshold is crossed, which may require the team's manual intervention in certain cloud providers.
- Uncontrolled resource usage can significantly and unexpectedly increase the system's costs.
- Application modules or services may be spread across many infrastructure resources.
- Resource usage may vary immensely depending on the ongoing operation and the number of concurrent requests.

- Infrastructure provisioning may be under a contractual agreement (e.g. SLA) that needs to comply.
- Reusable solutions are more desirable as they help mitigate development and infrastructure costs.

Solution

Instrument the server and runtimes to capture relevant metrics of the operative system and underlying infrastructure and collect them in a centralised server, allowing the team to get a real-time overview of the application's environment.

As explained in Application Metrics (see Section 5.13), **Metrics** are usually composed of a name, a value and a timestamp [72]. Tags can be stuffed into the metrics to provide further context. Tags are key-value pairs of properties, so they can come with nearly any kind of information. However, the higher the number of different values each tag can have, or, in other words, the higher its cardinality, the harder it is to store and process.

Instrumentation can take many forms, but it is essentially the act of adding measuring instruments to the system. These instruments collect the **Infrastructure** information and generate metrics out of it. In the context of this pattern, instrumenting the server and runtimes means having a **Metrics Exporter** that is responsible for gathering a predefined set of metrics. It must have access to the OS and underlying infrastructure and, ideally, be configurable to specify what should be captured and at what resolution (i.e. the number of data points per time interval).

Some public clouds, like GCP, provide an API with each application engine that exposes infrastructure metrics. Note that this API, by itself, is already a suitable exporter. It is gathering the necessary information and exposing it. Even though this kind of API may not be configurable (e.g. you can not change the sampling rate), the team can request exclusively the metrics they need and at a larger interval than what the exporter samples.

Configuring the sampling rate may be necessary since storing and processing large amounts of data can demand a considerable monitoring infrastructure and slow the overall monitoring process. Therefore, when adopting this pattern, the team should decide how many data points should be collected and how many should be stored.

Another crucial decision is what metrics to monitor. As previously stated, the more metrics the team collects, the easier it is to understand the infrastructure's state. Thus, the team should start with the most useful ones for their case and expand as the need arises. The USE method is a common method to help with this decision [42]. It suggests that the following metrics should be the first to be collected:

- Utilisation - "the percentage of time that the resource is busy servicing work during a specific time interval" [42];
- Saturation - "the degree to which the resource has extra work which it cannot service, often queued" [41];

- Errors - "the count of error events" [42].

Note that the USE method is more than just an indication of the metrics that should be gathered. It also incentivises the team to start by identifying the resources in their system (e.g. CPU, memory, storage). Only then should they define what each of the above metrics means to each resource and how they should be gathered. That way, the USE method increases the overall understanding of the system and turns most unknown unknowns into known unknowns. In other words, when the team notices a missing metric that would be useful to solve a currently ongoing application problem, they can start from the list of metrics they know they are not gathering.

Finally, metrics generated and spread across many services are not very helpful. To make actual use of them, the team needs a **Metrics Service** - a centralised system to collect and aggregate (e.g., calculate the average, sum or percentiles) the metrics and present them in a friendly way to the team (cf. Figure 5.18). This can be achieved in two ways [72]:

- push - the application sends the metrics to an API provided by the metrics service (e.g. AWS CloudWatch);
- pull - the metrics service fetches the metrics data from an API provided by the infrastructure (e.g. Prometheus)

Both options are viable. It all depends on the system's context. Usually, the pull model is more transparent to the application. In other words, it does not require as much instrumentation, keeping the business logic more isolated from these monitoring details.

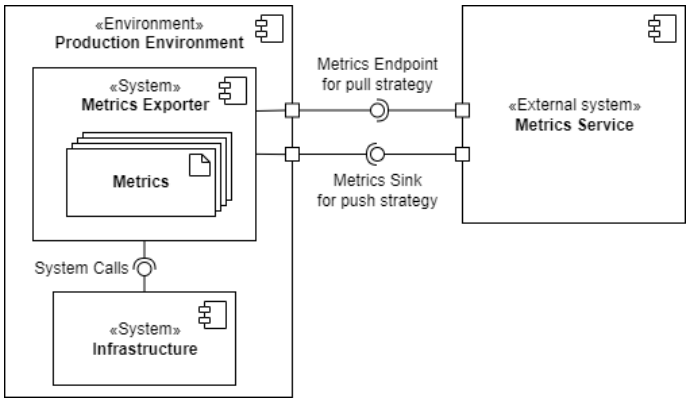


Figure 5.18: Overview of the structure for the INFRASTRUCTURE METRICS pattern. The solutions usually only follows the pull or the push strategy. We represented both for completeness.

Consequences

This pattern has the following advantages:

- Metrics enable the team to notice non-evident patterns and problems that would most likely go unnoticed otherwise.
- It provides a real-time and straightforward overview of the infrastructure's state.

- Operations teams can more easily cut down on unused resources, decreasing the system's costs.
- Issues with the code that directly impact the infrastructure (e.g. memory leaks) become easier to detect.
- It allows for correlation between user symptoms, like recurring failures when creating a new post, to infrastructure problems, like a full database.

This pattern also has the following drawbacks:

- Aggregating and storing a high number of metrics may require considerable infrastructure, increasing its costs.
- Correlating infrastructure and application metrics is based on the time of events; since the systems collecting the metrics are different, their clock times may also diverge slightly, making way for wrong relations between both metric types.
- Data can quickly become separated into different categories (or silos) without actual correlation, which heavily reduces this pattern's benefits.
- Collecting infrastructure data in the cloud depends heavily on what the team has access to, so this pattern gets harder to adopt the fewer access rights the team has to the infrastructure.
- In PaaS and FaaS scenarios, the only way to collect infrastructure metrics may be through the cloud provider's dedicated solutions, which reduces the team's flexibility and restricts the kind of data that can be monitored.
- Analysing this metrics may be misleading because these convey possible problems (i.e. the lack of resources) but do not point towards the actual problem (i.e. the faulty code).

Example

In an online blog post, Stragliotto [93] provides an example of how THRON adopted this pattern. The author considers that "monitoring is the most important starting point to improve your product" [93], so they decided to update their monitoring architecture to improve their troubleshooting and alerting capabilities. In addition, they also improve their ability to analyse long-term trends and build dashboards.

The team decided to use Prometheus to collect infrastructure metrics to achieve their goal. The decision to use that tool was mostly based on the team's needs regarding the software and infrastructure. According to the author, the features that they valued the most in Prometheus were [93]:

- "a data model based on time series data identified by metric name and key/value pairs".

- "a really flexible and powerful query language that helps to aggregate data, the results can be aggregated in real time and directly shown or consumed via HTTP API to allow external system to display the data".
- "no reliance on distributed storage; nodes are single server and autonomous".

Stragliotto explains that for their micro-service infrastructure, Prometheus was their best option. In addition, they adopted Grafana "to query, visualise and generate alerts from our metrics" [93]. From the grey literature, we found that combining these two tools is a ubiquitous way to set up a **Metrics Service**.

The author goes on to talk about the specific metrics that they chose to collect from their services. The post mentions the Four Golden Signals [31] as the most important to monitor a system, but for the infrastructure, they followed the USE method [42]. Stragliotto defends that this method is better suited when the need is to "keep the physical resources under control" [93]. Therefore, they collected utilisation, saturation and errors from many resources (e.g., CPU, memory, discs).

However, the author warns that:

> *"The USE Method helped us identify problems which could be system bottlenecks and take appropriate countermeasures, but it requires cautious investigation since systems are complex."* [93]

He argues that analysing these metrics is not obvious. Since they convey the state of the infrastructure, they point to possible problems, but they are not the actual problem. This is a consequence that we consider very important for the pattern.

To conclude, Stragliotto declares that they "are happy about how [their] new architecture turned out, it works and it's starting to really help [them] keep [their] software under control" [93]. This view of the system was essential to make their detection and troubleshooting processes faster and fix their services quicker.

Known uses

Stragliotto's [93] example described in the previous section is already a known use of this pattern that used Prometheus and Grafana to collect and visualise the utilisation, saturation and errors of their system's infrastructure.

An infrastructure software engineer post on callstats.io's blog [16] reports how the company uses Prometheus to monitor its services. They chose Prometheus for its seamless integration with Kubernetes, powerful query language and operational simplicity. The engineer states that the company's infrastructure and operations teams "use [Prometheus] to monitor resource usage and service performance, as well as data pipeline processing queues " [16].

Finally, a post by Santiago Campuzano on GumGum's Tech Blog [17] explains how the company adopted Prometheus to monitor its systems. In addition, they utilised Grafana to visualise the data with customised dashboards. Although the author does not write about the actual metrics they collected, he mentions the usage of Prometheus' node exporter. This metrics exporter exposes

"hardware and OS metrics on *nix systems" [17] and publishes more than 600 metrics. The author recognises that this amount of metrics can be a problem due to the high processing overhead they introduce.

Related patterns

The infrastructure metrics represent the state of the infrastructure. Analysing these logs helps notice symptoms of a problem. However, they are not usually enough to find what exactly is causing the issue. In such cases, the team should consider adopting APPLICATION METRICS to complement this pattern and get a full view of the system.

5.15 Conclusions

As this chapter comes to an end, we provide an answer to this dissertation's first research question:

RQ1. Which patterns can be defined from current monitoring practices?

In this chapter we described eleven monitoring design patterns: AUDIT LOGGING, STANDARD LOGGING, LOG SAMPLING, DISTRIBUTED TRACING, DEPLOYMENT TRACKING, EXCEPTION TRACKING, LIVENESS HEALTH ENDPOINT, READINESS HEALTH ENDPOINT, SYNTHETIC TESTING, APPLICATION METRICS and INFRASTRUCTURE METRICS.

These patterns were based on existing literature, monitoring tools, and our own experience. We believe their descriptions present a good level of detail and are suited for beginners and experienced professionals alike. They all provide thorough examples of their usage and three Known Uses, conforming to the *Rule of Three* [49]. Moreover, we analysed possible relations between these patterns and other existing monitoring patterns identified during the literature review of Chapter 3. Doing so enabled us to write the Related Patterns sections with a broader scope of patterns, providing more comprehensive information to the readers.

Most of these patterns have a similar granularity. In other words, they tackle problems of comparable sizes using solutions of similar complexity. However, the SYNTHETIC TESTING pattern seems slightly too broad compared to the remaining patterns. During our research, we planned to write two patterns for synthetic tests — MULTISTEP SYNTHETIC TESTING and API SYNTHETIC TESTING. Due to time constraints and difficulty finding a clear separation between these two smaller patterns, we decided to leave SYNTHETIC TESTING as is. Nonetheless, we are satisfied with the resulting pattern.

On the negative side, we could not arrange these patterns in a pattern language despite our efforts. We could not find a suitable order for their adoption. All these patterns seem equally important depending on the context. Although some relations we identified indicate that one pattern should be adopted before another, most of them can be used independently.

In addition, even though we proposed a classification for the literature monitoring practices and patterns (see Section 3.3), we did not use it to classify the proposed design patterns. The reason is twofold. On the one hand, separating them into categories would create the false idea

they apply on different fronts of the problem. They all share the same goal: improving monitoring capabilities for cloud-hosted applications. On the other hand, the initial classification we used was not pragmatic enough to provide clear differentiation between patterns, with some of them lying in the middle of two categories. Nonetheless, we realise that some readers might have preferred to have these categories rather than none, so we consider this a shortcoming of our pattern catalogue.

To conclude, these eleven patterns are the first contribution of this dissertation to the software engineering field. Although they have shortcomings, we believe they will be helpful for practitioners and evolve with time by other studies. However, these are new patterns without any empirical corroboration whatsoever, so they are as good as brand new theories [49]. In the next chapter, we pick the catalogue and apply some patterns to a real-world project to evaluate their value.

Chapter 6

Empirical study

With a pattern catalogue ready for actual usage, we can now move on to test it in a real-world case. In this chapter, we conduct and describe an empirical study to provide initial empirical corroboration of the proposed patterns.

In Section 6.1, we describe the empirical study's goals and explain how it answers the dissertation's second research question. Afterwards, in Section 6.2, we provide an overall methodology for the study and a thorough description of each of its phases. Moreover, the concrete goals, inputs and outputs of each phase of the study are explained in this section. In Sections 6.3, 6.4 and 6.5, we describe the results for the three phases of the study and explore them to reach meaningful conclusions. For each phase, we discuss its main findings and comment on what are their implications for our work. Next, in Section 6.6, we present the main threats to validity and explain how they were dealt with during the study. Finally, in Section 6.7, we summarise the findings more relevant to our work and provide an answer to the dissertation's second research question.

6.1 Goals

As Kohls & Panke state:

> *"Patterns that are mined based on "real stuff" must necessarily be formulated in a way that they account for previous cases. But do they hold for future cases? This we do not know. The degree of corroboration is higher if we test a pattern for new cases and we must allow the option that patterns can fail. That makes a pattern empirically vulnerable: if we apply a pattern in the right context but it does not solve the present problems it is indeed falsified. That patterns can fail in principle is a good thing because it is the only way to test them." [49, p. 6]*

Therefore, we conduct this empirical study to test some of the patterns proposed in this dissertation and provide initial corroboration of our work. If a pattern is applied in a real-world

scenario that meets its context but it is not able to solve the problem, then it should be reconsidered. Throughout the study, we seek to identify shortcomings in the patterns' descriptions and address them to improve the proposed pattern catalogue. More specifically, our main goal with the empirical study is to answer the dissertation's second research question:

RQ2. How accurate are the proposed patterns' contexts, forces, problems, solutions and consequences?

In the following section, we explain the methodology used to find an answer to the question above.

6.2 Methodology

To answer the research question mentioned in the previous section, we conduct a three-phase empirical study. The study follows an action research methodology where we get involved in a real-world project at Konkconsulting and interact directly with its team. First, we dive into the project to understand its context, including its architecture, production environment, deployment strategy and team composition. During this phase, we also try to understand which monitoring practices are already in place and which monitoring challenges the project is still facing. This information is crucial to select patterns that apply in the project's context and to guide the implementation of monitoring tools.

Secondly, we implement some of this dissertation's patterns in the project to get a deeper understanding of the project and first-hand experience with its monitoring challenges. The pattern selection process is based on the patterns' contexts and solutions. In other words, we ensure the selected patterns' contexts match the project's and that the patterns address the project's monitoring challenges. This way we apply the patterns in a real-world scenario and start noticing possible shortcomings in their descriptions.

Finally, we conduct a focus group to evaluate the work done so far. We invite a group of professionals who have had recent contact with the project to a brainstorming session around the project's monitoring challenges identified during the previous two phases. We compare the findings from this session with the selected patterns and discuss some implications of their adoption in the project.

The following subsections further detail the process we take for each phase of the empirical study.

6.2.1 Project characterisation

The project characterisation phase is the first phase of the empirical study. We start with a very informal semi-structured interview (SSI) with the project's team leader. Its structure is outlined in Appendix A. Our goal with the interview is not to provide a detailed analysis of the project but to get a first contact with the project in order to get comfortable with its context and identify the

main monitoring challenges it faces. During the interview, we ask about the project's architecture, technology stack, production and development environments, deployment strategy and team composition. Additionally, we question the team leader about existing monitoring practices and challenges in the project.

Afterwards, we analyse the project's source code and technical documentation. On the one hand, this analysis helps us confirm the findings from the interview. On the other hand, analysing these artefacts allows us to fill in some gaps and get acquainted with the code for the next phase of the empirical study. We focus the analysis on the same aspects that we seek in the interview, with the addition of monitoring tools and their configurations.

To end this phase, we prepare a small survey to confirm what we found until this moment and explore possible paths for the implementation phase. It is aimed at everyone with recent contact with the project that had to, at least once, fix a bug/issue in the code. The survey's questions are outlined in Appendix B. With questions 1, 2 and 3, we aim to characterise the surveyees based on their experience with the project. This characterisation allows us to correlate answers with experience and make conclusions based on it. Questions 4 and 5 are primarily confirmatory. In other words, we seek to confirm our conclusions regarding monitoring practices from the two last parts of the project characterisation phase.

Then, from questions 6 to 9, we try to collect relevant metrics that could show the impact of improving the project's monitoring practices. Questions 10 to 19 aim at understanding which monitoring problems affect the project the most, confirming what we concluded from the team leader interview. The questions map directly to each pattern's problem. Thus we also use these results to help guide the pattern selection process. Questions 20 and 21 take a slightly different approach and focus on the solutions we identified in the proposed patterns. The goal is to compare the adopted solutions with the problems from the previous set of questions, allowing us to conclude a match or mismatch between existing problems and solutions. Finally, with questions 22, 23 and 24, we look towards understanding how failures in the system happen, where they are most noticeable, and what is the team's experience with monitoring tools. This information helps us guide the implementation and focus it on the parts of the system that will benefit the most from better monitoring solutions. These last three questions are based on another survey of APM tools adoption originally made by eG Innovations and DevOps Institute[1].

To conclude, we analyse and correlate the results of each of these parts to get a complete view of the project's context, its organisation, and monitoring challenges. With this knowledge, we can move on to the next phase of the empirical study.

6.2.2 Pattern selection and implementation

The second phase of this empirical study has two separate parts. The first one is pattern selection, whose goal is to pick suitable solutions for the project's monitoring challenges and identify shortcomings in the description of the patterns' contexts and solutions. The monitoring challenges

[1]The survey can be found at https://www.surveymonkey.com/r/APM_Survey

we find during the project characterisation phase are unlikely to match directly with the patterns' problems. Therefore, we discuss these challenges and elaborate on the underlying monitoring problems. Afterwards, we look for patterns that tackle similar problems and ensure they apply to the project, i.e., their contexts match the project's context. Note that we do not restrict our pattern search to the design patterns proposed in this dissertation. Instead, we consider all monitoring patterns we found during the state-of-the-art research (see Chapter 3). Then, we check if the solution described in each pattern is suitable for the project or not by ensuring that there are no implementation restrictions that make the solution unfeasible and that the solution solves the problem we are tackling. This exercise is already a way to test if the pattern's solution applies in the described context. Suppose there is a pattern for which the contexts match, but the solution does not apply to the project. In that case, the pattern's context needs further refinement to exclude the condition that makes the solution unfeasible in our current scenario. The pattern selection process results in a list of design patterns to be implemented in the project.

The second part is the implementation of the selected patterns in the project. Its goal is to increase our acquaintance with the source code and provide a first-hand experience with the project's monitoring challenges. We start by looking for monitoring tools that match the selected design pattern's solution. Plenty of tools are dedicated to monitoring and observability [64], so there is no need to create dedicated solutions. We choose and implement the tools considering the project's technical details and the conclusions from the project characterisation phase survey. In addition, we look into each pattern's Known Uses section to search for tools that are guaranteed implementations of the pattern. After selecting the tools, we implement them to better understand the project's intricacies.

By the end of this phase, we have a list of implemented patterns that still requires validation. Our interpretation of the problems and the patterns' descriptions is susceptible to bias. Hence, we take the selected patterns to the team and discuss them in the next phase of the empirical study.

6.2.3 Focus group

The last phase of this empirical study is the focus group activity. Its goals are to evaluate the conclusions from the previous phases of the study and reduce possible bias, identify shortcomings in the selected patterns' descriptions and reiterate them to improve the patterns' quality and find discrepancies between the selected patterns and what the participants comment throughout the activity.

Following the two previous phases, the focus group takes place in the project context. This implies that, while we look for new ideas and insights regarding the project's monitoring challenges, we only invite people who have had recent contact with the project. We believe this is a reasonable criterion because the contextualisation, pattern selection and implementation happen in the project's scope.

The focus group is planned for a hybrid format, with participants online and on-site, and is organised into two parts: the exploratory and the confirmatory. The first part aims to find additional information that could fit in the patterns or become new patterns altogether. To that

end, we explore other possible solutions to the project's monitoring challenges and explore their contexts, forces and consequences through a brainstorming session.

We follow a similar methodology to the one that Manns & Rising [56] use for their training sessions to teach the fearless change patterns. We write the monitoring challenges in the header of A3 paper sheets. Instead of having them go around the table with the participants [56], we expose them on a whiteboard to have more control during the start of the activity. The initial setup of the room is shown in Appendix C. We begin by explaining brainstorming rules and guidelines, so the participants understand our goal and how to contribute to the session. Afterwards, we run a small example with the participants to help them get into a creative mindset. We present them with an A4 paper sheet with the challenge "Mushrooms in a pizza do not get fully cooked when the dough is just right" and let them contribute with at least one idea for each section of a possible pattern. Then, we guide the conversation to incentivise participation, get the ideas flowing and maintain a focus for the session. Once we go through all the challenges, we give the participants space and time for solo contributions to help more reserved people express themselves. For online participants, we kept a video conference on for the duration of the activity, and we set up a Mural[2] to resemble the room's whiteboard.

For the confirmatory part of the focus group, our goal is to uncover the differences between what we found during the first part and the patterns we chose to tackle the monitoring challenges. We start by quickly going through the notes the participants created on paper and Mural to clarify the ones that we felt were harder to grasp. Afterwards, we clarify them to understand better what the person meant, thus reducing our interpretation bias. We briefly explain the adopted patterns for each monitoring challenge and gather the team's feedback on them and their adequacy to address the problem. Finally, we compare the patterns' critical ideas to the notes of the exploratory part of the focus group and search for discrepancies between them. These are discussed with the participants and can become part of the patterns' description.

To conclude this phase and the empirical study, we gather all the notes created during the focus group and analyse them. So far, these notes are the direct input of the participants with some minor adjustments on our part to correct grammar and spelling mistakes. During the analysis, we interpret the notes and extrapolate more information that we can use to improve the patterns. This improvement is made by going into a deeper comparison between the design patterns and the notes to understand their differences. The higher the number of differences is, the more the patterns can be improved. On the contrary, there is not much we can conclude if there are no differences, which we believe is an unlikely scenario. Lastly, we will revisit the patterns' descriptions based on these conclusions and improve their overall quality.

6.3 Project characterisation

To start this empirical study, we must familiarise ourselves with the project. Our main goal when doing so is to understand its architecture, organisation and most relevant monitoring issues. To

[2]Mural's official web page available at `https://www.mural.co/`.

that end, we cross-reference three sources to get an as complete and unbiased view of the project as possible. These sources are (1) an interview with the project's team leader, (2) its technical documentation and source code, and (3) the answers to a survey directed to the team members.

6.3.1 Team leader interview

In this section, we present the main takeaways of the initial interview with the team leader. Any information that could be used to identify the project or its client (e.g. project official name, client's name or company) is omitted for confidentiality reasons.

The interviewee explained that the project started as a tool to perform queries on employee data. It was a simple web application that provided many different filtering options for the data. Initially, the data were not specific to the project since they were consumed directly from the client's SAP system. As the solution evolved, it started to have proprietary data and a dedicated database. At the time of the interview, the project now utilises both internal and external data. It has grown to accommodate a succession planning module that calculates the line of succession for a specific role within the company, and a career management module that allows the employees to indicate future job preferences within the company.

The project team comprises one project manager, one quality assurance professional and two to five software developers. At the time of the interview, there were two developers currently working on the project, so the team had four members. The project is currently under active development, with new features being added regularly. The team is also responsible for its maintenance, fixing issues as new tickets arrive with user complaints. The application is accessed daily by around 160 users.

Afterwards, we questioned the team leader about the project's technical characteristics. He declared that it is built upon ASP.Net, a web development framework for .Net and programmed using C#. The database is implemented using SQL Server, and the front-end single page application (SPA) is built with Angular. The system features two distinct services:

1. The **monolithic application** that bundles together three internal services — SPA, API and Identity Server — responsible for the front-end, back-end and authentication, respectively; these modules communicate through HTTP and are always deployed together due to cost restrictions; moreover, the application follows an MVC architectural pattern.

2. The **background jobs** for computationally intensive calculations and operations; this service is updated every time the external data source changes, so it changes more often than the previous service and has to be deployed separately so its updates do not impact the monolithic application's availability.

Regarding development practices, the team uses Git as a version control system and has a shared repository hosted in Azure DevOps. The deployment is automated through pipelines in Azure DevOps that deploy the application to two separate Azure App Services, one for each of the previously mentioned services. The application is containerised using Docker and runs on Linux-based images.

When asked about monitoring practices, the team leader explained that error detection is done by the user. When something goes wrong, the product owner sends an email to the team with an error report which is then analysed to fix the problem. On the other hand, errors in the background jobs are found by regularly (almost daily) checking the message queue and looking for errors or unprocessed requests. Afterwards, for both services, the team troubleshoots the error by checking the logs generated around the time the occurrence happened and manually going through the log file to find the cause.

In addition, the interviewee informed that they use logs and out-of-the-box metrics, i.e. metrics that the cloud computing service provides by default on its hosts. Both services produce logs that are structured using SeriLog [3]. The logs are locally aggregated in files on each service's file system. There is one file for information level logs and another one for error level logs. As previously mentioned, the team manually checks the logs when there is an issue in production.

Regarding metrics, Azure automatically collects infrastructure metrics from the App Services, which the team cannot check because the Azure plan where the production environment runs is managed by the client and, consequently, not accessible by Konkconsulting's team. In addition, the team collects application-level metrics, more precisely, who logged in on the application and how many times they did it per day. This information is sent to the product owner from the client company. When asked about tracing, remediation and health monitoring practices, the interviewee said the project did not implement any of these practices.

Finally, we asked the team leader about the project's main monitoring challenges. He explained that there are errors that users do not detect and are, consequently, not reported. Moreover, when background processes fail, the team is not notified, requiring periodic inspection of the execution results. The interviewee believes the project needs alerts to cover both of these cases. In addition, he told us that it is sometimes difficult to go through and find the correct stack trace in the log files. Thus, the team leader wants a tool to parse and query the logs. Lastly, the interviewee stated that the project lacks dashboards to overview system metrics, allowing them to perform health checks more often.

6.3.2 Documentation and source code analysis

To further validate the information extracted from the interview with the team leader, we read the project's technical documentation and analysed its source code. Most of what we found corroborates the information from the interview. Nonetheless, the documentation goes a bit further to detail certain aspects of the solution that we believe to be useful to cross-reference with previous information and to understand some system intricacies. From the source code, we searched for existing configurations and usage of monitoring practices.

Most of the architectural design previously explained is also part of the documentation. However, there is greater detail over the four components that make up the system. These are:

[3] Official SeriLog documentation available at https://serilog.net/

1. **Front-end browser-based web application**, a responsive web interface for desktop and mobile implemented using an Angular client-side JavaScript framework.

2. **Server-side data API** that implements REST/OData HTTP endpoints for pure data exchange in JSON and XML format; the back-end API is consumed by the front-end browser-based web application; the data API is secured with OAuth 2.0 tokens —- bearer authentication — that carry authorisation grants provided by the project's security profiles.

3. **Server-side background processing**, performing asynchronous and scheduled tasks, like data ingestion and resource-intensive operations.

4. **Persistent relational data store**, to store parsed employee and organisational master data.

The document goes into further detail regarding each of these components, but we do not consider it relevant for the characterisation. These components represent a more logical perspective on the system. In other words, we can consider these components as the logical services used by the system, while the two services that were described in the previous section are closer to units of deployment. This means that we are not dealing with a microservice architecture, but with a service-oriented architecture bundled into a monolithic architecture. Newman [64] calls this case a distributed monolith.

According to the document, there are three deployment environments:

1. **Development** – Used by the development team to implement new features and conduct alpha tests with demo data.

2. **Quality** - Used by the development team to conduct beta tests with quality data.

3. **Production** – Used exclusively by end-users with production data.

The documentation provides more information about logging practices. It states that there is an Azure storage account to hold infrastructure logs because the web application, being on a shared PaaS service, should not and cannot store logs on the web application server. Moreover, the document asserts that infrastructure logs never carry sensitive information. We complemented this information by looking for logging calls in the source code. According to ASP.NET's documentation, there are six log writing levels: Information, Error, Warning, Trace, Debug and Critical [53]. Each log level has a corresponding log call, so we used Visual Studio Code to search the project for these calls with the following regular expression:

```
\.Log(Information|Error|Warning|Trace|Debug|Critical)
```

We excluded the *src/Mocks* folder because we realised that the classes inside it are used for development purposes only, even though the documentation does not explain its purpose. These classes spin up mocks of other external services the application uses which are not accessible in a local environment. Since they are not deployed in production, we excluded them from our monitoring analysis. The search results are shown in Figure 6.1. We found twenty-nine occurrences

spread through fifteen files in three main folders: *src/BaseLibs*, *src/Services* and *src/Web*. According to the technical documentation, the first folder contains the base classes of the application, the second folder holds the service layer of the application, and the last folder contains the web layer of the application. We dug a bit deeper into the *BaseLibs* folder and we concluded that most of the logic for the background jobs service is located there. Therefore, it makes sense that nineteen of the twenty-nine logging calls are inside this folder.

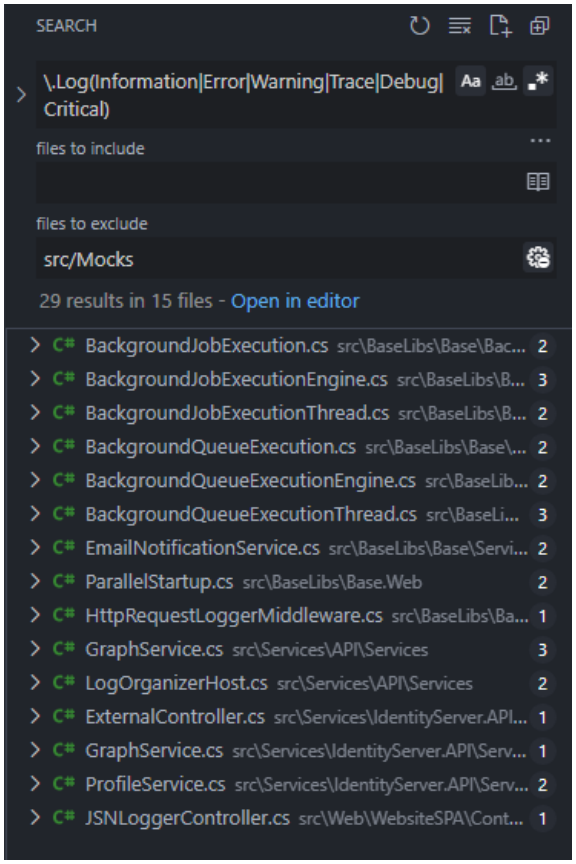


Figure 6.1: Search results for logging calls in the project source code. There are a total of twenty-nine calls spread across fifteen files, excluding the *src/Mocks* folder. The used regular expression can be seen at the top.

Finally, we confirmed the usage of Serilog to format the produced logs. Its configuration is written in a configuration file under the *src/Web/WebsiteMonolithic* package, shown in Listing 6.1. This package is the monolithic service that merely initialises all other services and bundles them together for deployment. Part of the initialisation process consists of propagating Serilog's configuration to the other services to make them all use the same logging format. The configuration sets two rules: one to log everything to the file *AppLogFiles/<ProjectName>_log.txt* and another to log just the errors to the file *AppLogFiles/<ProjectName>Error_log.txt*. Apart from the destination, both rules share the same properties. Most importantly, a log template guarantees that every log has a timestamp, log level, context to track it back to its source, and the actual content of the log

message. The template is configured for each file rule in the *outputTemplate* property, which can be seen in Listing 6.1.

```
"Serilog": {
  "MinimumLevel": {
    "Default": "Warning"
  },

  "WriteTo:0": {
    "Name": "File",
    "Args": {
      "outputTemplate": "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {
          SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}",
      "path": "AppLogFiles/<ProjectName>_log.txt",
      "rollingInterval": "Hour",
      "fileSizeLimitBytes": "1000000",
      "rollOnFileSizeLimit": "true",
      "retainedFileCountLimit": null,
      "shared": "true",
      "flushToDiskInterval": "00:00:01"
    }
  },

  "WriteTo:1": {
    "Name": "File",
    "Args": {
      "outputTemplate": "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {
          SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}",
      "path": "AppLogFiles/<ProjectName>Error_log.txt",
      "restrictedToMinimumLevel": "Error",
      "rollingInterval": "Hour",
      "fileSizeLimitBytes": "1000000",
      "rollOnFileSizeLimit": "true",
      "retainedFileCountLimit": null,
      "shared": "true",
      "flushToDiskInterval": "00:00:01"
    }
  },

  "Enrich": [
    "FromLogContext"
  ]
}
```

Listing 6.1: SeriLog configuration from file appsettings.json inside the monolithic service folder.

6.3.3 Team member survey

Finally, as explained in Section 6.2.1, we surveyed the project's team to confirm what we had found and guide the pattern selection and implementation phase. Of the seven people that had recent contact with the project, only four had gone through the issue-fixing process at least once. Hence, there were only four possible surveyees, and all of them replied to the survey. The answers show that the respondents have different levels of experience with the project: two have one to three years, one has three to six months, and the last one has less than three months. From this point onward, we refer to the two people with one to three years of experience with the project as the experienced members.

All the participants have full-stack developer responsibilities within the project. Additionally, the experienced members also have research and development roles. One of the experienced members answered with the lowest scale to all the questions regarding project knowledge. We believe this was a mistake, but since we guaranteed total anonymity, there was no way for us to talk to the person to correct the answer. Assuming that the person wanted to reply with the highest scale due to their years of experience with the project, we can conclude that both experienced members are completely aware of the project except for its monitoring infrastructure. For this specific characteristic, every surveyee replied with the medium scale (i.e. *Neutral*) even though there was a *Not Applicable* option. This neutrality leads us to think that there are monitoring practices and tools in place, but the team is not confident enough to state that they understand them thoroughly. The non-experienced members are the least familiar with the deployment strategy, followed by the production environment and monitoring infrastructure. This conclusion is not surprising since the non-experienced members only have developer roles within the project, so they are unaware of the production details.

From the next question, we conclude that the most used method to detect a bug/issue in production is through user complaints, followed by periodically checking log files (see Figure 6.2). When finding the root cause, as portrayed in Figure 6.3, the most adopted solution is to repeat user actions in a debug environment. These results corroborate what we discussed in Section 6.3.1, increasing our confidence in our conclusions from the interview. In addition, we notice that experienced members use the same five methods for finding root causes: log files, traces of execution, user reports, exceptions and repeating the user actions in a debug environment. We believe this is due to their experience with the project, which empowers these team members to extract information from more places than the others.

We can not extract any solid conclusion from questions 6, 7, 8 and 9 due to the low amount of answers. Questions 6, 8 and 9 are optional because they are mainly oriented to the production environment, thus not suited for every surveyee. There are few and widely dispersed answers to these questions, so we cannot conclude anything with certainty. Question 7 is mandatory, so we have four replies, as shown in Figure 6.4. Nonetheless, there is not much we can conclude from this question because the answers are somewhat dispersed.

We asked the participants their level of agreement regarding how quickly and easily the team

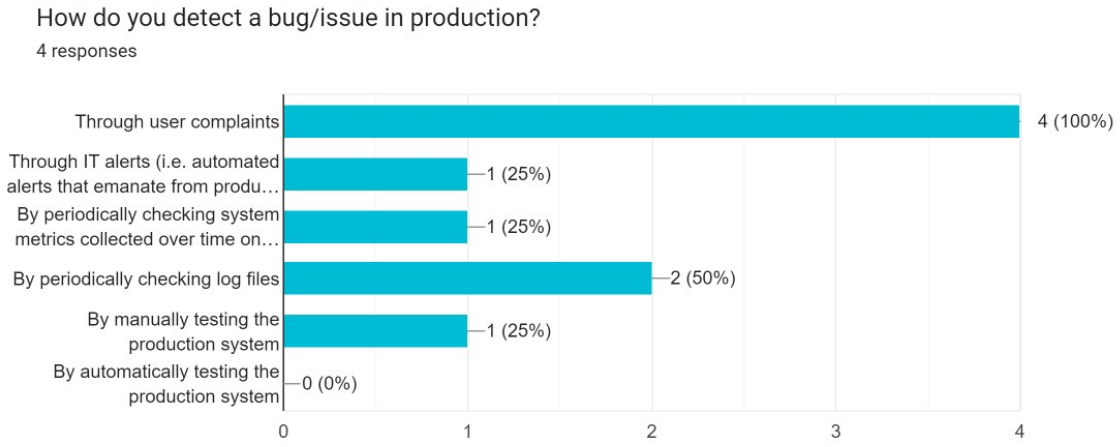


Figure 6.2: Results for question 4 of the team member survey. The first and fourth options are the ones with more votes.

could accomplish a set of activities. These activities mapped directly to each pattern's problem and were presented to the surveyees in the following nine questions:

Q10 Understand the application's code-level performance.

Q11 Know the state of the application's infrastructure.

Q12 Notice when a service unexpectedly becomes inoperative or extensively degraded and needs recovery.

Q13 Notice when a running service becomes unable to process requests.

Q14 Detect bugs before they impact end-users.

Q15 Manage and store the application logs.

Q16 Read and understand logs.

Q17 Correlate production problems with the deployment that caused it.

Q18 Trace a failure seen in the front-end service to the service that caused the failure.

Q19 Track exceptions and find their causes.

The answers to these questions are shown below in Figure 6.5. As seen in the figure, the most voted scale is *Agree*, and the least voted one is *Strongly Disagree*, so we conclude the participants are overall optimistic regarding the team's ability to tackle the mentioned monitoring challenges. From what we had analysed in the earlier parts of the project characterisation, we did not expect such confidence since the project only had three or four well-established monitoring practices. **Q14** has the lowest level of agreement, followed by **Q15** and **Q16**. To our surprise, these questions are the only ones with a value lower or equal to the *Neutral* scale. **Q19** has the highest level of agreement (between the *Agree* and *Strongly Agree* scales), which can be explained by their experience and frequency of dealing with exceptions, even though they do not have any dedicated solution to handle them.

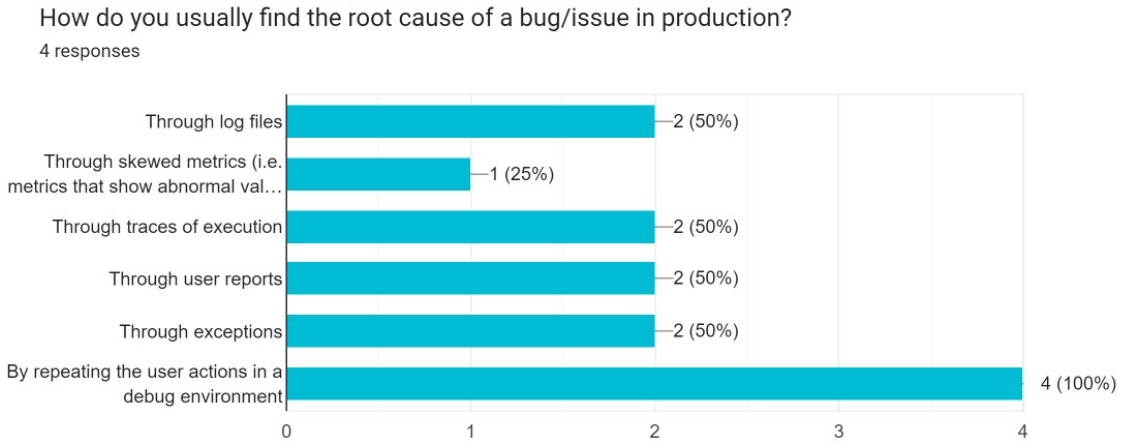


Figure 6.3: Results for question 5 of the team member survey. The last option was voted by every team member. The first, third, fourth and fifth options were voted only by experienced members.

In question 20, we asked the participants for their level of agreement regarding the level of adoption of the following monitoring practices in the project:

P1 Collection of business and performance metrics at the application code level.

P2 Collection of operative system and infrastructure metrics from the production environment.

P3 Periodic pings of a specific endpoint to ensure the system is healthy (i.e. not crashed or extremely slow).

P4 Periodic pings of a specific endpoint to ensure the system is ready (i.e. it is ready to accept and fully able to process user requests).

P5 Running tests against the production system.

P6 Record of user activity and system changes for auditing purposes.

P7 Tracking of deployments and changes to the production environment.

P8 Tracing the flow of a request through the system.

P9 Sampling and prioritisation of logs.

P10 Standard format for logs in the system.

P11 Tracking and aggregation of exceptions.

The answers to this question reveal an overall lower level of confidence in comparison with questions 10 to 19. The responses, depicted in Figure 6.6, are mainly *Neutral* and there are more *Disagree* and *Strongly Disagree* answers (16 in total) than their positive counterparts (12 in total). **P3** and **P9** are the lowest-ranked practices with no positive-scale answers. **P4** is voted with both *Strongly Disagree* and *Strongly Agree* scales, which can be caused by a poor description of the practice and consequent confusion regarding its interpretation. On the other hand, **P7** and **P8** are the highest-ranked practices with no negative-scale answers. The overall lower degree of confidence for question 20 was expected because there are few monitoring practices adopted in

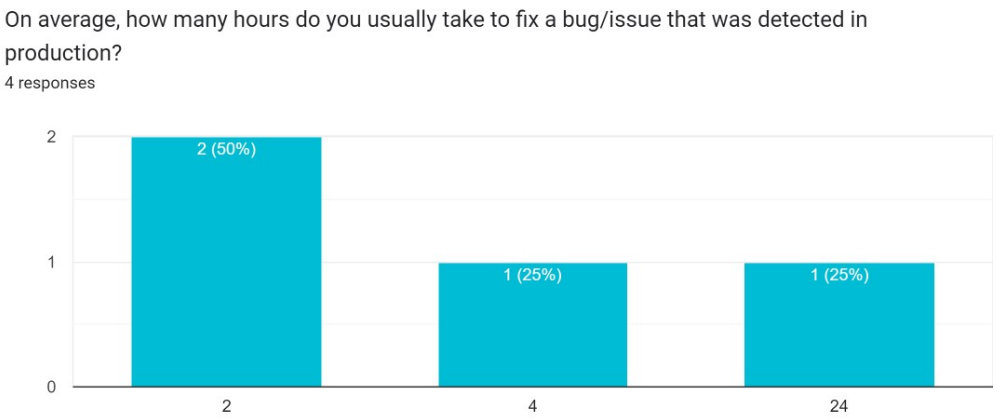


Figure 6.4: Results for question 7 of the team member survey. Four replies were spread among the 2, 4 and 24 hours marks.

the project. However, we also expected a stronger match between the lowest-ranked 10 to 19 questions and the current question. In other words, we expected that the problems the team felt less able to solve would map to missing monitoring practices that could solve said problems.

The remaining questions are more focused on guiding the implementation process. The answers to question 22 reveal that most problems occur in the application code (4 votes) and the Database (4 votes). One participant replied with *Errors in data itself or DataLake sync* which was not initially part of the checklist. From question 23, we conclude that the most common symptoms of a problem in production are database query slowness (3 votes), exceptions and errors (3 votes), and deadlocks and synchronisation issues in the application code (3 votes). These symptoms are natural candidates for system metrics we should consider for the implementation process. Finally, the answers to question 24 reveal that the participants have little experience with monitoring tools because each person only voted on a single tool. In addition, the experienced team members state that they have experience with Azure Monitor, which can be motivated by their contact with the project's deployment strategy and production environment.

6.3.4 Discussion

To conclude this section, we summarise the main takeaways from the project characterisation phase, starting with the project's context. The project is developed by Konkconsulting for a customer, which imposes some access restrictions. It is currently under active development with new features being implemented. It is developed and maintained by a small team of four people and used by around 160 daily users at the client company. Moving on to the project's more technical aspects, it follows a layered and service-oriented architecture.

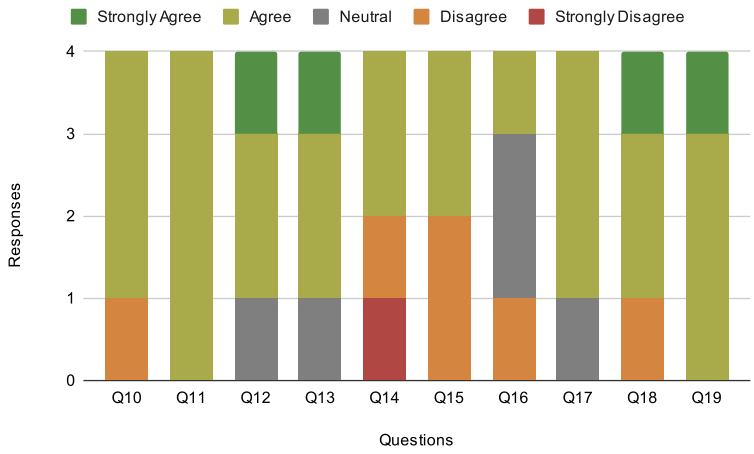


Figure 6.5: Results for questions 10 to 19 of the team member survey.

Nonetheless, there are only two independently deployed services, so it remains pretty monolithic, in our opinion. It is hosted on the cloud, more precisely, on Microsoft's Azure cloud computing services. Being on the cloud is a crucial aspect of its suitability for this empirical study. Otherwise, the project would not be within the scope of this dissertation's patterns. The application is deployed in two distinct Azure app services: one for the monolithic web application and the other for a service that performs background tasks when new data reaches the database. The former encompasses three smaller services that represent logical components of the system — its API, the authentication and authorisation API and a front-end single-page application — deployed together in a single app service for cost savings. The application also depends on external services: the client's database and a proprietary database. Lastly, the project's technical document declares there are three deployment environments — development, quality and production — in which the application is deployed. Knowing about these environments can help guide the adoption of some patterns (e.g. synthetic testing) that can be applied to the quality environment in the first place and then, once the team sees the value in them, move on to production.

Regarding monitoring practices, from the interview and survey, we conclude that error detection is mainly done by the end-users. When it comes to the background tasks, since they are not accessible to the users, errors are regularly checked by the developers that look for errors in the queue's state logs. Troubleshooting of production issues relies mainly on two tasks carried out by the development team: analysing logs and repeating user actions to recreate the error. The latter is possible due to user error reports that the client files once an issue with the system arises.

Logs are very present across the system. We found 29 calls of logging generation methods in 15 different files while exploring the source code. According to the technical document, they never carry sensitive information, so we have not found any additional security measures around log files. The team leverages standardised logging to make logs easier to read and understand for all members. To do so, they use Serilog with a template that guarantees the presence of a

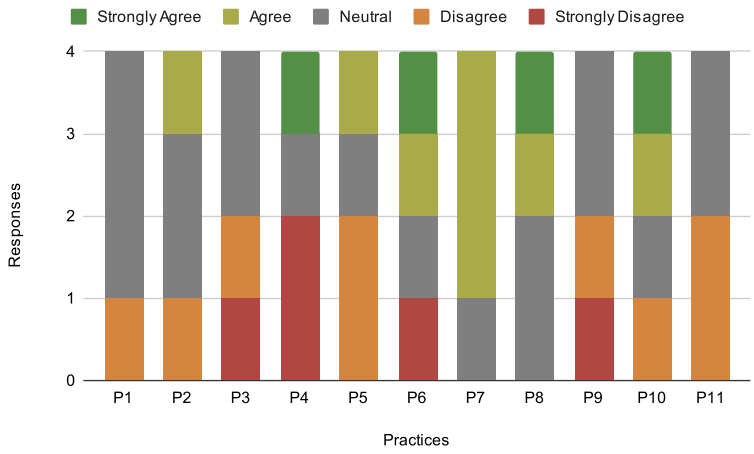


Figure 6.6: Results for each practice in question 20 of the team member survey.

timestamp, severity level, source context, and the actual message in every log generated by the services. The logs are saved in separate log files depending on their severity level. Because we are dealing with a monolithic application, these logs are generated and saved on the same service, so one could say that they are already aggregated.

In addition, we conclude that the team collects metrics on the application and infrastructure levels. However, the latter is only available to the customer because only they have access to the production environment's dashboards. In terms of application metrics, the team leader stated that they provide a report with all daily logins to the customer, so they are collecting that information in the monolithic service. That said, the project partially adopts the INFRASTRUCTURE METRICS and APPLICATION METRICS patterns. We say *partially* because there is no *Metrics Service* component to store and aggregate the data and allow the team to leverage that information. However, this is not the case for the background tasks. For this service, in particular, no metrics are being collected yet. Lastly, apart from metrics and logs, no other monitoring practices are applied in the project. Nonetheless, according to the survey results, the team members believe they are overall capable of dealing with the monitoring problems we identified in the design patterns, even though they are not implementing any of our proposed solutions.

When it comes to issues with monitoring, we conclude there are three main monitoring challenges (MC):

MC1. Logs are hard to read, and it is difficult to find stack traces in the log files

Due to the sheer amount of logs being generated in the system and the absence of an indexing and querying system, searching through logs is laborious and time-consuming. In addition, an exception's stack trace is hard to read because it gets lost between the logs in the file.

MC2. The lack of dashboards makes it difficult to understand the metrics collected from the monolithic service

The metrics being collected are not available in any visual tool, so the team cannot fully leverage the metrics' benefits. In other words, correlating metrics, knowing their evolution and using them strategically become much more challenging without a dashboard.

MC3. There are no metrics on the background tasks service to keep track of errors and processing time

The team leader pointed out that alerts were required for the team to know when an error occurred in the background tasks service. However, alerts will be much harder to implement when no metrics are collected for said service. Thus, as further explained in Section 6.4.1, we believe the lack of metrics that convey this service's status is the underlying problem.

There are certainly more monitoring aspects in the project that could be improved. However, due to time constraints, we focus solely on these three challenges during the pattern selection and implementation phase. The team member survey helped us conclude the following guidelines for the next phase:

- The application code and database are probably the best places to add instrumentation since these are the source of most problems.
- The most relevant metrics to collect are the ones that convey database query slowness, the number of exceptions and errors, and deadlocks and synchronisation issues in the application code since these are the most common symptoms of a problem.
- There is not much previous experience with monitoring tools among the team members; thus, all tools can be equally considered for adoption.

6.4 Pattern selection and implementation

Now that we understand the project's context and challenges, we can move on to the next phase of this empirical study — the pattern selection and implementation. As explained in Section 6.2.2, this phase is carried out in two distinct parts. Its main goal is to evaluate whether each pattern's context, problem and solution work well together. In other words, we evaluate if the solution applies in the given context, if the solution is adequate to solve the problem, and if the context should be refined to guarantee the solution does apply. Moreover, it is a way to get a deeper understanding of the project's source code and challenges.

6.4.1 Pattern selection

The selection of patterns is based on the problems we are solving in the project. However, the way the challenges were presented to us is far from being a direct match with the patterns' described problems. Therefore, for each identified monitoring challenge (MC), we extrapolate underlying problems based on our opinion and experience with the project.

MC1. Logs are hard to read and it is difficult to find stack traces in the log files

The project already adopts STANDARD LOGGING (see Section 5.5) through a log formatting tool that the team uses to ensure a standard format for every application log (see Listing 6.1). Theoretically, this pattern already increases log readability, but other conditions can make them hard to read.

In the current scenario, we believe this challenge can have three underlying problems: (1) there are too many logs being generated, (2) they are too verbose, and (3) the logs are hard to query. We think the third is more likely to be the source of the problem because of how the team leader described the challenge. During the interview, he stated that the logs are merely text dumped into a log file. Moreover, he mentioned the need for an indexing and querying tool to facilitate information extraction from the logs. We conclude that the current challenge is caused by the difficulty in querying the logs because these are not parsed and indexed.

That said, this problem resembles the QUERY ENGINE problem description: "How do you view all of the different log entries that each service produces and make sense out of the structure of what is being logged? How can you extract information from all of the data in these log entries?" [14]. According to the pattern's context, it applies to everyone that is "building a system made up of many cooperating services" [14], including services built by themselves or third parties. The description depends on the reader's interpretation since *many* may mean different things to different people. In our concrete scenario, there are two deployment units but a total of four logical services, a persistent database hosted in the cloud provider and an external data lake with the client's data. Thus, we believe that the number of services is significant enough to justify the pattern adoption.

Finally, the QUERY ENGINE solution suggests the usage of "a query engine to treat the set of logs as a database" [14], so the team can query the logs to extract relevant information. This solution seems simple and effective for the current problem, and we consider it feasible in the current context. Therefore, we conclude that the QUERY ENGINE pattern should be enough to solve this challenge.

MC2. The lack of dashboards makes it difficult to understand the metrics collected from the monolithic service

As we conclude in Section 6.3.4, the monolithic service partially adopts the INFRASTRUCTURE METRICS and APPLICATION METRICS patterns. In the same section, we also explain that the project lacks a *Metrics Service*. In other words, it lacks a centralised service that collects and aggregates the metrics data. This component of the pattern's solution provides a real-time overview of the metrics, which most of the time translates to a dashboard.

Therefore, the team already collects metrics and stores them, but as the challenge description already hints, they still miss the dashboard. We believe that the lack of visualisation over the metrics is the underlying problem in this case. This problem has a small scope in the sense that

it just pertains to the metrics visualisation. The design patterns we found and wrote do not tackle such detailed problems.

Nonetheless, we think this problem is encompassed in the description of two design patterns. On the one hand, we have the INFRASTRUCTURE METRICS pattern that seeks a way for the team better to understand the state of their application's infrastructure. On the other hand, we have the APPLICATION METRICS pattern that tackles the challenge of getting "an overview of how the application is performing". Since these patterns are already partially adopted in the project, it is clear that their contexts match the project's background and that their solutions are suitable for the current context.

To summarise, we think that adding the *Metrics Service* component that the implementation lacks will be enough to solve the current problem. Doing so implies that we fully adopt the INFRASTRUCTURE METRICS and APPLICATION METRICS patterns.

MC3. There are no metrics on the background tasks service to keep track of errors and processing time.

As we wrote down this challenge, we were already hinting at the solution. However, this might introduce some informality to the selection process, which we try to avoid by following the steps explained in Section 6.2.2. In the following paragraphs, we elaborate further on the train of thought that led to it.

Currently, the team periodically checks the state of the job queue to see if there are errors in the execution of background tasks. This service works separately from the monolithic service and is not generating any metrics. In addition, it is not a user-facing service, so there are no user complaints when a background job fails. Therefore, we boil down the problem to the lack of an automated way to report an error in the background tasks service. This issue may lead to errors lingering in the service for many days without notice.

As the team leader suggested during the interview, a possible solution for this problem is to have alerts set off automatically when there is an error in a background job execution. However, we think having alerts set off every time something fails is counter-productive. Some errors are caused by temporary malfunctions of the system or its environment (e.g. network slowness at the data centre, high RAM usage in the host caused by other hosts) and, thus, should not always be reported. Hence, we believe that alerts should be based on metrics so the team can precisely configure the threshold of errors or processing time that will trigger the alert.

We go the extra mile on elaborating the problem to conclude that, before configuring alerts, the team requires a real-time view of the background tasks service's performance. This problem resembles the one that the APPLICATION METRICS pattern tackles: "How can the team get an overview of how the application is performing and understand the emerging usage patterns?". This pattern's context mentions a cloud environment where "teams must be able to keep track of their system's performance" because "users expect things to answer quickly and fail rarely". We think these properties match the project's context and make the pattern applicable in the current scenario.

Lastly, the APPLICATION METRICS solution suggests the instrumentation of the application to gather performance metrics. Instrumenting the code can be a problem in specific projects due to the added complexity of metric generation entwined with business logic or the impossibility of changing the source code (e.g. legacy systems). In our case, the service's source code can be freely modified, so adding instrumentation is not an issue. Moreover, the generated metrics should be collected "in a centralised service that provides aggregation and visualisation". This is the responsibility of the *Metrics Service* component described in the design pattern. As explained in the discussion around **MC2**, the monolithic service also lacks that component. We believe that aggregating and visualising the metrics from both application services in a single *Metrics Service* makes the consumption of metrics data more straightforward. In other words, the *Metrics Service* implementation to tackle the previous challenge could be extended to have the background tasks' data. Thus, we feel that adopting the APPLICATION METRICS pattern is a suitable solution for the problem at hand.

As we end the pattern selection part, we conclude that each challenge is caused by an underlying monitoring problem. These problems are mentioned in design patterns proposed by this dissertation or related work, from which we selected the following list of patterns: QUERY ENGINE to tackle **MC1**, APPLICATION METRICS to tackle **MC2** and **MC3**, and INFRASTRUCTURE METRICS to tackle **MC2**. We have ensured that all the selected patterns' context matches the project's context and that the solutions are feasible and adequate to solve the monitoring challenges. Thus, we can move on to the implementation of these design patterns.

6.4.2 Pattern implementation

To implement the list of design patterns that resulted from the last section, we start by looking for existing monitoring tools that match the roles described in each pattern's solution. Afterwards, we dive into the source code to implement the tools and, consequently, the selected design patterns. This part of the study aims to deepen our understanding of the project and its source code and provide first-hand experience with the project's monitoring problems to help us prepare a more thorough focus group for the next phase of the empirical study.

6.4.2.1 Tool selection

The criteria for tool selection are explained in Section 6.2.2. We highlight the chosen tools in bold to make them easier to identify in the text and provide links to their official web pages in the footnotes.

Firstly, the Known Uses section of the QUERY ENGINE pattern mentions the combination of LogStash with Elasticsearch [14]. The former takes the role of *Log Aggregator*, which is not what we need, while the latter "allows full-text search of the collected entries" [14]. Further research online revealed that **Elasticsearch**[4] is "a distributed, RESTful search and analytics engine" [29]

[4]Official Elasticsearch web page available at https://www.elastic.co/elasticsearch/.

that stores data centrally. This tool can be used "to treat the set of logs as a database" [14], so it is a way of adopting the QUERY ENGINE pattern. Nonetheless, we found that this tool is often used in cooperation with **Kibana**[5], which provides a user interface (UI) for enhanced visualisation of the data saved in Elasticsearch, alongside the possibility to query the data directly from the UI. Hence, we opted to use both tools to make it easier for the team to query the system's logs. Combining the two tools is enough to adopt the QUERY ENGINE pattern and tackle **MC1**.

For the APPLICATION METRICS pattern, we start by looking into **Prometheus**[6], which is mentioned in the pattern's Known Uses section. This tool "collects and stores its metrics as time series data" [70] by periodically pulling the metrics data from an API endpoint that is previously configured. However, its visualisation features are limited, so we followed the pattern's example and used Prometheus together with **Grafana**[7]. The latter provides a customisable dashboard with various graphs available and effortless integration with many metrics storage tools, including Prometheus. The combination of these two tools fulfils the role of *Metrics Service*, which is what the team needs to solve **MC2**. Therefore, the adoption of the *Infrastructure Metrics* and *Application Metrics* patterns to address that challenge can be achieved by using Prometheus and Grafana together.

On the contrary, to tackle **MC3**, we still lack something that generates and exports the metrics (i.e. the *Metrics Exporter* described in the pattern's solution) for the background tasks service. The design pattern did not mention any tool for this specific task, so we set out to find instrumentation libraries for ASP.NET projects. We found a tutorial video [19] that suggests the usage of Prometheus and Grafana alongside an open-source .NET library called **AppMetrics**[8]. This library automatically generates basic metrics for the application with very little instrumentation, enough to fulfil our current needs. By using the three tools — Prometheus, Grafana and AppMetrics — together, we can entirely adopt the APPLICATION METRICS pattern for the monolithic service.

Concluding, after analysing the design patterns' descriptions and some possible tools, we decided to use the following tools: Elasticsearch and Kibana to implement QUERY ENGINE; AppMetrics, Prometheus and Grafana to implement APPLICATION METRICS in the background tasks service; and Prometheus and Grafana to complement the partial adoption of the INFRASTRUCTURE METRICS and APPLICATION METRICS patterns in the monolithic service. With the set of tools defined, we move on to their implementation in the project.

6.4.2.2 Implementation details

In this section, we describe the implementation process and its details. This part of the empirical study feeds into the focus group phase because we get on a common ground with the team members. In other words, we get a better view of the project by experiencing the possible solutions to its monitoring challenges first hand.

[5] Official Kibana web page available at https://www.elastic.co/kibana/.
[6] Official Prometheus web page available at https://prometheus.io/.
[7] Official Grafana web page available at https://grafana.com/.
[8] Official AppMetrics documentation available at https://www.app-metrics.io/.

The project's context is explained in Section 6.3. Nonetheless, we reiterate on the most relevant aspects for the implementation. The company did not want to disclose R&D work with the client while it was not clear if it would bring them value. Therefore, we worked on a copy of the project's original repository of March 2022. The implementation was done on a local machine and did not make into any deployment environment. We worked with Visual Studio Code[9] and Visual Studio Community 2022[10] to edit source code and manage NuGet[11] packages. The project is built on ASP.Net Core 2.1, which is an old version of the framework and imposes some versioning restrictions. Finally, the project's Docker containers are used solely for deployments, so the local development environment is not containerised.

The selected tools are containerised, so having a containerised environment would make their implementation easier. Thus, our first step was to create the containers to resemble the production environment. We used Docker Compose[12] to orchestrate the containers for local development. Apart from the application's deployment units (the monolithic and background tasks services), we had to containerise a MySQL database and an Azure Active Directory Federation Services (ADFS) mock application. These project dependencies are provided by the cloud in the production environment, but not available for the local development environment. The final Docker Compose file is shown in Listing 6.2.

```
1  version: "3.5"
2  services:
3    ts-db:
4      image: "mcr.microsoft.com/mssql/server"
5      user: root
6      hostname: "localdev-sqlserver"
7      ports:
8        - "1433:1433"
9      environment:
10       SA_PASSWORD: "Local_Dev2022"
11       ACCEPT_EULA: "Y"
12     volumes:
13       - ./dev/Database/data:/var/opt/mssql/data
14       - ./dev/Database/log:/var/opt/mssql/log
15       - ./dev/Database/secrets:/var/opt/mssql/secrets
16
17   ts-bgtasks:
18     build:
19       context: .
20       dockerfile: ./dev/BackgroundTasksImage/Dockerfile
21     depends_on:
22       - ts-db
23
```

[9] Visual Studio Code official page available at https://code.visualstudio.com/.
[10] Visual Studio Community 2022 official page available at https://visualstudio.microsoft.com/.
[11] NuGet official page available at https://www.nuget.org/.
[12] Docker Compose official documentation available at https://docs.docker.com/compose/.

```
  ts-mockadfs:
    build:
      context: .
      dockerfile: ./dev/MockADFSImage/Dockerfile
    ports:
      - "44390:44390"
    depends_on:
      - ts-db

  ts-monolithic:
    tty: true
    build:
      context: .
      dockerfile: ./dev/WebsiteMonolithicImage/Dockerfile
    ports:
      - "2222:2222"
      - "80:80"
    volumes:
      - ./src/Web/WebsiteMonolithic/AppLogFiles:/app/AppLogFiles
    depends_on:
      - ts-mockadfs
networks:
  default:
    name: localdev_network
```

Listing 6.2: docker-compose.yml file for the local development environment.

We started with the adoption of the APPLICATION METRICS and INFRASTRUCTURE METRICS pattern in the monolithic service. The first step was to add metrics generation with the AppMetrics library. Its installation required four NuGet packages: *App.Metrics.AspNetCore*, *App.Metrics.AspNetCore.Endpoints*, *App.Metrics.AspNetCore.Tracking*, and *App.Metrics.Formatters.Prometheus*. We used version 3.0.0 of each of these packages for compatibility with the project's .Net version. Implementing the library involved two things. First, we added a call to *UseMetricsWebTracking* and *UseMetricsEndpoints* to the web host builder in the monolithic project so the library generated the metrics for web requests and exposed them on two endpoints: a */metrics* endpoint that exposes metrics in Prometheus' Protobuf format, and a */metrics-text* endpoint that exposes the same metric in a text format. This code is depicted below in Listing 6.3.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
        .UseMetricsWebTracking()
        .UseMetricsEndpoints(options => {
            options.MetricsEndpointOutputFormatter =
                new MetricsPrometheusProtobufOutputFormatter();
            options.MetricsTextEndpointOutputFormatter =
```

```
                new MetricsPrometheusTextOutputFormatter();
        })
        .UseStartup<Startup>()
        .UseSerilog(SerilogConfigurationDelegate, true);
}
```

Listing 6.3: CreateWebHostBuilder method from the Program.cs file of the WebsiteMonolithic project.

Secondly, we added a metric collection middleware to every service by adding calls to *AddMetrics* for each one of them. The method calls were added to the *ConfigureServices* function of the *WebsiteMonolithic* project. This project corresponds to the monolithic service. Since all web application services are being configured in the monolithic project's startup file, this was the only place we had to instrument. The code is shown below in Listing 6.4.

```
public void ConfigureServices(IServiceCollection services)
{
    this.LogAllEnvironmentVariables(this.logger);
    this.LogAllConfiguration(Configuration, this.logger);

    services.AddMetrics();
    services.AddMetricsEndpoints();
    services.AddSingleton<API.Startup>().AddMetrics();
    services.AddSingleton<IdentityServer.API.Startup>().AddMetrics();
    services.AddSingleton<Website.SPA.Startup>().AddMetrics();
}
```

Listing 6.4: ConfigureServices method from the Startup.cs file of the WebsiteMonolithic project.

With AppMetrics configured, the next step was to configure Prometheus to scrape the endpoints and store the metrics data. Setting it up was pretty straightforward using its Docker image[13]. We created a different Docker Compose file (see the service *prometheus* in Listing 6.7) for the monitoring stack to keep all tools isolated from the application. Nonetheless, we configured these containers' *default* network to be the same as the application's network. The Prometheus' configuration we used is shown below in Listing 6.5. We used a simple configuration that instructs the Prometheus Server to ping the */metrics-text* endpoint of the monolithic service's container every fifteen seconds. We were not able to configure Prometheus to use the */metrics* endpoint because the metrics were not showing up in the tool's UI and there was no indication of error. Due to the small number of metrics being collected, we believe using the simpler text format is still efficient. From this point on, Prometheus will pull the system's metrics and store them in a time-series with the project's name.

```
global:
```

[13] Official Prometheus' docker image available at https://hub.docker.com/r/prom/prometheus.

```
  scrape_interval: 15s
  external_labels:
    monitor: 'codelab-monitor'

scrape_configs:
  - job_name: '<ProjectName>'
    static_configs:
      - targets: ['ts-monolithic:80']
    metrics_path: /metrics-text
```

Listing 6.5: Prometheus configuration file. The project's name was redacted for confidentiality reasons.

To conclude the adoption of the metrics patterns, we lack Grafana for data visualisation. This tool also has a Docker image[14], so implementing it was just a matter of adding another service to the Docker Compose file for the monitoring stack (see the service *grafana* in Listing 6.7). Configuring Grafana was quite simple as well because it already provides integration with Prometheus. We created a data source that pointed to the Prometheus' container and displayed its data in a dashboard created from a template provided by the AppMetrics library[15]. The final result is depicted in Figure 6.7 and is a solution to **MC2**.

Figure 6.7: Screenshot of the Grafana dashboard that resulted from the adoption of APPLICATION METRICS and INFRASTRUCTURE METRICS in the monolithic service.

The adoption of the QUERY ENGINE pattern was straightforward as well. We started by adding Elasticsearch to the monitoring stack. The tool has a Docker image[16] available, so we added another service to the monitoring Docker Compose file (see the service *elasticsearch* in Listing 6.7). With the Elasticsearch node set up, we needed to redirect logs to it.

[14]Official Grafana's open source Docker image available at https://hub.docker.com/r/grafana/grafana.

[15]Template available at https://grafana.com/grafana/dashboards/2204

[16]Official Elasticsearch's Docker image available at https://hub.docker.com/_/elasticsearch.

As explained in Section 6.3, the team uses SeriLog to handle logging in the system. It is being used to flush logs to the console and to a file. However, the library also provides a sink for Elasticsearch through the *Serilog.Sinks.Elasticsearch* package. We installed the package with version 8.4.1 and added a SeriLog rule to the monolithic project's configuration file (see Listing 6.6). This rule instructs SeriLog to flush the logs to the Elasticsearch container we set up before.

```
{
    "Serilog": {
        "MinimumLevel": {
            "Default": "Debug",
            "Override": {
                "Microsoft": "Information",
                "System": "Information"
            }
        },
        "WriteTo:2": {
            "Name": "Elasticsearch",
            "Args": {
                "nodeUris": "http://elasticsearch:9200/",
                "autoRegisterTemplate": true,
                "numberOfShards": 2,
                "numberOfReplicas": 1,
                "indexFormat": "<ProjectName>-logs-localdev-{0:yyyy-MM}"
            }
        }
    }
}
```

Listing 6.6: SeriLog's configuration in the appsettings.json file from the WebsiteMonolithic project.

The logging format used in the project already relies on named parameters. These parameters, and the ones added by a log enrichment process done by SeriLog, are automatically indexed by Elasticsearch and turned into queriable fields. To make querying easier, we added a Kibana container[17] to the monitoring Docker Compose file. To configure this tool to get the data from our Elasticsearch node, we simply set the environment variables *ELASTICSEARCH_URL* and *ELASTICSEARCH_HOSTS* to point to the node's network address. Finally, Kibana fetches the data and provides an UI with all logs from the used index format in SeriLog's configuration (see Listing 6.6). Figure 6.8 portrays an example of how the dashboard can be used to sort through the logs. This concludes the implementation of the QUERY ENGINE pattern and solves **MC1**.

```
version: "3.5"
services:
```

[17] Official Kibana's Docker image available at https://hub.docker.com/_/kibana.

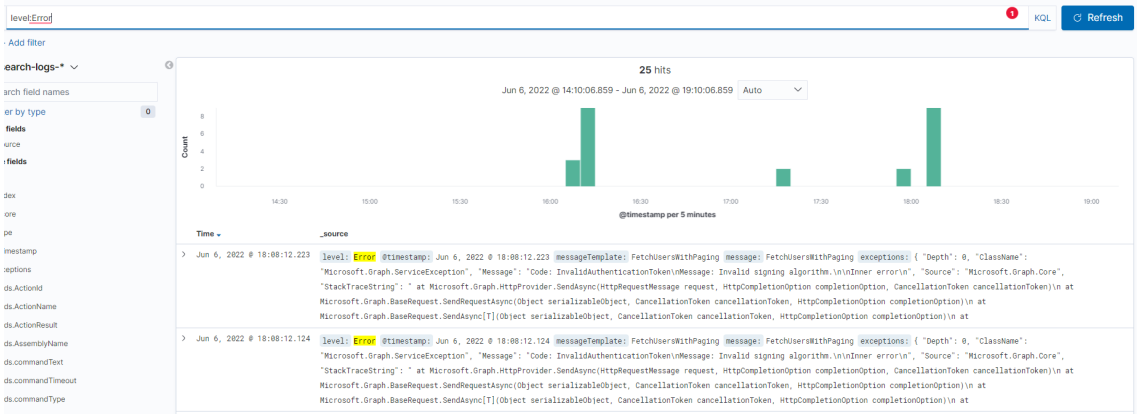


Figure 6.8: Screenshot of the Kibana dashboard that resulted from the adoption of QUERY ENGINE in the monolithic service.

```
3   prometheus:
4     image: prom/prometheus
5     ports:
6       - 9090:9090
7     volumes:
8       - prometheus_data:/prometheus
9       - ./dev/Monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
10
11  grafana:
12    image: grafana/grafana-oss
13    ports:
14      - 9080:3000
15    environment:
16      GF_INSTALL_PLUGINS: "grafana-piechart-panel"
17    volumes:
18      - grafana_data:/var/lib/grafana
19
20  elasticsearch:
21    image: docker.elastic.co/elasticsearch/elasticsearch:7.9.0
22    ports:
23      - 9200:9200
24    environment:
25      - node.name=es01
26      - cluster.name=es-docker-cluster
27      - cluster.initial_master_nodes=es01
28      - bootstrap.memory_lock=true
29      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
30    ulimits:
31      memlock:
32        soft: -1
33        hard: -1
34    volumes:
```

```
35        - elasticsearch_data:/usr/share/elasticsearch/data
36
37    kibana:
38      image: docker.elastic.co/kibana/kibana:7.9.0
39      ports:
40        - 5601:5601
41      environment:
42        ELASTICSEARCH_URL: "http://elasticsearch:9200"
43        ELASTICSEARCH_HOSTS: "http://elasticsearch:9200"
44      depends_on:
45        - elasticsearch
46
47  networks:
48    default:
49      name: localdev_network
50      external: true
51
52  volumes:
53    prometheus_data:
54    grafana_data:
55    elasticsearch_data:
```

Listing 6.7: docker-compose.yml file for the monitoring stack.

Lastly, we are still missing a solution to **MC3**. As explained in Section 6.4.1, this challenge required adopting the APPLICATION METRICS pattern to the background tasks service. Due to time constraints, we decided not to apply this solution because it required the exact steps we took to solve **MC2**. Nonetheless, we believe that the way we adopted the tools allows the team to easily extend them to encompass more services or even be transferred to other projects.

6.4.3 Discussion

This phase of the empirical study was an initial test for this dissertation's design patterns. From the pattern selection process we conclude that all the identified monitoring challenges, and their underlying problems, were covered by the pattern catalogue. The patterns we chose — QUERY ENGINE, APPLICATION METRICS, and INFRASTRUCTURE METRICS — were the ones that solved problems that matched the project challenges more directly, even though we would have liked to try out more patterns. Due to time constraints, we decided to focus just on these three that tackle the most relevant challenges with the project.

The pattern selection part was a way to test how the problem, context and solution of each pattern work together. We conclude that, for each of the chosen patterns:

1. The problem was easy to match with the project's challenges, so we believe its description is intuitive enough.

2. Its context matched with the project's context, so its problem is an issue in the described context.

3. The solution made sense to tackle the identified problem, so we think its solution is good for addressing the described problem.

4. The solution was applicable in the project's context, so its context adequately encompasses the solution the pattern proposes.

The pattern implementation part touched on different sections of the patterns. On one hand, the tool selection process helped us evaluate how easy it is to find appropriate tools to adopt the chosen patterns. We conclude that the Example and Known Uses sections of each pattern had enough detail to make us find the tools quickly. Thus, for now, we consider their descriptions good enough.

On the other hand, the implementation part allowed us to apply the patterns in practice. The description of the implementation details was very superficial because it was not our goal to explain thoroughly how to use each tool. Instead, we wanted to provide some background to make our conclusions easier to understand. The first being that all of the selected tools were containerised, so their implementation was quite straightforward. While searching for them, we found that there are plenty of monitoring tools available [64]. Hence, we conclude that adopting the patterns does not require much development effort for simple cases where these tools fulfil the project's needs.

In addition, we experienced the difficulty in reading and extracting information from the logs that was previously mentioned in Section 6.3. We had issues with SeriLog's configuration and felt the need to add additional logs to the system. However, finding those logs in the middle of the log files was hard and, most of the times, the remaining logs did not provide enough context or information to help us find the issue. Our lack of knowledge regarding the system's source code could be a reason for this problem to have emerged. Nonetheless, it helped us understand the challenge better and have first-hand experience with it. Moreover, by the end of this phase we have a better idea of how large the project is, what are its services and what is their role in the system, and that there are still monitoring capabilities that could be developed. All of these conclusions help us prepare a more contextualised focus group for next phase and give us a more informed view of the team members' comments that we get from that phase.

6.5 Focus group

In this section, we describe how the focus group happened and what were the results we got out of it. As explained in Section 6.2.3, this phase of the empirical study aims at getting feedback on our findings so far and getting new ideas to improve the selected patterns' descriptions.

6.5.1 Running the activity

In Section 6.2.3, we explain how we planned the focus group. We conducted the activity with six participants, three of whom were on-site in a meeting room, and the rest were online in a video conference. Thus, we used a hybrid format to get everybody involved. All of the participants were

Konkconsulting's employees at the time and, as such, they had a consulting background. The activity took around one and a half hours.

On the one hand, in the room, we hung the paper sheets on a whiteboard and the session's rules and guidelines on a wall. Appendix C is a photograph of this initial setup. On the other hand, we created a Mural board for real-time collaboration between all participants, remote and physical. The initial setup of the Mural board can be found in Appendix D.

We started by clarifying the overall format of the focus group and its goals. Next, we explained the brainstorming rules and guidelines to get everybody on the same page. By exposing our objectives, we tried to keep ideas more focused on what we were looking for with the activity. As part of these initial steps, we asked the team members if they agreed with the project's monitoring challenges or not, providing a way of evaluating our interpretation of the project characterisation phase (see Section 6.3).

To kick off the brainstorming part of the focus group, we ran a small example with the participants. We presented them with an A4 paper sheet with the following challenge: "Mushrooms in a pizza do not get fully cooked when the dough is just right". Then, we asked the participants all the questions from the guidelines to get insight into each section of a possible pattern to tackle that challenge. This exercise got us an almost full sheet in less than 10 minutes, and we believe it helped the participants get into a creative mindset.

With this warm-up completed, we moved on to the first monitoring challenge and repeated the same strategy. Instead of getting a single answer for each of the questions, we allocated around 15 minutes for each challenge and ensured there was at least one comment or idea to answer all the guideline questions. Every idea or comment that came up was discussed between the group and then written down on the paper sheets. Moreover, one of the participants volunteered to write the ideas on the Mural board at the same time we did on the paper sheets. During this first round of brainstorming, we took the role of facilitators and asked questions to keep the conversation going. We did so as impartially as we could to avoid inducing our ideas on the participants.

Afterwards, we took down the sheets from the whiteboard and passed them around the group. Every person had the chance to add written notes and ideas to contribute by herself, helping more reserved people to express their ideas. The online participants did the same task on the Mural board by adding notes to the challenge they were targeting. This exercise took around 15 minutes and provided additional feedback for each monitoring challenge.

Finally, we got to the second part of the focus group and clarified some of the notes and ideas of the brainstorming part. We only needed to clarify two things from everything we collected. During the last 20 minutes of the activity, we presented the patterns adopted during the implementation phase to the participants. We showed them A4 paper sheets with the main points of each pattern and briefly explained the problem they solved and the proposed solution. The sheets can be found in Appendix E. We ended the activity with the group providing feedback on the pattern adequacy for the project's challenges and comparing the patterns' main aspects to the ideas and comments gathered during the brainstorming part.

6.5.2 Analysis of results

The transcripts of the notes written for each challenge are available in Appendices F, G and H. We analysed these notes to extrapolate actionable insights. We started by reading all the notes and then correlating them to one another to include additional notes that convey more generalised insights that we can include in the patterns. The additional notes are portrayed in Figures 6.9, 6.10 and 6.11 as orange post-its, but are not included in the appendices.

We asked the participants for their level of agreement with the challenges we identified during the project characterisation phase. The whole group agreed that all the challenges existed in the project. In addition, there was a comment about the first challenge description. A participant mentioned that it was addressing two different problems — one regarding log readability and another regarding stack trace identification. Nonetheless, the team agreed it was a real challenge in the project's context.

In the following sections, we explain our analysis of the brainstorming session's notes and our main findings during this exercise. Note that this analysis relied solely on our interpretation of the written notes and was not validated by the focus group participants.

6.5.2.1 MC1. Logs are hard to read and it is difficult to find stack traces in the log files

In Figure 6.9 we present the annotated results for the project's first monitoring challenge. During the analysis of the initial notes, we made the following conclusions:

- A new pattern for managing the number of logs generated in the system could be helpful — the team pointed out that too many logs make for less readable log files. Thus, it can be interesting to develop a pattern that deals with this issue and provides a solution, or some guidelines, to help practitioners balance their system's log generation.

- STANDARD LOGGING (see Section 5.5) could be a solution for this challenge — the group mentioned that formatting logs would help see the errors more clearly. This resembles the STANDARD LOGGING pattern, which, in Section 6.4.1, we concluded is already implemented in the project. Therefore, the pattern on its own is not enough to tackle this challenge, but there seems to be a relation with the QUERY ENGINE pattern we chose to solve the current problem. In other words, QUERY ENGINE complements STANDARD LOGGING, which confirms our decision to add the former pattern in the latter's Related Patterns section.

- Having an adjustable logging level is a possible force to control log verbosity in the AUDIT LOGGING (see Section 5.2) and STANDARD LOGGING (see Section 5.5) patterns — the participants mentioned that the number of logs could be controlled through the log level configuration. This property seems a factor to consider as an addition in those logging patterns to give the practitioners more control over their logs. However, this force does not seem suitable to LOG SAMPLING because this pattern focus is to control log verbosity through many criteria.

- The LOG SAMPLING (see Section 5.6) pattern could be a possible solution to the current challenge — sampling and prioritising logs can help reduce the number of logs in the system and, thus, make it easier to go through the log files and find the stack traces. However, the group concluded that processing logs introduce a performance overhead in the system. This issue could be tackled by offloading the processing to other services or delaying the log processing because they mentioned that logs are only needed for debugging.

- Separating logs into different files by log level is not enough to solve the challenge — the team mentioned that it could help reduce the number of logs in each file. However, correlation and timeline are lost, and there is already a separation between *Warning* and *Error* log levels in the project. Hence, we believe the proposed solution is insufficient to tackle the current challenge and become a pattern.

- The EXCEPTION TRACKING (see Section 5.9) pattern can be a possible solution to the challenge — the participants mentioned that handling exceptions separate from logs could help make the stack traces easier to find. Although this is true, most exceptions are already logged to a separate log file, which still does not solve the issue. A dedicated solution to handle exceptions like the one described in EXCEPTION TRACKING could be more valuable due to its advanced capabilities.

- A possible solution could involve adopting EVENT SOURCING [59, 71] — the event-driven logging note and its associated ideas resemble part of the EVENT SOURCING pattern. It could be a suitable solution, but we think that the group did not list any of its disadvantages that could discourage the pattern's adoption in the current context. We argue that the complexity it adds to the system is not worth the value it would bring to the project.

6.5.2.2 MC2. The lack of dashboards makes it difficult to understand the metrics collected from the monolithic service

In Figure 6.10 we present the annotated results for the project's second monitoring challenge. During the analysis of the initial notes, we made the following conclusions:

- The LOG AGGREGATION [83] and QUERY ENGINE could be possible patterns to solve the issue — the participants mentioned a combined solution to aggregate, store, process and query the logs/data. These actions resemble the solutions of the mentioned patterns, but they seem to miss the point of the challenge. We believe that LOG AGGREGATION would not apply in this case because we are dealing with metrics, and QUERY ENGINE could help, but it is more suitable for logs or complex data than metrics. Thus, we deem these solutions slightly inadequate for the current challenge.

- The learning effort required for the monitoring tools is a relevant aspect to include in most design patterns — the group mentioned the big learning curve as a consequence of the

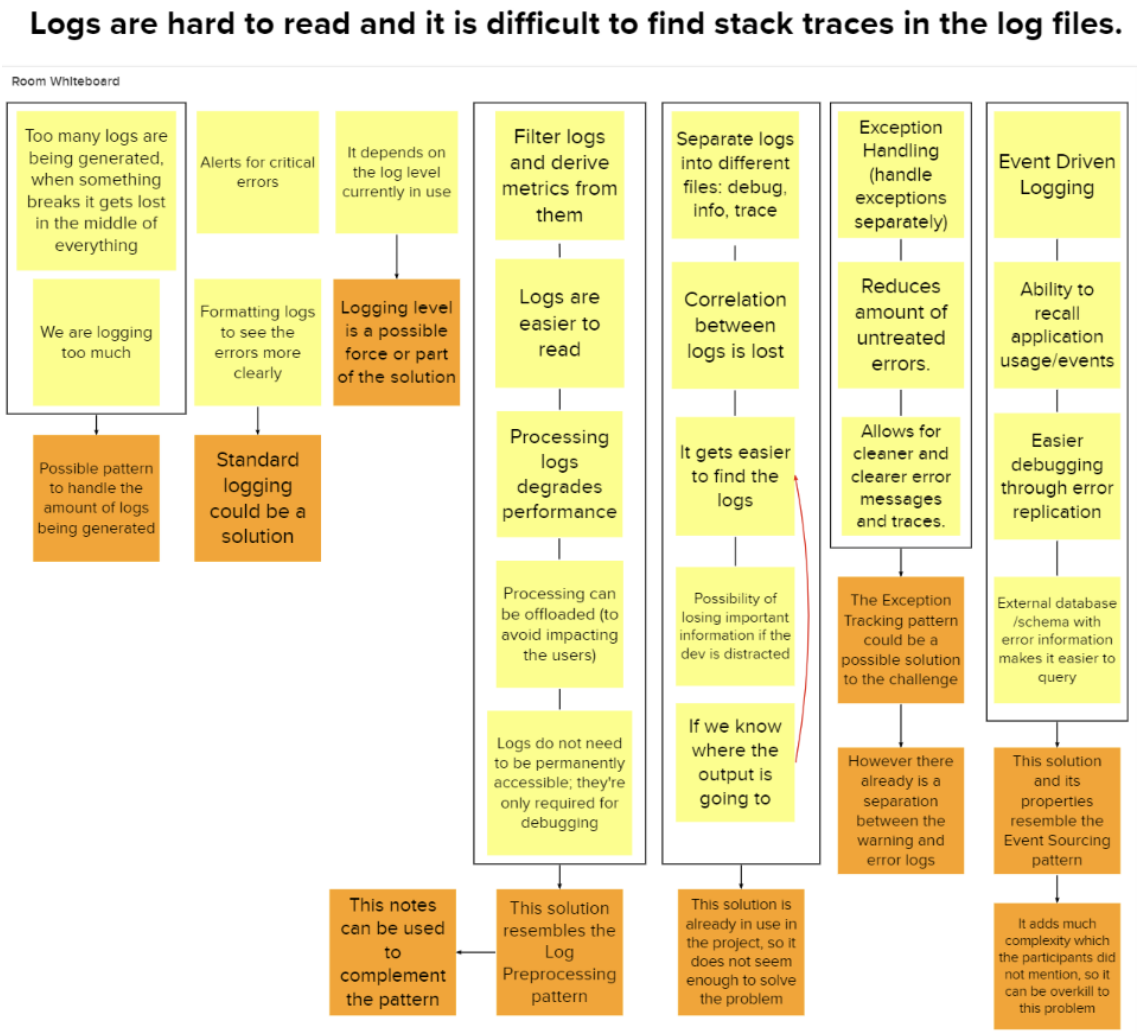


Figure 6.9: Annotated board with the focus group results for the monitoring challenge 1. The yellow notes were written during the brainstorming session and the orange notes were written as part of the analysis process.

proposed solutions. We agree with them and believe that the learning factor is important for democratising the use of monitoring solutions within companies. Therefore, that could fit as a force for the more complex design patterns.

- The participants notice the monitoring challenge when the app is in production — we think this sort of context is a bit too generic to help in the description of the patterns.
- Cost is a possible obstacle to the adoption of some solutions — perhaps due to the consulting background of the team, they mentioned that the customer could not be willing to support the cost of adopting a dashboard. Therefore, we conclude that costs are not only a trade-off but also something that can ultimately hinder the pattern adoption.

- Having a dashboard implies lower maintenance costs for the service but higher development costs for the monitoring infrastructure — the group pointed out these two consequences of adopting a dashboard. We think these consequences lack some nuance that we believe is important to consider. For example, with the adoption of a dashboard, the maintenance cost of the infrastructure to support the logs should increase, while the cost of maintaining a quality service should decrease. Therefore, these insights are too superficial to become part of the design patterns.
- Using AI to process the logs can become a new design pattern altogether — the participants mentioned this possibility to make log processing more effective and cheaper. We believe this resembles AIOps monitoring practices we have found in some literature [8, 20, 63], so a pattern could emerge from these practices.

6.5.2.3 MC3. There are no metrics on the background tasks service to keep track of errors and processing time

In Figure 6.11 we present the annotated results for the project's third monitoring challenge. During the analysis of the initial notes, we made the following conclusions:

- The challenge occurs more often when a job is triggered, or the database reset — the team provided further context to the problem by talking about these two conditions. However, these inputs are particular to the project's context and, thus, hard to generalise for the APPLICATION METRICS pattern.
- Access conditions to the different production environments impact the usability of metrics — the participants mentioned that it could be hard to collect and use metrics because they do not have access to some deployment environments. We think we overlooked this perspective on the design patterns' descriptions, which is crucial in consulting scenarios. Nonetheless, it seems to be a reasonable consequence to add to both the APPLICATION METRICS and INFRASTRUCTURE METRICS patterns.
- Without metrics, non-evident patterns and problems may go unnoticed — the group talked about this statement which we believe holds an underlying consequence of the metrics-related design patterns. Both of them help uncover things that were not seen before their adoption.
- The LIVENESS HEALTH ENDPOINT pattern could be part of a solution to this challenge — during the session, someone mentioned that native health monitoring could help tackle the issue. With further investigation, we concluded that it resembles the LIVENESS HEALTH ENDPOINT pattern. Even though this design pattern could help ensure that the service is not crashed, we do not think it would provide the visibility the team is lacking over the state of the background jobs. Therefore, we do not consider the mentioned solution to answer the project's challenge directly.

Figure 6.10: Annotated board with the focus group results for the monitoring challenge 2. The yellow notes were written during the brainstorming session and the orange notes were written as part of the analysis process.

- The participants mentioned that using a dashboard to display the state of each job and having a load-balancing mechanism to optimise the calculations were two possible solutions — We do not think that any of the two fully tackles the problem. The former still lacks the generation of metrics as information to be displayed, and the latter tackles the underlying performance issue instead of the lack of visibility the team has over it. Thus, we do not think they have any relevant conclusions for this dissertation's design patterns.

- Solutions to the challenge are unlikely to create value for the customer/user, so their adoption can be hindered by the implementation and maintenance costs — the team explained that the clients would not be willing to pay for the time to set up the monitoring tools. Even though we were considering costs as part of the forces of some patterns, we saw them more as a trade-off than an obstacle. The way the team exposed the problem helped us realise the importance of solution costs

- Reusability is a significant aspect to consider for solutions — the participants commented that reusing one of the solutions would be easier than reusing the other one as part of the discussion. These comments make us think that reusability is a relevant force for most of this dissertation's design patterns, one that we had not considered so far. Furthermore, reusability was mentioned as a way to divide the costs of maintaining a monitoring infrastructure between many projects inside the company, which is yet another advantage of reusable solutions.

6.5.3 Discussion

The focus group phase closed this dissertation's empirical study and provided many actionable results. The focus group made us conclude that the team members shared our interpretation of the project's monitoring challenges, reinforcing what we have learned during the characterization of the project that we have done earlier. In addition, from the confirmatory section of the activity, we conclude that the team found the selected design patterns adequate for the challenges.

Nonetheless, other solutions mentioned during the brainstorming section resembled design patterns. For **MC1**, we conclude that the LOG SAMPLING (cf. Section 5.6), EXCEPTION TRACKING (cf. Section 5.9) and EVENT SOURCING [59, 71] patterns could also address the challenge. The first and second aim to reduce the number of things that go into the log files to make them easier to read and information easier to find. The third aims to make error replication easier and logs more event-focused. However, this one involves many changes to the business logic and storage and requires a lot of effort from the team, so we think it is not appropriate for the solution. From the three design patterns mentioned for **MC1**, it seems that the participants were trying to solve a different underlying problem than us. While they were targeting the reduction of logs stored, we were looking for parsing and indexation of information to make it queriable. This could mean that our interpretation of the challenge is different from theirs.

The STANDARD LOGGING (cf. Section 5.5) pattern was also mentioned as a possible solution to **MC1**. However, in Section 6.4.1, we concluded that this pattern is already adopted in the project and, as such, is not enough to solve the current challenge. From the notes in Figure 6.9, we think that log formatting was associated with better readability, a consequence that we already included in the pattern's description. In addition, we confirmed that there may be a complementary relationship between the STANDARD LOGGING and QUERY ENGINE [14] patterns. In other words, we think a standard logging format is necessary for log indexation in the query engine system. This

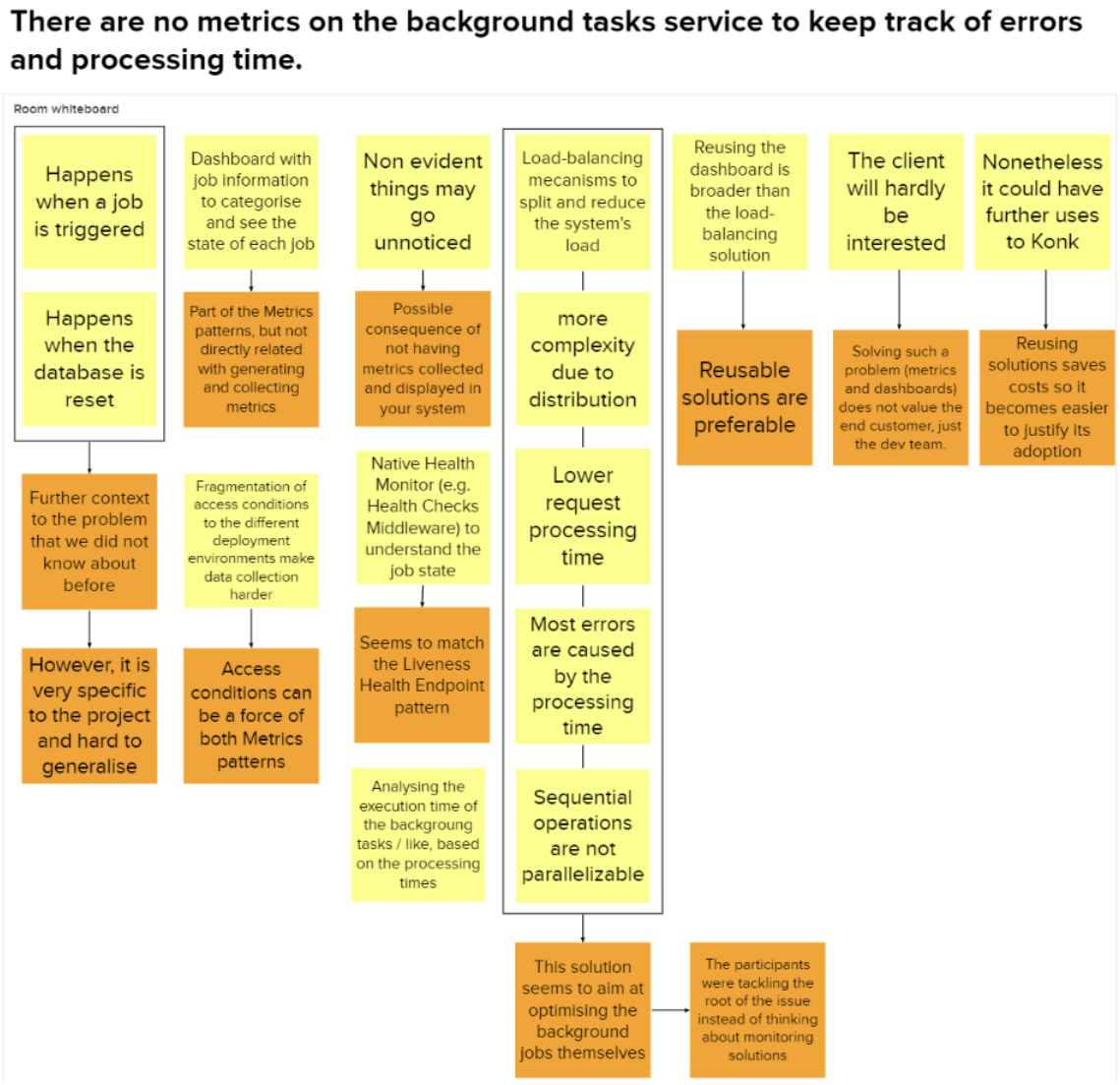


Figure 6.11: Annotated board with the focus group results for the monitoring challenge 3. The yellow notes were written during the brainstorming session and the orange notes were written as part of the analysis process.

relation was already part of the STANDARD LOGGING's Related Patterns section (cf. Section 5.5 and it got more corroborated from this conclusion.

Moving on to **MC2**, we found that LOG AGGREGATION [83] and QUERY ENGINE could be other pattern choices. These were mentioned together as two parts of the same solution. This occurrence corroborates the usage of these two patterns in combination, as mentioned in [14]. However, in the current challenge's context, these still seem insufficient to improve the visibility of the system's metrics. The participants commented that these two would provide a dashboard with an app overview, but we tend to disagree. None of the patterns mentions a dashboard as part of their descriptions. We think they focus on the storage of aggregated data and are not concerned with its visualisation. Therefore, we conclude that, for this problem, the APPLICATION

METRICS(cf. Section 5.13) and INFRASTRUCTURE METRICS (cf. Section 5.14) patterns are still the most suitable choices.

Regarding the **MC3**, the LIVENESS HEALTH ENDPOINT (cf. Section 5.10) pattern was mentioned as a way to address the issue. However, we do not think the lack of visibility over the service's internal state is solved with periodic liveness health checks. These should test the ability of the service to respond, so they will not, for example, inform the team about failed background jobs. The only thing we can conclude from this contribution is that the team did not provide any better alternative than the APPLICATION METRICS pattern to tackle **MC3**.

Two of the proposed solutions had the potential to become new design patterns. The participants mentioned using AI to process the logs and extract relevant insights. We found some literature regarding the topic [8, 20, 63] that proposes AIOps monitoring practices. As far as we know, this is still an area in development, but a design pattern could emerge from these AI-driven monitoring methods.

Moreover, the team pointed out that too many logs lead to less readable log files. One proposed solution resembled the LOG SAMPLING pattern. Although this does reduce the number of logs stored, it also causes the loss of information. A new pattern could be created to help practitioners log more efficiently without the need to filter the logs after they are generated. This improvement could be achieved through, for example, collecting guidelines for higher quality logging.

One of the goals of the empirical study was to improve the quality of this dissertation's patterns. Some of the information we got from the focus group activity helped us achieve that goal.

One of the first things we noticed was quite important to the participants was the solution's cost, in all its aspects (e.g. development, maintenance, infrastructure). Due to their IT consulting background, they pointed out that when something does not bring direct value to the customer or end-user, they are much less likely to pay for it. The cost either hinders monitoring efforts or makes them an afterthought. Therefore, we conclude that cost is a crucial aspect of everything monitoring-related because, most of the time, its practices do not bring direct value to the system's users. As such, we revisited the design patterns to ensure that cost was explicitly treated as a negative consequence. We conclude that most patterns already mentioned some sort of cost or effort as part of their drawbacks, but we still tweaked most of their descriptions to clarify that costs would increase. The only patterns that remain unchanged were LIVENESS HEALTH ENDPOINT, READINESS HEALTH ENDPOINT and STANDARD LOGGING because we think these do not involve as much costs as the other ones.

During the activity, the participants also mentioned the importance of a tool's learning curve. They explained that the easier it is to work with a solution, the better, enabling them to move freely from project to project without additional overhead. This still ties with the cost factor from the previous paragraph, i.e., the learning curve is the learning cost of the solution. We believe that this concern can be addressed mainly in two ways. On the one hand, a company can decide to use the same monitoring solutions across all projects, eliminating the learning factor from exchanging projects. On the other hand, the team can adopt easy-to-learn monitoring solutions, reducing the learning overhead. Nonetheless, we agree with the participants' perspective and incorporated the

learning curve as a drawback of the DISTRIBUTED TRACING and AUDIT LOGGING patterns, which we believe can have pretty complex implementations.

We believe some of the group's comments are relevant forces for this dissertation's patterns. First, reusability was mentioned as an efficient way to deal with cost concerns. The higher a solution's reusability, the more projects it can fit in. This leads to costs being divided by all the projects and, consequently, makes it easier for the pattern to be adopted. We believe this is a vital aspect to consider, although there is always a limit on how reusable a solution can be. We mean that, since monitoring practices should address the system's needs, they most of the time need to be customised. Nevertheless, we revised the pattern catalogue to include references to reusability as a force of the design patterns that forcefully used a *Metrics Service* because this component appears in many patterns, even if optionally. Thus, we added one additional force regarding reusability to the APPLICATION METRICS, INFRASTRUCTURE METRICS and DEPLOYMENT TRACKING patterns (cf. Sections 5.13, 5.14 and 5.8).

Secondly, we concluded that having an adjustable logging level is a relevant force for the AUDIT LOGGING and STANDARD LOGGING patterns because the team needs control over log verbosity (cf. Sections 5.4 and 5.5). However, this force does not apply so well to LOG SAMPLING because this pattern is already concerned with controlling log verbosity through many criteria, including the logging level.

Moving on to the consequences, the participants mentioned that access conditions to the different deployment environments impact what they can do with the collected metrics. Because some metrics depend directly on the host machine and end-user usage data, both available exclusively in production, this consequence applies to the APPLICATION METRICS and INFRASTRUCTURE METRICS patterns. We revised these design patterns to include access restrictions in their Consequences sections (cf. Sections 5.13 and 5.14).

Moreover, we conclude that metrics enable the team to notice non-evident patterns and problems that would most likely go unnoticed. This consequence applies to the same design patterns. These patterns' Consequences section already mentions the impact of metrics on strategic decisions, but we think this can be complemented with the input from the focus group. Therefore, we slightly adjusted each pattern's Consequences to include this idea (cf. Sections 5.13 and 5.14).

Finally, we also concluded that the QUERY ENGINE complements the STANDARD LOGGING pattern. The latter works as a prerequisite for the former. The query engine system could even be in charge of converting all logs to the standard format. This close relationship between the patterns is already reflected in their Related Patterns sections, so it corroborates our initial description for these patterns.

In conclusion, we believe that the focus group and the results it produced helped us improve our design patterns and evaluate the work done in the previous phases of the empirical study. The feedback we gathered was used for a second iteration of the design patterns to make them better theories [49]. We only sought corroboration for the patterns selected in Section 6.4.1, so there is still a lack of corroboration for all other patterns.

6.6 Threats to validity

Even though we deem the overall empirical study a vital part of this dissertation, it is not without shortcomings. In this section, we evaluate the validity of the study. In other words, we analyse "to what extent the results are true and not biased by the researchers' subjective point of view " [74]. In the following paragraphs, we analyse the main threats to this empirical study's validity and how they were handled throughout it. Following the guidelines proposed in [74], we focus on four validity aspects: external validity, internal validity, construct validity and reliability.

Regarding construct validity, the lack of access to the production environment stopped us from getting a direct view of the production system. Given that all monitoring practices and design patterns aim at that environment, characterising the project's monitoring capabilities and challenges without looking into production partially misses the point of our research. To mitigate this issue, we used methodological triangulation and looked for the same conclusions in the team leader interview, the team member survey and the source code and documentation analysis.

Regarding internal validity, the description of the focus group challenges and their presentation to the participants might have induced a bias towards the solutions we used. Since we had an idea of what the solution could be when we wrote the monitoring challenges, that may have caused the challenges to point towards that solution. To mitigate this issue, we encouraged the participants to look beyond the questions and asked further questions on their comments to go beyond the written challenges. Moreover, the team members' experience may have impacted their answers to the survey conducted for the project characterisation phase. Although knowing the surveyees' experience was not our ultimate goal, we collected it to be able to analyse the data based on that factor and, thus, reduce its impact on our conclusions.

The external validity issues we faced mainly regarded the project itself. Firstly, there were very few participants in the study because we were limited to the project's team. In addition, the focus group participants were all aware of the project's context and that we were working within its context. During the brainstorming session, we clearly stated that we were looking for ideas for the challenge, not the project. In other words, we told them to imagine the problem in other contexts and provide insights that could also happen in those. However, we believe the brainstorming results are still very tied to the project's context.

These external validity issues make our conclusions harder to generalise to other scenarios. Given that we followed an action research methodology, we expected the narrow context of a single case and the difficulty in generalising the results. However, we believe that our results can still apply to another project that matches the same context and faces similar monitoring challenges. Thus, we think our results are generalisable within a similar context to the one we studied, which is what we intended to achieve.

Finally, regarding reliability, our involvement in the study and experience with the design patterns may have biased our conclusions. Furthermore, most of this research's findings were based on our interpretation of the results from the three phases of the study. To improve reliability, we tried to provide a clear chain of evidence from observations to conclusions and thoroughly explain

our rationale. For instance, during the analysis of the focus group results we present the activity notes and our own interpretation notes separately so the original ideas remain transparent. Moreover, in Section 6.5.2, our analysis starts with a description of what was said or written and only then we try to establish a comparison with other parts of our work. Lastly, the methodological triangulation we did throughout the many parts of the study and the team's feedback on our findings helped us reduce some of the interpretation bias.

6.7 Conclusions

As the empirical study comes to an end, we are now able to provide an answer to this dissertation's second research question:

RQ2. How accurate are the proposed patterns' contexts, forces, problems, solutions and consequences?

By applying the patterns in a real-world project with real monitoring needs and challenges, we were able to test how the patterns worked in practice. According to the project's context and challenges, we only selected the QUERY ENGINE, APPLICATION METRICS, and INFRASTRUCTURE METRICS patterns. Thus, these were the only patterns in which we could test the Context, Problem and Solution sections.

Overall, we conclude that these patterns' contexts, problems and solutions are adequate because (1) the problems were easy to match with the project's challenges, so we believe their descriptions are intuitive; (2) their contexts matched with the project's context, so their problems are an issue in the context they describe; (3) their solutions made sense to tackle the identified problem, so we think their solutions are well described and adequately address the problems the patterns describe; and (4) their solutions were applicable in the project's context, so their contexts adequately encompass the solutions described by each pattern.

Nonetheless, we could extract relevant information regarding the patterns because the focus group discussion was broader than the project's context. We conclude that the patterns' descriptions are overall well done and encompass a big part of the things mentioned in the activity. Nonetheless, we implemented some changes in the patterns to improve their initial descriptions. These changes are summarised in Table 6.1.

Although these changes reveal that the patterns' Consequences and Forces lacked some details, the result is a reiterated pattern catalogue that incorporates the results of this empirical research. Therefore, we believe the revised catalogue is better prepared for real-world usage.

To conclude this chapter, we want to reinforce that a more thorough empirical study could have been conducted. However, due to time constraints, we had to reduce the scope of the study to focus on developing the design patterns. With more time, we would have liked to implement a more robust monitoring stack in the project and let the team work with it as part of their day-to-day activities. We would add more metrics to the system and provide training to the team.

Table 6.1: Summary of changes to this dissertation's design patterns after the empirical study. The actual change appear highlighted in *italic*.

Pattern name	Section	Action	Changelog
INFRASTRUCTURE METRICS	Consequences	Add	*Collecting infrastructure data in the cloud depends heavily on what the team has access to, so this pattern gets harder to adopt the fewer access rights the team has to the infrastructure.*
			Metrics enable the team to notice non-evident patterns and problems that would most likely go unnoticed otherwise.
			Aggregating and storing a high number of metrics may require considerable infrastructure, increasing its costs.
	Forces	Add	*Reusable solutions are more desirable as they help mitigate development and infrastructure costs.*
APPLICATION METRICS	Consequences	Add	*Collecting application data in the cloud depends heavily on what the team has access to, so this pattern gets harder to adopt the fewer access rights the team has to the deployment environments.*
			Metrics enable the team to notice non-evident patterns and problems that would most likely go unnoticed otherwise.
		Change	Aggregating and storing a high number of metrics may require considerable infrastructure*, increasing its costs*.
	Forces	Add	*Reusable solutions are more desirable as they help mitigate development and infrastructure costs.*
AUDIT LOGGING	Consequences	Add	*Since the solution is a bit complex, its learning curve may create some initial friction with the team.*
		Change	Audit logs are difficult to process, and tools to do so may require considerable maintenance or infrastructure *costs*.
	Forces	Add	*Log verbosity must be easily adjustable so the team can somewhat control the number of generated logs.*
STANDARD LOGGING	Forces	Add	*Log verbosity must be easily adjustable so the team can somewhat control the number of generated logs.*
DISTRIBUTED TRACING	Consequences	Add	*Since the solution is a bit complex, its learning curve may create some initial friction with the team.*
DEPLOYMENT TRACKING	Forces	Add	*Reusable solutions are more desirable as they help mitigate development and infrastructure costs.*
SYNTHETIC TESTING	Consequences	Add	*Having an external testing platform requires additional infrastructure and maintenance, increasing the costs of the system.*
LOG SAMPLING	Consequences	Change	Implementing adaptive sampling and choosing the appropriate log criteria for sampling may require considerable *development* effort *and costs*.

Afterwards, we would observe how the team leveraged the new monitoring capabilities of the project for a longer period (somewhere from 3 to 9 months) so we could report on the evolution of the monitoring processes. This would take considerably more time, so it was unfeasible for our empirical study. Nonetheless, the study as it was performed provided some interesting results for Konkconsulting's project and this dissertation, so we believe it was still successful.

Chapter 7

Conclusions and future work

As this dissertation ends, we reflect on its main contributions and possible research directions it opens. In Section 7.1, we list the main contributions of this dissertation. In Section 7.2, we explore future research directions that can improve this work. Finally, in Section 7.3, we summarise its main conclusions and end our work.

7.1 Contributions

This work has three major contributions to the field of monitoring and software engineering in general. The first one are the results the literature review conducted in Chapter 3. Even though this was not of this dissertation's goals, we had to perform a thorough literature review to identify and understand existing monitoring practices and design patterns. Therefore, we believe the resulting list of unique monitoring practices and patterns that were identified is, by itself, a relevant contribution for future research that intends to work on this topic.

However, the most important contribution is the catalogue of design patterns to monitor cloud applications (see Chapter 5). It includes eleven new design patterns: AUDIT LOGGING, STANDARD LOGGING, LOG SAMPLING, DISTRIBUTED TRACING, DEPLOYMENT TRACKING, EXCEPTION TRACKING, LIVENESS HEALTH ENDPOINT, READINESS HEALTH ENDPOINT, SYNTHETIC TESTING, APPLICATION METRICS, and INFRASTRUCTURE METRICS. We think the patterns have a good level of detail that makes them easy to pick up by beginners and professionals in the field. We tried to explore the nuances and expose the trade-offs of each solution as thoroughly as we could without locking the solution to a specific implementation. Moreover, we were able to provide three real-world known uses for each of the patterns based on grey literature we found online. This kind of information is commonly found in blog post, so we were able to just a few scientific studies that corroborated the pattern adoption by the industry. Lastly, due to this dissertation's empirical study (see Chapter 6), we managed to improve most patterns even further.

This results in a revised pattern catalogue that we strongly believe is of use to the industry and practitioners.

Finally, a slightly minor contribution was the initial empirical corroboration of this dissertation's design patterns. As theories, the design patterns could have applied to the empirical study's project but still fail to solve the problems it had, in which case they would be falsified [49]. Even though we did not manage to falsify any pattern concretely, the empirical exercise of applying it to the project revealed some flaws in the patterns. We conclude the patterns, especially the ones that were selected for the project (i.e. QUERY ENGINE, APPLICATION METRICS and INFRASTRUCTURE METRICS), successfully passed their first test, even though it was a small one.

7.2 Future work

All three contributions of this dissertation are open for expansion. Future research can pick on the literature review we provide and further deepen it by analysing more literature. This may reveal new monitoring practices and patterns we could not find during this work.

During the literature review, we argue that some practices are not good pattern candidates and that some patterns are already sufficiently described and, thus, not included in this dissertation's pattern catalogue. Future work may disagree with our view and pick on these ideas to develop or improve more monitoring design patterns. This is especially the case for alerting practices. These are very relevant for practitioners, but out of the scope of our work.

On a similar note, the design patterns proposed in this dissertation are not set in stone. Future research may improve them by adding new information or changing their structure, for example. Even if they are not changed, the patterns' relevance grows when other patterns reference them. Thus, other works that propose design patterns can, and should, point towards this work's patterns to help the reader find more information about monitoring cloud applications.

Lastly, our empirical study was only an initial test for this dissertation's patterns. Future research can continue to test the adoption of these patterns to corroborate or, contrarily, falsify them further. Either way, conducting more empirical tests and reporting their results is a relevant contribution to increasing the confidence that practitioners and the industry can put on design patterns, particularly the ones we proposed.

7.3 Conclusions

This dissertation explored existing practices for monitoring cloud-hosted applications. Recent trends in cloud computing and microservices architecture forced monitoring processes to change. This work aimed at solving two issues with monitoring practices: (1) the lack of formality and standardization on these practices, and (2) the lack of empirical validation to evaluate them and prove their usefulness. The first makes practices hard to communicate and reuse between cases. The second does not distinguishing good practices from bad practices.

We conducted a literature review from which we identified seven monitoring practices and four design patterns that were suitable pattern candidates. We developed these eleven patterns to form a comprehensive pattern catalogue for monitoring cloud applications. Through a small empirical study, we corroborated these patterns descriptions and improved them to provide a revised pattern catalogue as this work's main contribution to the field of software engineering.

Overall, we believe this dissertation achieved its goals. With more time, alerting patterns would have made it into the catalogue. Alerting is ubiquitous among tools and grey literature, and there are even good alerting practices outlined in some literature. In addition, with more time we would conduct a more in-depth empirical study. The chosen project, although suitable for the patterns, was small and could not touch on most aspects of the design patterns. To conclude, three of this dissertation's design pattern — LIVENESS HEALTH ENDPOINT, READINESS HEALTH ENDPOINT, and SYNTHETIC TESTING — got accepted for a paper at the 25th European Conference on Pattern Languages of Programs (EuroPLop 2022) [5], which, at the time of writing, still does not have its proceedings available.

References

[1] Cloud design patterns - Azure Architecture Center. Available at https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns. (Accessed in Jun. 11, 2022).

[2] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[3] Pradeep Agarwal. Using distributed tracing to improve logging and debugging processes. Available at https://meesho.io//blog/using-distributed-tracing-to-improve-logging-and-debugging-processes. (Accessed in Jul. 03, 2022).

[4] Samrose Ahmed. What makes a good audit trail? Available at https://apptrail.com/blog/2022/02/05/what-makes-a-good-audit-trail. (Accessed in Jul. 03, 2022).

[5] Carlos Albuquerque, Kadu Relvas Barral, Filipe F Correia, and Kyle Brown. Proactive monitoring design patterns for cloud applications. In *Proceedings of the 25th European Conference on Pattern Languages of Programs*, page 21, Irsee, Germany, 2022. ACM.

[6] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Oxford university press, New York, 1977.

[7] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Correct Audit Logging: Theory and Practice. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 139–162, Berlin, Heidelberg, 2016. Springer.

[8] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated Analysis of Distributed Tracing: Challenges and Research Directions. *Journal of Grid Computing*, 19(1):9, February 2021.

[9] Kevin Borders. How to Manage Exceptions at Scale. Available at https://betterprogramming.pub/how-to-manage-exceptions-at-scale-c0ee5bef05e1. (Accessed in Jul. 05, 2022).

[10] Tobias Bradtke. Wait for it - Using Readiness Probes for Service Dependencies in Kubernetes. Available at https://www.giantswarm.io/blog/wait-for-it-using-readiness-probes-for-service-dependencies-in-kubernetes. (Accessed in May. 05, 2022).

[11] Colin Breck. Kubernetes Liveness and Readiness Probes: How to Avoid Shooting Yourself in the Foot. Available at https://blog.colinbreck.com/kubernetes-liveness-and-readiness-probes-how-to-avoid-shooting-yourself-in- (Accessed in Apr. 27, 2022).

[12] Mike Brittain. Tracking Every Release. Available at https://www.etsy.com/codeascraft/track-every-release?utm_source=OpenGraph&utm_medium=PageTools&utm_campaign=Share. (Accessed in Jul. 03, 2022).

[13] Kyle Brown and Bobby Woolf. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pages 1–35, 2016.

[14] Kyle Brown, Bobby Woolf, Joseph Yoder, Cees De Groot, Chris Hay, and Ian J. Mitchell. Patterns for Developers and Architects building for the cloud. Available at https://kgb1001001.github.io/cloudadoptionpatterns/. (Accessed in Feb. 24, 2022).

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture, volume 1: A system of patterns*, volume 1 of *Wiley Software Patterns Series*. Wiley New York, Great Britain, 1 edition, 1996.

[16] callstats.io. How We Use Prometheus for Simple and Powerful Monitoring. Available at https://medium.com/callstatsio/how-we-use-prometheus-for-simple-and-powerful-monitoring-68ee5240fc01. (Accessed in Jun. 30, 2022).

[17] Santiago Campuzano. Prometheus Monitoring at Scale: War Stories from the GumGum Trenches. Available at https://medium.com/gumgum-tech/prometheus-monitoring-at-scale-war-stories-from-the-gumgum-trenches-f66393c (Accessed in Jun. 30, 2022).

[18] Joel Carlbark. Synthetic Monitoring: A Case Study of the Meltwater API. Available at https://underthehood.meltwater.com/blog/2018/05/07/synthetic-monitoring-a-case-study-of-the-meltwater-api/. (Accessed in Apr. 29, 2022).

[19] Nick Chapsas. How to collect metrics and create dashboards using Grafana, Prometheus and AppMetrics in .NET Core. Available at https://www.youtube.com/watch?v=sM7D8biBf4k. (Accessed in Jun. 21, 2022).

[20] Hao Chen, Kegang Wei, An Li, Tao Wang, and Wenbo Zhang. Trace-based Intelligent Fault Diagnosis for Microservices with Deep Learning. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 884–893, Madrid, Spain, July 2021. IEEE. ISSN: 0730-3157.

[21] Jürgen Cito, Devan Gotowka, Philipp Leitner, Ryan Pelette, Dritan Suljoti, and Schahram Dustdar. Identifying web performance degradations through synthetic and real-user monitoring. *Journal of Web Engineering*, 14(5&6):414–442, 2015.

[22] Cloud Native Computing Foundation. CNCF Archives the OpenTracing Project. Available at https://www.cncf.io/blog/2022/01/31/cncf-archives-the-opentracing-project/. (Accessed in Jul. 04, 2022).

[23] Rodrigo da Rosa Righi, Matheus Lehmann, Marcio Miguel Gomes, Jeferson Campos Nobre, Cristiano André da Costa, Sandro José Rigo, Marcio Lena, Rodrigo Fraga Mohr, and Luiz Ricardo Bertoldi de Oliveira. A Survey on Global Management View: Toward Combining System Monitoring, Resource Management, and Load Prediction. *Journal of Grid Computing*, 17(3):473–502, September 2019.

[24] Datadog. Synthetic Testing: What It Is & How It Works. Available at https://www.datadoghq.com/knowledge-center/synthetic-testing/. (Accessed in Apr. 26, 2022).

[25] Chris Dermody. Best practices for audit logging in a SAAS business/application. Available at https://medium.com/@cderm/best-practices-for-audit-logging-in-a-saas-business-application-f1ace8ffd9b (Accessed in Jul. 02, 2022).

[26] Sandeep Dinesh. Readiness vs liveliness probes: How to set them up and when to use them in your Kubernetes cluster. Available at https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-setting-up-health-checks-with-readiness-and-liven (Accessed in Apr. 26, 2022).

[27] Trung Dinh. Using AWS X-Ray to Trace and Understand Your Application. Available at https://floqast.com/engineering-blog/post/using-aws-x-ray/. (Accessed in Jul. 03, 2022).

[28] Jürgen Dobaj, Markus Schuss, Michael Krisper, Carlo Alberto Boano, and Georg Macher. Dependable mesh networking patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–14, 07 2019.

[29] Elasticsearch. Elasticsearch: The Official Distributed Search & Analytics Engine. Available at https://www.elastic.co/elasticsearch. (Accessed in Jun. 21, 2022).

[30] Rob Ewaschuk. My Philosophy on Alerting. Available at https://docs.google.com/document/d/199PqyG3UsyXlwieHaqbGiWVa8eMWi8zzAn0YfcApr8Q/edit?usp=embed_facebook. (Accessed in Feb. 4, 2022).

[31] Rob Ewaschuk and Betsy Beyer. *Monitoring Distributed Systems*, page 550. O'Reilly Media, Inc., U.S.A., 1st edition, April 2016.

[32] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933, 2014.

[33] Fauna Inc. Serverless Change Capture for Ruby on Rails. Available at https://medium.com/fauna/serverless-change-capture-for-ruby-on-rails-255511164687. (Accessed in Jul. 02, 2022).

[34] Flexera. 2021 State of the Cloud Report. Technical report, Flexera, 2021.

[35] Martin Fowler. Audit Log. Available at https://martinfowler.com/eaaDev/AuditLog.html. (Accessed in Jul. 02, 2022).

[36] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 1 edition, 2002.

[37] Freyja. How to monitor deployments for performance problems with Raygun. Available at https://raygun.com/blog/monitor-deployments/. (Accessed in Jul. 03, 2022).

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, United States, 1994.

[39] GitLab. 2021 DevSecOps Survey Results. Technical report, GitLab, 2021.

[40] Google. Cloud Audit Logs overview | Cloud Logging. Available at https://cloud.google.com/logging/docs/audit. (Accessed in Jul. 02, 2022).

[41] Brendan Gregg. The USE Method. Available at https://www.brendangregg.com/usemethod.html. (Accessed in Apr. 11, 2022).

[42] Brendan Gregg. Thinking methodically about performance. *Communications of the ACM*, 56(2):45–51, February 2013.

[43] Nishant Gupta. 5 Design Patterns for Building Observable Services. Available at https://engineering.salesforce.com/5-design-patterns-for-building-observable-services-d56e7a330419. (Accessed in Feb. 20, 2022).

[44] Nishant Gupta. READS: Service Health Metrics. https://engineering.salesforce.com/reads-service-health-metrics-1bfa99033adc. (accessed Feb. 20, 2022).

[45] Joab Jackson. The RED Method: A New Approach to Monitoring Microservices. Available at https://thenewstack.io/monitoring-microservices-red-method/. (Accessed in Apr. 11, 2022).

[46] Gerhard Jacobs. Supporting Rapid Business Growth with Scalable Solutions. Available at https://blog.sentry.io/2022/04/05/scaling-a-high-growth-business-with-sentry. (Accessed in Jul. 05, 2022).

[47] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, March 1987.

[48] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 220–242, Berlin, Heidelberg, 1997. Springer.

[49] Christian Kohls and Stefanie Panke. Is that true...? Thoughts on the epistemology of patterns. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, pages 1–14, 2009.

[50] Konkconsulting. konkconsulting. Available at http://www.konkconsulting.com/EN. (Accessed in Feb. 18, 2022).

[51] Georgios Kornaros and Dionisios Pnevmatikatos. A survey and taxonomy of on-chip monitoring of multicore systems-on-chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2):1–38, 2013.

[52] Kubernetes. Configure Liveness, Readiness and Startup Probes. Available at https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. (Accessed in Apr. 27, 2022).

[53] Kirk Larkin, Juergen Gutsch, and Rick Anderson. Logging in .NET Core and ASP.NET Core. Available at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/. (Accessed in May. 13, 2022).

[54] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27(1):25, November 2021.

[55] Tiago Maia and Filipe Correia. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLoP '22, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Mary Lynn Manns and Linda Rising. *Fearless change: patterns for introducing new ideas*. Addison-Wesley, Boston, 2005.

[57] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems. *IEEE network*, 7(6):20–30, 1993.

[58] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[59] Microsoft. Event Sourcing pattern. Available at https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing. (Accessed in Jul. 02, 2022).

[60] Microsoft. Health Endpoint Monitoring pattern. Available at https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring. (Accessed in Apr. 11, 2022).

[61] Microsoft. Synthetic Monitoring Tests. Available at https://microsoft.github.io/code-with-engineering-playbook/automated-testing/synthetic-monitoring-tests/. (Accessed in Apr. 28, 2022).

[62] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Test activities in the continuous integration and delivery pipeline. *Journal of Software: Evolution and Process*, 31(4):22, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2153.

[63] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 241–250, Larnaca, Cyprus, May 2019. IEEE.

[64] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., Canada, 2nd edition, August 2021.

[65] OpenTelemetry. Logs Data Model. Available at https://opentelemetry.io/docs/reference/specification/logs/data-model/. (Accessed in Jul. 04, 2022).

[66] Jonathan Owens. Caring for Container-Based Services with Checks, Monitoring, and Alerts. Available at https://newrelic.com/blog/how-to-relic/container-service-checks. (Accessed in May. 02, 2022).

[67] Maulik Pandey. Building Netflix's Distributed Tracing Infrastructure. Available at https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304. (Accessed in Jul. 03, 2022).

[68] Jamie Penney. How we use Raygun to support Raygun. Available at https://raygun.com/blog/how-we-use-raygun-to-support-raygun/. (Accessed in Jul. 05, 2022).

[69] Aditya Praharaj. Structured Logging: The Best Friend You'll Want When Things Go Wrong. Available at https://medium.com/grab/structured-logging-the-best-friend-youll-want-when-things-go-wrong-1504222a
(Accessed in Jul. 04, 2022).

[70] Prometheus. Prometheus Overview. Available at https://prometheus.io/docs/introduction/overview/. (Accessed in Jun. 21, 2022).

[71] Chris Richardson. A pattern language for microservices. Available at http://microservices.io/patterns/. (Accessed in Feb. 25, 2022).

[72] Chris Richardson. *Microservices patterns: with examples in Java*. Manning Publications Co., Shelter Island, NY, October 2018.

[73] Armin Ronacher. Debug Issues Faster with Breadcrumbs. Available at https://blog.sentry.io/2016/05/04/breadcrumbs. (Accessed in Jul. 05, 2022).

[74] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, December 2008.

[75] Ben Schmaus. Making the Netflix API More Resilient. Available at https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d. (Accessed in Jul. 04, 2022).

[76] Rob Scott. Utilizing Kubernetes Liveness and Readiness Probes to Automatically Recover From Failure. Available at https://medium.com/spire-labs/utilizing-kubernetes-liveness-and-readiness-probes-to-automatically-recover-
(Accessed in May. 02, 2022).

[77] SentinelOne. Log Formatting: 7 Best Practices for Readable Log Files. Available at https://www.sentinelone.com/blog/log-formatting-best-practices-readable/. (Accessed in Jul. 04, 2022).

[78] Sentry. Sentry for iOS. Available at https://docs.sentry.io/platforms/apple/guides/ios/. (Accessed in Jul. 05, 2022).

[79] Sentry. Sentry vs Logging. Available at https://sentry.io/vs/logging/. (Accessed in Jul. 05, 2022).

[80] Will Sewell. How we deploy to production over 100 times a day. Available at https://monzo.com/blog/2022/05/16/how-we-deploy-to-production-over-100-times-a-day. (Accessed in Jul. 03, 2022).

[81] Yuri Shkuro. Evolving Distributed Tracing at Uber Engineering. Available at https://eng.uber.com/distributed-tracing/. (Accessed in Jul. 04, 2022).

[82] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, Google, 2010.

[83] Tiago Boldt Sousa. *Engineering Software for the Cloud: A Pattern Language*. PhD thesis, University of Porto, Porto, May 2020.

[84] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, PLoP '15, USA, 2015. The Hillside Group.

[85] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a pattern language for engineering software for the cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs*, PLoP '18, USA, 2018. The Hillside Group.

[86] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 48(6):2128–2140, 2022.

[87] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[88] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[90] Splunk. Blue Apron Decreases Load Time by 30% With Splunk Synthetic Monitoring. Available at https://www.splunk.com/en_us/customers/success-stories/blue-apron.html. (Accessed in Apr. 30, 2022).

[91] Cindy Sridharan. *Distributed systems observability: a guide to building robust systems*. O'Reilly Media, U.S.A., 2018.

[92] Eugene Stepnov. 10+ Best Error Monitoring and Error Tracking Tools. Available at https://flatlogic.com/blog/10-best-error-monitoring-and-error-tracking-tools/. (Accessed in Jul. 04, 2022).

[93] Domenico Stragliotto. How we implemented RED and USE metrics for monitoring. Available at https://medium.com/thron-tech/how-we-implemented-red-and-use-metrics-for-monitoring-9a7db29382af. (Accessed in Jun. 21, 2022).

[94] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 69–79, 2022.

[95] Muhammad Waseem, Peng Liang, Aakash Ahmad, Mojtaba Shahin, Arif Ali Khan, and Gastón Márquez. Decision Models for Selecting Patterns and Strategies in Microservices Systems and their Evaluation by Practitioners. *arXiv:2201.05825 [cs]*, page 10, January 2022. arXiv: 2201.05825.

[96] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182, December 2021.

[97] David Yanacek. Implementing health checks. Available at https://aws.amazon.com/builders-library/implementing-health-checks/. (Accessed in May. 03, 2022).

[98] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163–188, January 2005.

[99] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, May 1998. Conference Name: Computer.

Appendix A

Team leader semi-structured interview skeleton

1. Initial project characterisation
   (a) Can you please explain the project and its context?
   (b) How is the team constituted?
   (c) What is the state of the project? Is it in active development, maintenance or other?
2. Technical characterisation
   (a) Which programming languages do you use? What is the project's technological stack?
   (b) What is the project's high-level architecture?
   (c) How many daily users does the system have, approximately?
   (d) What is the project size? You can use, for example, lines of code, number of services...
3. Development practices
   (a) Where is the repository hosted?
   (b) How do you deploy the application into production?
   (c) Is the application containerised?
   (d) How do you detect errors/failures in production?
   (e) How do you debug/troubleshoot a problem in production?
   (f) Do you perform issue/bug tracking for failures?
   (g) Do you reevaluate internal processes after a system outages? In other words, do you perform *postmortems*?
4. Major challenges
   (a) What are the major development challenges you face with this project?

(b) What are the major monitoring challenges you face with this project?

(c) Are there other major challenges?

5. Monitoring practices

   (a) Logging

      i. Does your application produce logs?

      ii. Do the logs follow a structured format?

      iii. Where are the logs stored?

      iv. Are the logs aggregated in a single place?

      v. How are the logs used and analysed by the team?

   (b) Tracing

      i. Do you use any tracing tool?

      ii. Where are the traces stored?

      iii. Are the traces aggregated in a single place?

      iv. How are the traces used and analysed by the team?

   (c) Remediation

      i. When there is a production outage, for example, when the application or service crashes, what is the usual remediation process?

      ii. Do you have alerts configured to set off when something goes wrong?

      iii. Which channels do you use for alerts? Some example include Microsoft Teams, email, text message and pages.

   (d) Metrics

      i. Do you collect and store metrics about the deployment infrastructure, like disk and CPU usage of the host machine? If so, which?

      ii. Do you collect and store metrics about the deployed application, like user requests per day or user orders? If so, which?

      iii. Where are the metrics stored?

      iv. Are the metrics aggregated in a single place?

      v. How are the metrics used and analysed by the team?

   (e) Health monitor

      i. How do you become aware that the deployed application is *down*?

      ii. On the contrary, consider that the deployed application is up and running but requests are always returning with an error. Can you notice such a scenario?

      iii. What if, instead of returning with an error, users are waiting too long to get a reply. Can you notice such a scenario?

iv. Do you monitor these scenarios from within the production environment or from an external environment?

6. Dissertation specific

   (a) Am I allowed to instrument the application code?

   (b) Where can I deploy the application?

   (c) Are you using or have you used Azure Application Insights?

   (d) Am I allowed to set up a service on an external environment to perform external monitoring?

   (e) Will I have access to the working repository or a copy of it? In the latter case, how can I measure the impact of the design patterns on the project stakeholders?

   (f) One way to validate the design patterns is to provide them to someone and observe how they are interpreted and used. Would that be possible considering the project state?

   (g) Another way to validate them is to perform *member checking*. In other words, to ask experts in cloud, microservices or monitoring to provide their feedback on the patterns. Within the company, who do you think has enough experience on the mentioned topics to be considered an expert?

Appendix B

Team member survey

Monitoring challenges in [redacted]

Monitoring is a process that allows stakeholders to understand the state of the production system at any given time by collecting information from the many services involved. Effective monitoring enhances the performance, reliability and overall quality of the application. However, often it does not produce as much value as a feature, so it usually comes as an afterthought.

This survey aims at understanding existing monitoring challenges and shortcomings in [redacted] The information gathered in this survey will help me implement a monitoring solution that tackles the identified issues. It takes around 7 minutes to answer and is made by 3 sections with a total of 22 questions.

All data gathered is strictly confidential and will be used solely for academic purposes, more precisely, to develop an MSc thesis titled "Monitoring design patterns for cloud applications". They will be published exclusively in an anonymized form, as part of the MSc thesis or of articles that may arise from it.

*Required

1. How long have you been part of the project? *

Mark only one oval.

- ◯ less than 3 months
- ◯ 3 to 6 months
- ◯ 6 months to 1 year
- ◯ 1 to 3 years
- ◯ 3 or more years

2. What are your responsibilities within the project? *

Tick all that apply.

- [ ] Front-end Development
- [ ] Back-end Development
- [ ] Quality Assurance
- [ ] Operations
- [ ] Product Management
- [ ] Research & Development
- [ ] Other: _____

3. I know and understand the project's... *

In case an option is not implemented in the project vote N/A.

Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	N/A
architecture	◯	◯	◯	◯	◯	◯
code organisation	◯	◯	◯	◯	◯	◯
deployment strategy	◯	◯	◯	◯	◯	◯
local development environment	◯	◯	◯	◯	◯	◯
production environment	◯	◯	◯	◯	◯	◯
monitoring infrastructure	◯	◯	◯	◯	◯	◯

4. How do you detect a bug/issue in production?

   Select all that apply

   Tick all that apply.

   - [ ] Through user complaints
   - [ ] Through IT alerts (i.e. automated alerts that emanate from production systems)
   - [ ] By periodically checking system metrics collected over time on a dedicated dashboard
   - [ ] By periodically checking log files
   - [ ] By manually testing the production system
   - [ ] By automatically testing the production system
   - [ ] Other: _____

5. How do you usually find the root cause of a bug/issue in production?

   Select all that apply

   Tick all that apply.

   - [ ] Through log files
   - [ ] Through skewed metrics (i.e. metrics that show abnormal values)
   - [ ] Through traces of execution
   - [ ] Through user reports
   - [ ] Through exceptions
   - [ ] By repeating the user actions in a debug environment
   - [ ] Other: _____

6. On average, how many hours do you usually take to detect a bug/issue in production?

   Consider the time elapsed since the bug made it into production and the time you became aware of it.

   _____

7. On average, how many hours do you usually take to fix a bug/issue that was detected in production? *

   Consider the time elapsed since the bug was detected and the time you commited the fix on the version control system.

   _____

8. What is the average time, in hours, between system breakdowns?

   In other words, what is the system's mean time between failures? Formula = # of operational hours / # of failures

   _____

9. Approximately, what was the percentage of time that the system was operational during the last 30 days?

   In other words, what is the system's availability for the last 30 days? Formula = (total elapsed time – sum of downtime) / total elapsed time

   _____

Please select your degree of agreement with the statements below, regarding how quickly and easily the team can accomplish each of these activities.

10. Understand the application's code-level performance. *

    Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

11. Know the state of the application's infrastructure. *

    Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

12. Notice when a service has unexpectedly become inoperative or extensively degraded and needs recovery. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

13. Notice when a running service becomes unable to process requests. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

14. Detect bugs before they impact end-users. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

15. Manage and store the application logs. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

16. Read and understand logs. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

17. Correlate production problems with the deployment that caused it. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

18. Trace a failure seen in the front-end service to the service that caused the failure. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

19. Track exceptions and find their causes. *

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

20. I believe that the following monitoring practices are adopted in the project: *

Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Collection of business and performance metrics at the application code level	◯	◯	◯	◯	◯
Collection of operative system and infrastructure metrics from the production environment	◯	◯	◯	◯	◯
Periodic pings of a specific endpoint to ensure the system is healthy (i.e. not crashed or extremely slow	◯	◯	◯	◯	◯
Periodic pings of a specific endpoint to ensure the system is ready (i.e. it is ready to accept and fully able to process user requests)	◯	◯	◯	◯	◯
Running tests against the production system	◯	◯	◯	◯	◯
Record of user activity and system changes for auditing purposes	◯	◯	◯	◯	◯
Tracking of deployments and changes to the production environment	◯	◯	◯	◯	◯
Tracing the flow of a request through the system	◯	◯	◯	◯	◯
Sampling and prioritisation of logs	◯	◯	◯	◯	◯
Standard format for logs in the system	◯	◯	◯	◯	◯
Tracking and aggregation of exceptions	◯	◯	◯	◯	◯

21. Are there any other monitoring practices that you think are being used in the project?

Monitoring challenges in [redacted]

The following questions aim at eliciting requirements for a monitoring stack to be used in the project.

22. Historically, when there is a performance issue with one of the IT services you support, where is the problem most likely to be? *

Select your top three

Tick all that apply.

- [ ] Application code/framework
- [ ] Cloud
- [ ] Database
- [ ] End user mobile devices
- [ ] Infrastructure services (DNS, DHCP, profile, etc.)
- [ ] Middleware
- [ ] Network
- [ ] Server
- [ ] Storage
- [ ] Third-party service (external to your organization)
- [ ] Virtual server
- [ ] Other: _____

23. When there is a problem in your application code, which of these performance problems occurs most often? *

Select all that apply

Tick all that apply.

- [ ] High CPU usage issues
- [ ] Traffic spikes causing application issues
- [ ] Memory leaks in the application
- [ ] Database query slowness
- [ ] Database connections leaks in the application
- [ ] Third-party service slowness
- [ ] Exceptions/Errors
- [ ] Front-end slowness
- [ ] Deadlocks and synchronization issues in the application code
- [ ] Improper configuration/sizing of the application causing bottlenecks
- [ ] Issues with application code logic (inefficient data structures, loops, etc.)
- [ ] Other: _____

24. Do you have experience with the following tools? *

Select all that apply.

Tick all that apply.

- [ ] Amazon CloudWatch
- [ ] AppDynamics by Cisco
- [ ] Azure Monitor
- [ ] DataDog
- [ ] Dynatrace
- [ ] ElasticSearch
- [ ] elmah.io
- [ ] Grafana
- [ ] Graphite
- [ ] Honeycomb
- [ ] Jaeger
- [ ] Kibana
- [ ] Logstash
- [ ] logz.io
- [ ] New Relic APM
- [ ] Prometheus
- [ ] Splunk
- [ ] Nagios
- [ ] Other: _____

Monitoring challenges in [redacted]

Thank you for participating in the survey. Your help is crucial for the thesis work.
If you have any doubt, feel free to contact me at carlos.albuquerque@konkconsulting.com

25. Comments

Is there anything you would like to add? For example, if you think I missed an important question or there is some project specific issue that I should know about.

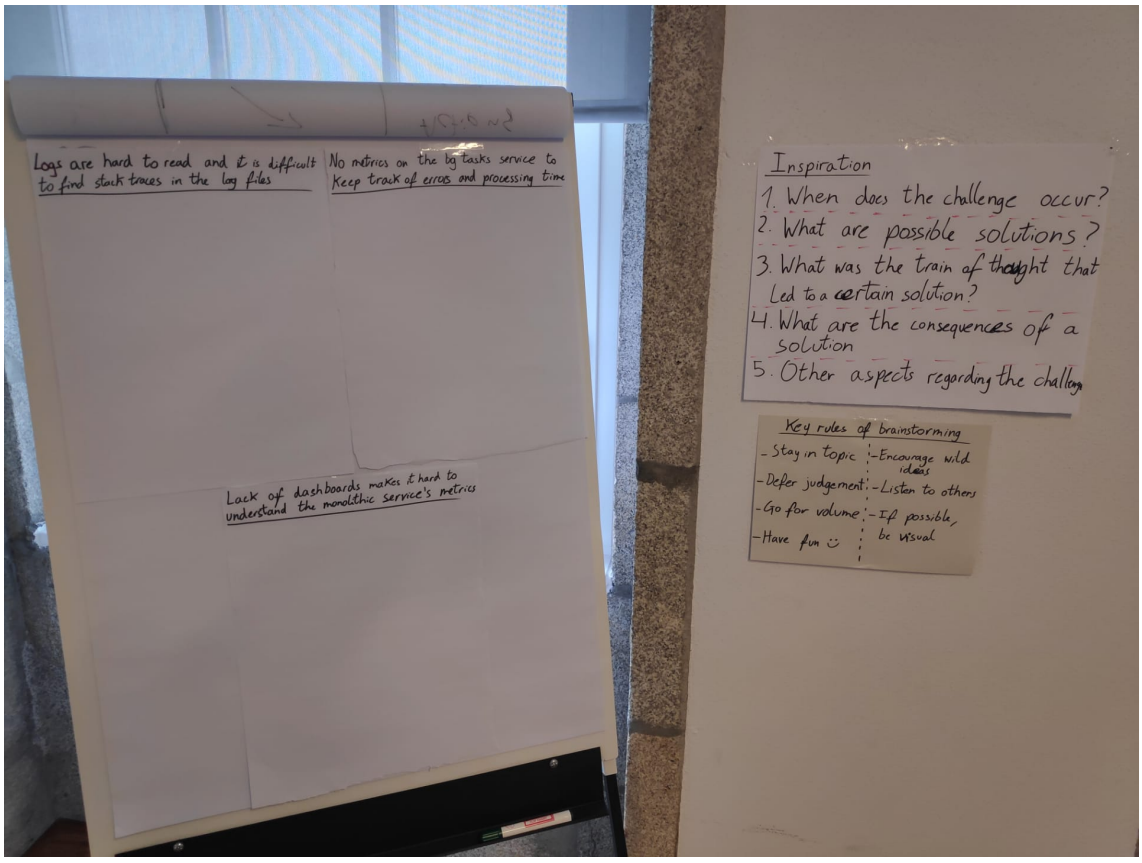
26. GDPR *

Tick all that apply.

☐ I agree with the collection and treatment of the data I provided in this survey by Carlos Albuquerque, Konkconsulting intern and FEUP student, for the development of his MSc thesis and creation of a monitoring stack for Konkconsulting projects.

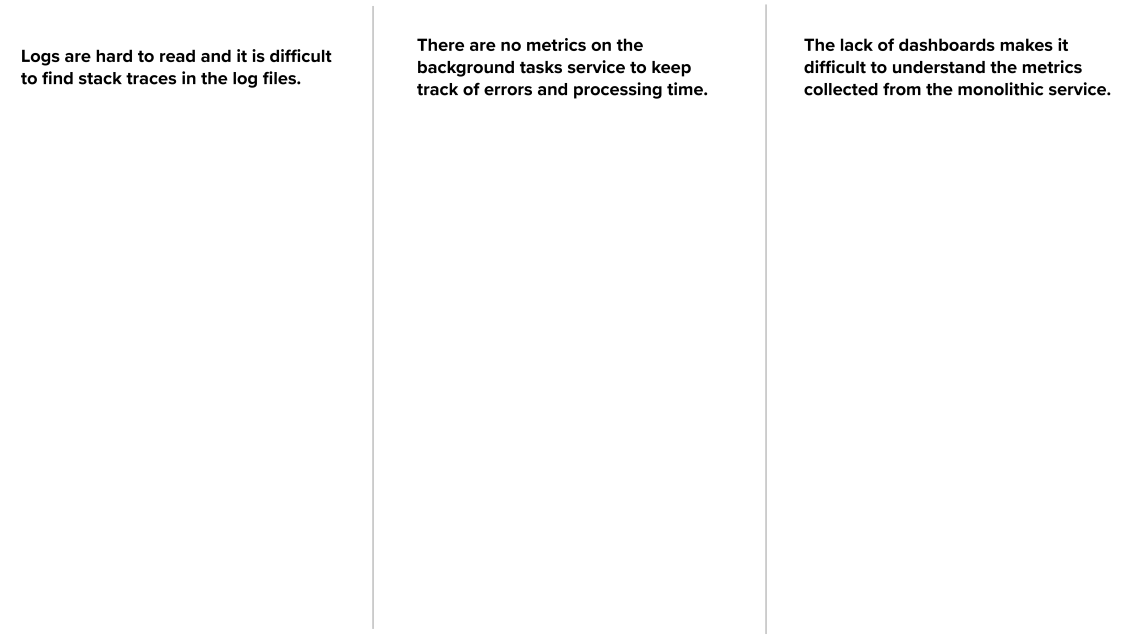
Appendix C

Focus group room's initial setup



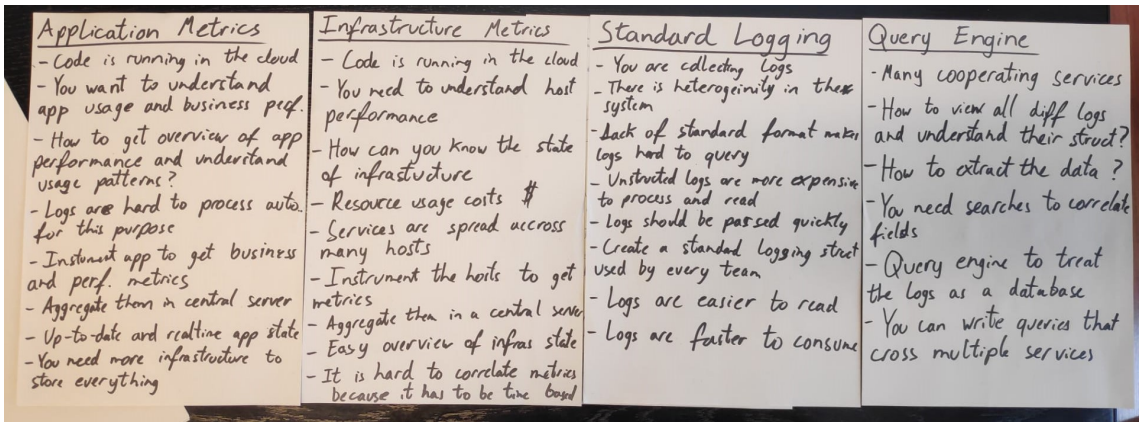
Appendix D

Focus group Mural board's initial setup



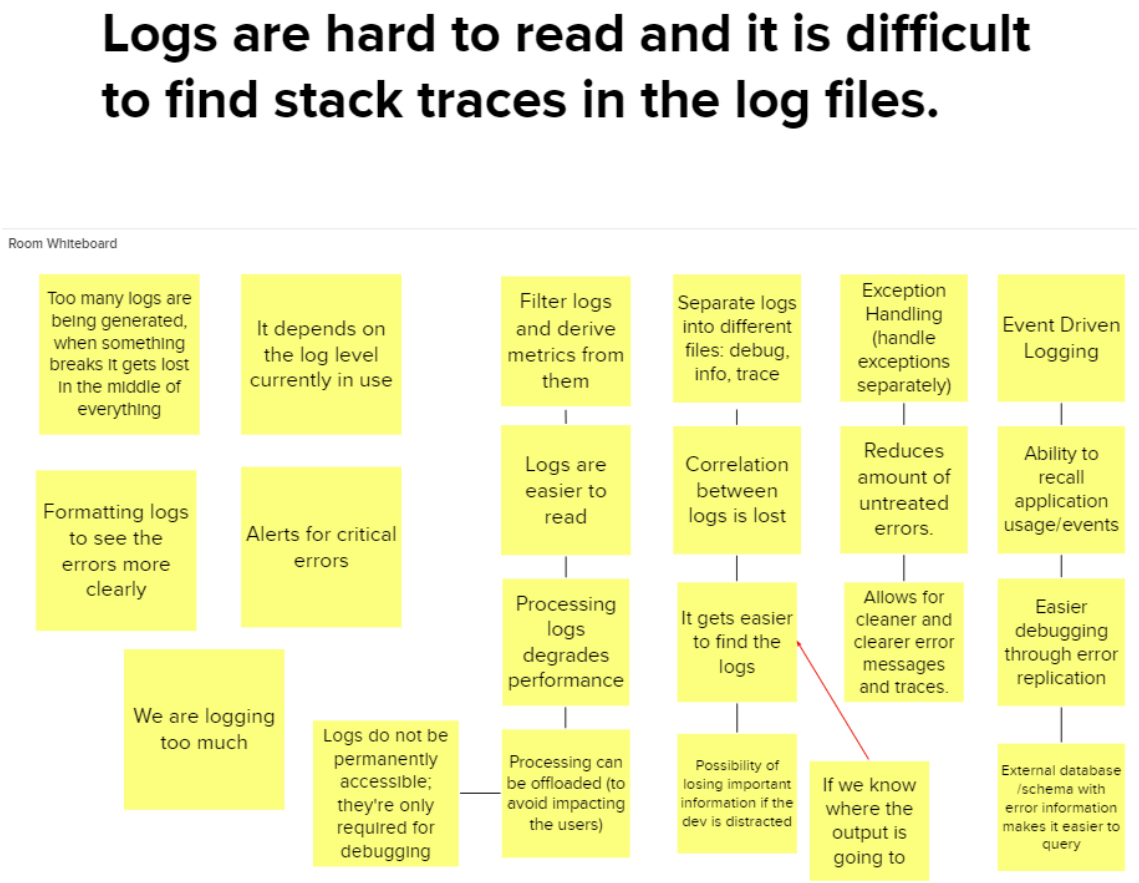
Appendix E

Focus group pattern sheets



Appendix F

Focus group results for challenge 1



Appendix G

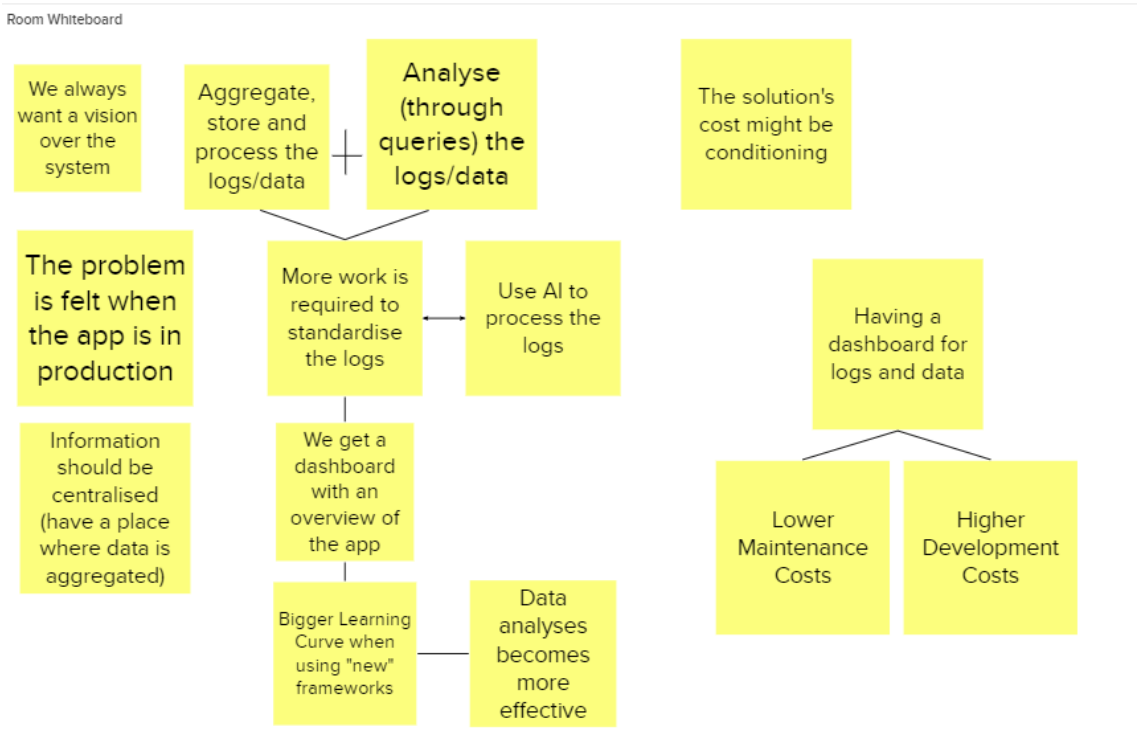
Focus group results for challenge 2

There are no metrics on the background tasks service to keep track of errors and processing time.

Appendix H

Focus group results for challenge 3

The lack of dashboards makes it difficult to understand the metrics collected from the monolithic service.



Appendix I

INFRASTRUCTURE METRICS design pattern's initial description

Infrastructure Metrics

Software needs infrastructure to execute. Even though the cloud abstracts this infrastructure from the developing team, applications still depend on the underlying resources. Scarcity of resources like CPU, memory or disk can bring extreme system slowness or even an outage. The team should be able to know the state of their application's infrastructure and correlate it with ongoing problems. Therefore, they should instrument the server and runtimes to capture relevant metrics of the operative system and underlying infrastructure and collect them in a centralised server, allowing the team to get a real-time overview of the application's environment.

Also known as
Inside-Out Health Check [43]

Context
Software needs infrastructure to execute. Even though the cloud abstracts this infrastructure from the developing team, applications still depend on the underlying resources. Scarcity of resources like CPU, memory or disk can bring extreme system slowness or even an outage. In addition, these issues can be caused by faults in the code (e.g. memory leaks), so it is the developing team's responsibility to find and fix them. The team should monitor the essential infrastructure resources to identify code-level problems and correlate end-user issues with the infrastructure.

In a pay-as-you-go scenario, which allows for greater flexibility and cost savings, it will drive the costs up if the application starts to utilise more and more resources. Being able to notice the increasing usage of infrastructure is necessary so the team can figure out if that is due to greater demand on the application (e.g. a new product release with a significant influx of users), in which case it is to be expected, or an issue with the code (e.g. poorly optimised loops or memory leaks) that needs to be fixed. Moreover, public clouds are usually under service level agreements (SLAs) that declare the minimum requirements the infrastructure must provide. If a

given resource constantly drops below the agreed threshold, the cloud consumer should be able to confront the provider and take action according to what is stated in the SLA. Thus, understanding the performance and state of the infrastructure is relevant for both cloud providers and consumers, so the machines the software is running on should be monitored.

Problem

How can the team know the state of their application's infrastructure and correlate it with ongoing problems?

Forces

- The system needs to scale when a particular resource threshold is crossed, which may require the team's manual intervention in certain cloud providers.
- Uncontrolled resource usage can significantly and unexpectedly increase the system's costs.
- Application modules or services may be spread across many infrastructure resources.
- Resource usage may vary immensely depending on the ongoing operation and the number of concurrent requests.
- Infrastructure provisioning may be under a contractual agreement (e.g. SLA) that needs to comply.

Solution

Instrument the server and runtimes to capture relevant metrics of the operative system and underlying infrastructure and collect them in a centralised server, allowing the team to get a real-time overview of the application's environment.

As explained in Application Metrics (see Section 5.13), **Metrics** are usually composed of a name, a value and a timestamp [72]. Tags can be stuffed into the metrics to provide further context. Tags are key-value pairs of properties, so they can come with nearly any kind of information. However, the higher the number of different values each tag can have, or, in other words, the higher its cardinality, the harder it is to store and process.

Instrumentation can take many forms, but it is essentially the act of adding measuring instruments to the system. These instruments collect the **Infrastructure** information and generate metrics out of it. In the context of this pattern, instrumenting the server and runtimes means having a **Metrics Exporter** that is responsible for gathering a predefined set of metrics. It must have access to the OS and underlying infrastructure and, ideally, be configurable to specify what should be captured and at what resolution (i.e. the number of data points per time interval).

Some public clouds, like GCP, provide an API with each application engine that exposes infrastructure metrics. Note that this API, by itself, is already a suitable exporter. It is gathering the necessary information and exposing it. Even though this kind of API may not be configurable (e.g.

you can not change the sampling rate), the team can request exclusively the metrics they need and at a larger interval than what the exporter samples.

Configuring the sampling rate may be necessary since storing and processing large amounts of data can demand a considerable monitoring infrastructure and slow the overall monitoring process. Therefore, when adopting this pattern, the team should decide how many data points should be collected and how many should be stored.

Another crucial decision is what metrics to monitor. As previously stated, the more metrics the team collects, the easier it is to understand the infrastructure's state. Thus, the team should start with the most useful ones for their case and expand as the need arises. The USE method is a common method to help with this decision [42]. It suggests that the following metrics should be the first to be collected:

- Utilisation - "the percentage of time that the resource is busy servicing work during a specific time interval" [42];
- Saturation - "the degree to which the resource has extra work which it cannot service, often queued" [41];
- Errors - "the count of error events" [42].

Note that the USE method is more than just an indication of the metrics that should be gathered. It also incentivises the team to start by identifying the resources in their system (e.g. CPU, memory, storage). Only then should they define what each of the above metrics means to each resource and how they should be gathered. That way, the USE method increases the overall understanding of the system and turns most unknown unknowns into known unknowns. In other words, when the team notices a missing metric that would be useful to solve a currently ongoing application problem, they can start from the list of metrics they know they are not gathering.

Finally, metrics generated and spread across many services are not very helpful. To make actual use of them, the team needs a **Metrics Service** - a centralised system to collect and aggregate (e.g., calculate the average, sum or percentiles) the metrics and present them in a friendly way to the team. This can be achieved in two ways [72]:

- push - the application sends the metrics to an API provided by the metrics service (e.g. AWS CloudWatch);
- pull - the metrics service fetches the metrics data from an API provided by the infrastructure (e.g. Prometheus)

Both options are viable. It all depends on the system's context. Usually, the pull model is more transparent to the application. In other words, it does not require as much instrumentation, keeping the business logic more isolated from these monitoring details.

Consequences

This pattern has the following advantages:

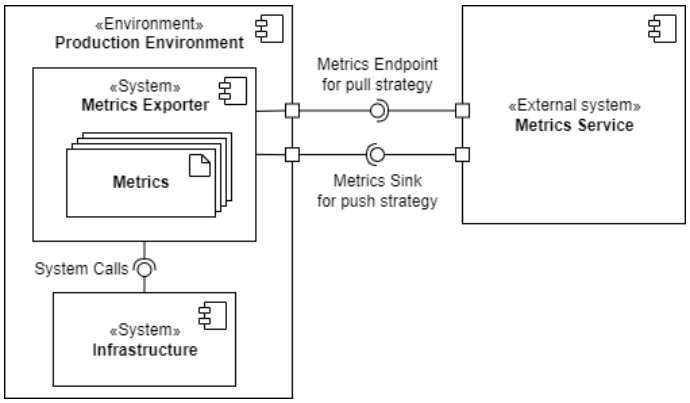


Figure I.1: Overview of the structure for the INFRASTRUCTURE METRICS pattern. The solutions usually only follows the pull or the push strategy. We represented both for completeness.

- It provides a real-time and straightforward overview of the infrastructure's state.
- Operations teams can more easily cut down on unused resources, decreasing the system's costs.
- Issues with the code that directly impact the infrastructure (e.g. memory leaks) become easier to detect.
- It allows for correlation between user symptoms, like recurring failures when creating a new post, to infrastructure problems, like a full database.

This pattern also has the following drawbacks:

- Correlating infrastructure and application metrics is based on the time of events; since the systems collecting the metrics are different, their clock times may also diverge slightly, making way for wrong relations between both metric types.
- Data can quickly become separated into different categories (or silos) without actual correlation, which heavily reduces this pattern's benefits.
- In PaaS and FaaS scenarios, the only way to collect infrastructure metrics may be through the cloud provider's dedicated solutions, which reduces the team's flexibility and restricts the kind of data that can be monitored.
- Analysing this metrics may be misleading because these convey possible problems (i.e. the lack of resources) but do not point towards the actual problem (i.e. the faulty code).

Example

In an online blog post, Stragliotto [93] provides an example of how THRON adopted this pattern. The author considers that "monitoring is the most important starting point to improve your product" [93], so they decided to update their monitoring architecture to improve their troubleshooting

and alerting capabilities. In addition, they also improve their ability to analyse long-term trends and build dashboards.

The team decided to use Prometheus to collect infrastructure metrics to achieve their goal. The decision to use that tool was mostly based on the team's needs regarding the software and infrastructure. According to the author, the features that they valued the most in Prometheus were [93]:

- "a data model based on time series data identified by metric name and key/value pairs".
- "a really flexible and powerful query language that helps to aggregate data, the results can be aggregated in real time and directly shown or consumed via HTTP API to allow external system to display the data".
- "no reliance on distributed storage; nodes are single server and autonomous".

Stragliotto explains that for their micro-service infrastructure, Prometheus was their best option. In addition, they adopted Grafana "to query, visualise and generate alerts from our metrics" [93]. From the grey literature, we found that combining these two tools is a ubiquitous way to set up a **Metrics Service**.

The author goes on to talk about the specific metrics that they chose to collect from their services. The post mentions the Four Golden Signals [31] as the most important to monitor a system, but for the infrastructure, they followed the USE method [42]. Stragliotto defends that this method is better suited when the need is to "keep the physical resources under control" [93]. Therefore, they collected utilisation, saturation and errors from many resources (e.g., CPU, memory, discs).

However, the author warns that:

> *"The USE Method helped us identify problems which could be system bottlenecks and take appropriate countermeasures, but it requires cautious investigation since systems are complex."* [93]

He argues that analysing these metrics is not obvious. Since they convey the state of the infrastructure, they point to possible problems, but they are not the actual problem. This is a consequence that we consider very important for the pattern.

To conclude, Stragliotto declares that they "are happy about how [their] new architecture turned out, it works and it's starting to really help [them] keep [their] software under control" [93]. This view of the system was essential to make their detection and troubleshooting processes faster and fix their services quicker.

Known uses

Stragliotto's [93] example described in the previous section is already a known use of this pattern that used Prometheus and Grafana to collect and visualise the utilisation, saturation and errors of their system's infrastructure.

An infrastructure software engineer post on callstats.io's blog [16] reports how the company uses Prometheus to monitor its services. They chose Prometheus for its seamless integration with

Kubernetes, powerful query language and operational simplicity. The engineer states that the company's infrastructure and operations teams "use [Prometheus] to monitor resource usage and service performance, as well as data pipeline processing queues " [16].

Finally, a post by Santiago Campuzano on GumGum's Tech Blog [17] explains how the company adopted Prometheus to monitor its systems. In addition, they utilised Grafana to visualise the data with customised dashboards. Although the author does not write about the actual metrics they collected, he mentions the usage of Prometheus' node exporter. This metrics exporter exposes "hardware and OS metrics on *nix systems" [17] and publishes more than 600 metrics. The author recognises that this amount of metrics can be a problem due to the high processing overhead they introduce.

Related patterns

The infrastructure metrics represent the state of the infrastructure. Analysing these logs helps notice symptoms of a problem. However, they are not usually enough to find what exactly is causing the issue. In such cases, the team should consider adopting APPLICATION METRICS to complement this pattern and get a full view of the system.

Appendix J

APPLICATION METRICS design pattern's initial description

Application Metrics

Since the cloud abstracts away the infrastructure on which the code is running, performance becomes more about how well the system can respond to the user than how efficiently it uses the available resources. An overview of how the application is performing and understanding emerging usage patterns is crucial to maintaining a quality system. Therefore, the team should instrument the application to gather business and performance metrics to then collect these metrics in a centralised service that provides aggregation and visualisation, allowing deeper insight into the application's performance.

Also known as

Inside-out health check [43], Monitoring Metrics [96]

Context

Nowadays, software systems not only need to bring value through appealing features but also to work extremely fast. Users expect things to answer quickly and fail rarely. In such a demanding environment, teams must be able to keep track of their system's performance to ensure it meets certain requirements. These may be informal and more flexible, such as an end-user of a web application that expects a page to load in less than 30 seconds, or formal and strict, such as a service level agreement (SLAs) between an infrastructure as a service (IaaS) provider and its client. Both cases are equally important for a company since it might be losing customers or having to compensate them for a contractual breach, respectively.

Since the cloud abstracts away the infrastructure on which the code is running, performance becomes more about how well the system can respond to the user than how efficiently it uses the available resources. Moreover, performance can be seen from the technical and business lenses. Understanding customer trends and satisfaction is just as important as knowing availability and

the number of errors, for example. Both can bring helpful insight to a company and give it a more significant market presence.

Problem

How can the team get an overview of how the application is performing and understand the emerging usage patterns?

Forces

- Processing logs to extract relevant system performance measures is a computationally expensive process and does not provide real-time data.
- Simulating requests and testing in production provide a small amount of data and only for already previously defined scenarios.
- Infrastructure is managed by an external entity that usually provides infrastructure-level metrics.

Solution

Instrument the application to gather business and performance metrics. Collect these metrics in a centralised service that provides aggregation and visualisation, allowing deeper insight into the application's performance.

The team can collect **Metrics** on many things, such as request rate, error rate, customer orders, and user posts. Ideally, one would measure everything, but that introduces more overhead and requires considerable infrastructure. That said, the first decision the team must make when adopting this pattern is to define which **Application** metrics they wish to monitor. Consider The Four Golden Signals proposed in [31] as a starting point. They are briefly presented below, but for further detail refer to the original work:

- Latency - the time between a user request arrives in the system, and a response is issued to the user.
- Traffic - a "high-level system-specific metric" [31] that measures the number of incoming requests in your system.
- Errors - the rate of requests that result in a failure, of any kind, by a period of time.
- Saturation - the percentage of system resources being utilised at the moment.

There are other guidelines based on the Golden Signals, like the RED method [45] and the READS metrics [44], so the team should consider which one is more suited to their context.

On top of the Golden Signals, collecting metrics aimed at business objectives, user actions, or high-level application-specific values may be beneficial. In the case of websites, some of these are already provided by systems that collect analytics. In contrast, others can be included manually

(e.g. user orders, user turnover, user satisfaction, user posts or wishlists). Remember that the more the team collects, the harder it will be to handle.

Once we know what to measure, we should understand how to do it. Each metric is sampled periodically by a **Metrics Exporter** and is usually composed of a name, value and timestamp [71]. Further context must be added to the metric through tags, i.e. key-value pairs for external properties, that are collected each time step. The higher the cardinality of the added tags (e.g. usernames and emails, which are unbounded sets), the more the metrics system will struggle to keep all the information. When adopting this pattern, the team should consider reducing the cardinality of these tags or using a system that is prepared to receive high cardinality data.

Another thing to consider is the resolution of data, i.e. the number of data points collected by time period. Depending on the current context of the system, some metrics may be collected and stored in a greater resolution than others. As Newman exemplifies:

> *"I might want a CPU sample for my servers at the resolution of one sample every 10 seconds for the last 30 minutes, in order to better react to a situation that is currently unfolding. On the other hand, the CPU samples from my servers from last month are likely needed only for general trend analysis, so I might be happy with calculating an average CPU sample on a per-hour basis."* [64, p. 322]

Therefore, when adopting this pattern, the team should consider having the ability to change the resolution of the data that is stored or being collected. That allows the team to better manage storage space and overhead while still collecting a sufficient number of metric points for troubleshooting.

Finally, metrics generated and spread across many services are not very helpful. To make actual use of them, the team needs a **Metrics Service** - a centralised system to collect and aggregate (e.g. to calculate the average, sum or percentiles) the metrics and present them in a friendly way to the team. This can be achieved in two ways [72]:

- push - the application sends the metrics to an API provided by the metrics service (e.g. AWS CloudWatch).
- pull - the metrics service fetches the metrics data from an API provided by the application (e.g. Prometheus).

Both options are viable; it all depends on the context of the system. Usually, the pull model is more transparent to the application. In other words, it does not require as much instrumentation, keeping the business logic more isolated from these monitoring details.

Consequences

This pattern has the following advantages:

- It provides an up-to-date and straightforward overview of the application's state.

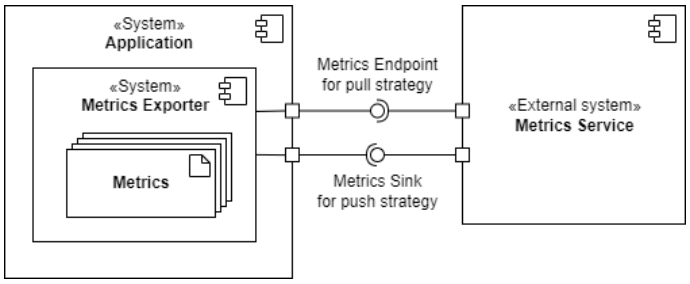


Figure J.1: Overview of the structure for the APPLICATION METRICS pattern. The solutions usually only follows the pull or the push strategy. We represented both for completeness.

- Application metrics can be helpful from both the technical perspective (e.g. capacity planning) and business perspective (e.g. business insights).
- Application metrics are a potential source of information for strategic decisions, motivating data-driven decision-making.
- The collected metrics allow the team to predict possible problems, such as disk space running out, and figure out trends of, for example, capacity planning.
- The team can identify issues before they affect the system's users when application metrics are paired with alerts.

This pattern also has the following drawbacks:

- Aggregating and storing a high number of metrics may require considerable infrastructure.
- Instrumenting the source code to measure the metrics makes the business logic more complex and harder to understand.

Example

A post by Santiago Campuzano on GumGum Tech Blog [17] explains how the company adopted Prometheus to monitor its systems. Before the company adopted the microservice architecture, the author states that monitoring was quite simple. Their "applications, servers, and services were pretty much fixed and well-known" [17]. However, things got much more complicated once they started containerising their applications and adopting the microservices architecture. As the author states: "the explosion in the quantity and complexity of the systems to be monitored was neither manageable nor sustainable with the legacy monitoring stack that we had in place" [17].

To address the new monitoring needs, they used Prometheus to create a monitoring stack capable of handling the complexity. According to the author, they "chose Prometheus among other systems, mostly because of its flexibility, extensibility and huge open source community backing the project" [17]. They used different Prometheus exporters to instrument the code to generate metrics. The tool follows a *pull* strategy to get the metrics, i.e., it periodically scrapes them from a configurable endpoint. GumGum's monitoring stack collected metrics from the application,

containers and servers. Additionally, they used Grafana to visualise the collected metrics on customised dashboards and Alert Manager to configure and manage alerts on top of the Prometheus metrics. Figure J.2 is part of the post and depicts the overall Prometheus architecture used by GumGum.

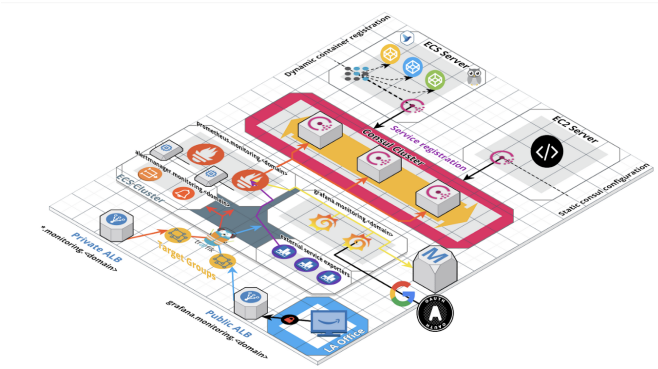


Figure J.2: Real-world example of the APPLICATION METRICS pattern. [17]

Unfortunately, the author does not precisely mention which metrics they were collecting. Nonetheless, Campuzano mentions that they used Prometheus Pushgateway for "ephemeral and batch jobs to expose its metrics to Prometheus" [17]. These jobs send the metrics to the Pushgateway, which Prometheus then scrapes. Thus, they could collect application-level metrics and visualise them in specific dashboards for their teams. With Prometheus and Grafana, GumGum became able to collect metrics from many parts of their system and "create beautiful and meaningful Grafana dashboards with those metrics" [17].

Known uses

Campuzano's [17] example described in the previous section is already a known use of this pattern that used Prometheus and Grafana to collect metrics from a microservice architecture.

A post by Domenico Stragliotto [93], a backend developer for THRON, elaborates how the company adopted metrics on their system. The author explains they were looking to improve their ability to analyse long-term trends, build dashboards, troubleshoot and get alerted. To achieve their goals, they used Prometheus to collect metrics from their services. They followed the RED method [45], so, for each service, they collected the number of requests per second, the number of errors per second and the amount of time to process each request. These metrics were then exposed in Grafana dashboards so the teams could easily view each service's state.

Finally, an infrastructure software engineer post on callstats.io's blog [16] reports how the company uses Prometheus to monitor its services. They chose Prometheus due to its seamless integration with Kubernetes, powerful query language and operational simplicity. The engineer states that the company's developers "use Prometheus to compare performance and resource usage between service releases " [16], while the analytics team uses it "for several artificial intelligence-related metrics " [16].

Related patterns

Although the cloud abstracts the infrastructure where the application runs, understanding the performance of the host machine can be helpful. The underlying machine's performance can impact the application's performance. Suppose the team needs a more comprehensive view of the system or further correlation between the application and its infrastructure. In that case, they should consider adopting INFRASTRUCTURE METRICS to complement the metrics provided by this pattern.

Appendix K

AUDIT LOGGING design pattern's initial description

Audit Logging

As more and more people use an application, more problems can arise. Some users will require customer support. Others may try to access the information they are not supposed to see or show suspicious behaviour that the team should be aware of to reinforce existing security measures. Recording the behaviour of users and reproducing it to uncover relevant information is vital in such scenarios. Therefore, the team should record user activity and relevant system changes in a data store "in order to help customer support, ensure compliance and detect suspicious behaviour" [72].

Also known as
Audit Logs [35], Audit Trail [4]

Context
As more and more people use an application, more problems can arise. Some users will require customer support, and a reproducible trace of their actions is helpful to solve the problem. However, sometimes the users cannot provide enough detail for the support to be of any use. Other users may try to access the information they are not supposed to see or show suspicious behaviour that the team should be aware of to reinforce existing security measures.

Being able to reproduce and understand user behaviour can be even more critical when dealing with sensitive data under regulatory policies. Consider, for example, that a care provider in a hospital leaked patient records to the press. In such a case, the hospital administration must be able to discover who did it to take legal action. If the underlying system does not register who did what, then finding out the culprit becomes a nearly impossible task.

Problem
How can the team record the behaviour of users and reproduce it to uncover relevant information?

Forces

- The goal of analysing this information is only known after an incident occurs, so it must be collected preemptively.
- The solution must have minimal temporal and storage overhead in order to avoid user experience degradation.
- If records are to be used as evidence in litigation, they must be stored securely and be immutable.

Solution

Record user activity and relevant system changes in a data store "in order to help customer support, ensure compliance and detect suspicious behaviour". [72]

These records are commonly called **Audit Logs**. They can take many formats, but they are a way to answer "who did what, where and when?" [40]. Therefore, when something important changes, the system should register who caused the change, which action was taken by that agent (either human or not), where was that action applied (i.e. which resource changed) and, finally, when the change took place.

The biggest challenge in adopting this pattern is figuring out the amount of information to register in each log. Logs may be of little to no use in auditing processes if relevant events are missed. On the contrary, if all system events are registered, the system gets slower, and audit logs are much more challenging to process and handle. Therefore, when adopting this pattern, developers should consider which parts of the system need audit logs the most and what events to register. From that point onwards, it is their responsibility to balance the trade-off between data collection and processing cost, aiming for the most comprehensive logs with the most negligible overhead possible.

Adopting this pattern requires a **Log Generator** and a **Data Store**. The former is responsible for generating the audit logs and sending them to the latter, which securely stores them for audits, usually following an append-only strategy. There are a few ways to implement this pattern [72] according to how the logs are generated and stored: (1) manually instrumenting the source code, (2) using aspect-oriented programming or (3) using EVENT SOURCING [59].

The simplest option is manually instrumenting the source code to write an actual log. These logs can be stored in a file or folder, which is quite common and straightforward, or saved in a database to be queried later. If developers opt for the former, then a standard format is required. An ASCII string is easily readable by humans and should be enough for a simple structure. For more complex structures, JSON or XML formats can be more convenient. Overall, instrumenting the code is easy and cheap, but Amir-Mohammadian et al. [7] argue that manual instrumentation may miss relevant information.

The second option is aspect-oriented programming (AOP). AOP is a programming paradigm that helps modularise crosscutting concerns (e.g. security and logging) across the application [48].

Audit logs are one of these concerns. When implementing it manually, the logging code must be replicated in many parts of the program, cluttering the business logic and making it harder to maintain. By implementing audit logs as an aspect of the system, developers can automatically record every method invocation without changing the business logic. However, because AOP usually only provides information about the method name and arguments, it is harder to log business-oriented events [72].

The last option is using the EVENT SOURCING pattern. This pattern "defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store" [71]. Thus, the events become the audit logs, and since they are immutable and the store is append-only, they can be used in legal actions. Developers do not need to do anything else - just by having EVENT SOURCING, they get AUDIT LOGGING. However, implementing it is a significant challenge that requires a change of mindset when dealing with business logic. The EVENT SOURCING pattern has already been explored in existing literature [72, 59], so we do not dive into further detail in this dissertation.

To conclude, every option has its advantages and drawbacks, so the team must choose what is more suitable for their context. Nonetheless, audit logging should be adopted at least in its simplest form as early as possible and not as an afterthought.

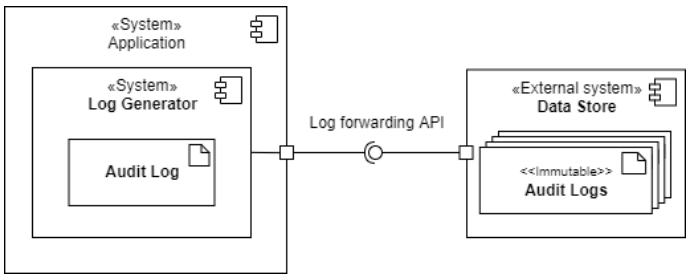


Figure K.1: Overview of the structure for the AUDIT LOGGING pattern. The stored audit logs should become immutable if they are to be used in auditing processes.

Consequences

This pattern has the following advantages:

- User actions become more accountable than before, so administrators can understand who did what in the case of an incident.
- Audit logs can be used as evidence of regulatory compliance or legal action.
- User activity becomes reproducible, which can be used to fix faults in the code or improve customer support.
- A registry of user actions provides a source for business-driven metrics that may be relevant for managers.
- Audit logs are a straightforward and effective solution for tracking temporal information.

This pattern also has the following drawbacks:

- Audit logs are difficult to process, and tools to do so may require considerable maintenance or infrastructure.
- Records must contain enough information *a priori*. Otherwise, when an incident occurs, they are useless. This necessity can lead to records with more information than needed, which leads to massive logs that are harder to analyse and store.
- As time goes by, logs pile up and will occupy more and more space, so they must be carefully managed to control storage overhead.
- Audit logs may contain sensitive information that should be appropriately secured to avoid possible information breaches through the audit data store.

Example

Amir-Mohammadian et al. [7] provide a real-world example of how audit logs can be helpful in electronic medical record systems. These provide *break the glass policies* so that "care providers can *break the glass* in an emergency situation to temporarily raise their authority and access patient records" [7]. The care providers know that, after they *break the glass*, their "subsequent sensitive operations will be logged and potentially audited" [7].

Suppose that a patient's medical records are leaked. In such a scenario, the hospital must pinpoint the source of the leakage, possibly by knowing who accessed the patient's records during a *break the glass*. "Since it cannot be predicted *a priori* whose information may leak, this goal can be supported by using an audit log that records all reads of sensitive files following glass breaking" [7]. Therefore, these systems must generate audit logs to record "all patient information file reads following a break the glass event, along with the identity of the user that broke the glass" [7]. The actual implementation is not part of the original example. It could be done by instrumenting the code with some logging calls that get activated when the *break the glass* event triggers. These calls would log data into an encrypted file, ensuring that no one would be able to access sensitive information through the logs.

Finally, whenever a leakage would occur, the logged information would be analysed to find all users that read the patient's leaked data. However, "if logging is incomplete then some potential recipients may be missed" [7]. On the other hand, "if logging is overzealous then bloat is possible and audit logs become *write only*" [7], i.e., logs become much harder to process and to store. The authors point out that "these types of errors are common in practice" [7]. Thus, balancing the amount of information in the logs is essential to this pattern.

Known uses

A blog post by Fauna Inc [33] explains how Breezeworks uses audit logs as a feature of their SaaS solution. The post reports that Breezeworks' "end users often want to know when an appointment or estimate was changed" [33], so Breezeworks needs to store all relevant changes in a record to

provide a change history to its users. "They accomplish this by adding an *after_commit* hook to their Ruby on Rails ActiveRecord base model, which maintains a copy of the data in FaunaDB Cloud" [33]. This strategy resembles an AOP approach to implementing the pattern. Whenever a change to a record is committed, the *after_commit* code runs and sends the data to FaunaDB. Hence, the code in the *after_commit* hook acts as the **Log Generator**, and FaunaDB is the **Data Store** where information is kept. This is an example of how audit logs can be used for more than just strict auditing purposes.

Chris Dermody [25], director of product at Flipdish, states that the company uses audit logs to provide customer support. "When a customer calls in and asks why their app seems broken or buggy, being able to track down who changed what, and when, is a powerful tool to have in your arsenal to get the issue resolved as quickly and painlessly as possible" [25]. Unfortunately, the author does not provide more detail on implementing the audit logs.

Finally, a blog post by Ahmed [4], founder of Apptrail, provides some examples of systems that use audit logs, or, as he calls it, audit trails. One of the examples is Google Workspace which the author claims "has a mature audit logs offering" [4]. A screenshot of the platform is shown in Figure K.2. The dashboard shows the event name, description and time. Some of the events provide information on what changed and which were the old and new values. In addition, the dashboard shows who did the action, i.e. the actor, and the IP address associated with the actor.

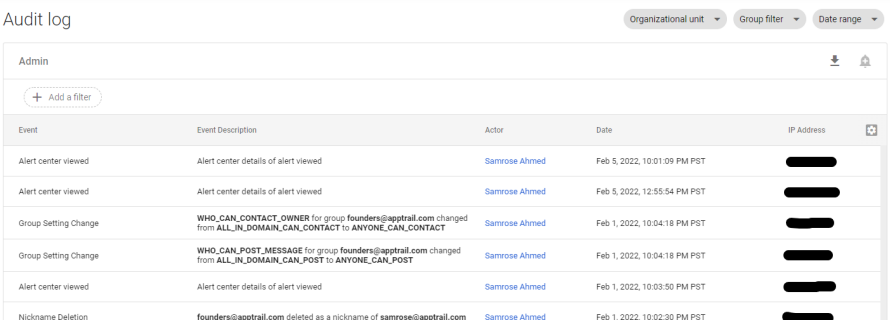


Figure K.2: Google Workspace's dashboard for audit logs.

Related patterns

As discussed in the solution section, this pattern can be implemented by adopting EVENT SOURCING [59]. Essentially, instead of only saving the current state of resources, the team should keep track of every event that changes them. These events become the audit logs, and actions become more reproducible than before. However, adopting EVENT SOURCING requires a significant change of the business logic and programming mindset, so the overhead might not be worth it for the team.

To query the audit logs, the team should consider adopting QUERY ENGINE [14]. The query system would parse and index the logs' data and make them easier to consume by the team. If the team does not feel the need for such a complex system, adopting the STANDARD LOGGING pattern may help with the logs' readability without the processing overhead. Lastly, to balance the

amount of information and ensure enough detail in the audit logs, the team can consider following the PREEMPTIVE LOGGING pattern.

Appendix L

STANDARD LOGGING design pattern's initial description

Standard Logging

Since different teams and services may generate logs differently, these come in all shapes and sizes. Sorting through a handful of logs may be easy at first, but as the system grows, so does the number of collected logs. The heterogeneity of log formats makes them harder to read and process by humans. In such a scenario, the team needs to improve log readability to make logs easier to read by humans and easier to parse by other services. Therefore, they should implement a consistent logging format across all services and teams, so logs are understood by everyone and consumed by other services independently of their origin.

Also known as
Structured Logging

Context
Virtually every service or application generates logs. These are easy to implement and widely supported [91]. Since different teams and services may generate logs differently, these come in all shapes and sizes. Sorting through a handful of logs may be easy at first, but as the system grows, so does the number of collected logs. The heterogeneity of log formats makes them harder to read and process by humans. Even if the logs are aggregated in a centralised repository (cf. LOG AGGREGATION [83]), it gets to the point that manually reading through the logs is a tiresome and error-prone process.

The team may be considering adopting a QUERY ENGINE [13] to help them extract relevant information from the logs. Even in that scenario, if logs follow different formats depending on the service they are generated from, the query system will take much longer to parse the data. This additional processing effort consumes resources that could be used for other operations more valuable to the system.

Problem

How can the team improve log readability to make logs easier to read by humans and easier to parse by other services?

Forces

- Parsing and consuming the logs must be efficient to avoid impacting the application.
- The solution must be reliable even if the services are written in different programming languages.
- Reformatting logs is a computationally intensive task, so it should be avoided [64].

Solution

Implement a consistent logging format across all services and teams, so logs are understood by everyone and consumed by other services independently of their origin.

As far as we know, there is not yet a common industry standard dedicated to logging [64]. Thus, the team can freely define the logs' structure and format. The **Logging Format** should be configurable in the **Logger**, i.e., the component that generates the **Logs**, be it a method call or a dedicated logging library.

Text logs are the simplest to use and work well enough for manual processing. However, processing a JSON or XML formatted log may be easier for other services. Optionally, the logger may be able to dump logs in different formats depending on where they are dumped. For example, it can use text logs when writing logs to the console but a JSON format when dumping them to an endpoint for their consumption by other services.

The information that the logs carry must be defined by the team. They must differentiate mandatory from optional properties and ensure that each property's purpose is easily understandable by everyone. Nonetheless, OpenTelemetry has defined a generic logging format that we believe is a helpful place to start [65]. The following properties are heavily inspired by its format [65]:

- Timestamp — "Time when the event occurred"; ideally follows a standard for date and time such as *ISO-8601* [77].
- TraceId — "Request trace id"; this is an optional field that helps to correlate logs; it is the same for every operation executed during a single request lifetime.
- SpanId — "Request span id"; this is an optional field that helps to correlate logs; it is unique for each operation executed during a request lifetime.
- SeverityText — "The severity text (also known as log level)"; this makes it easier to separate logs according to how severe the event was (e.g. Information, Error, Fatal).
- SeverityNumber — "Numerical value of the severity"; this is an optional field that helps when sorting logs by severity.

- Body — "The body of the log record"; this field can take any format and is the ideal place to insert more details about the event.
- Resource — "Describes the source of the log"; this is an optional field that helps identify the source of the log; it can be, for example, the name of a service, class or machine.
- Attributes — "Additional information about the event"; this is an optional field that, together with the Resource field, helps provide more context to the log.

Finally, the logging format should be adopted across all teams and SERVICES of the system to fully leverage its benefits.

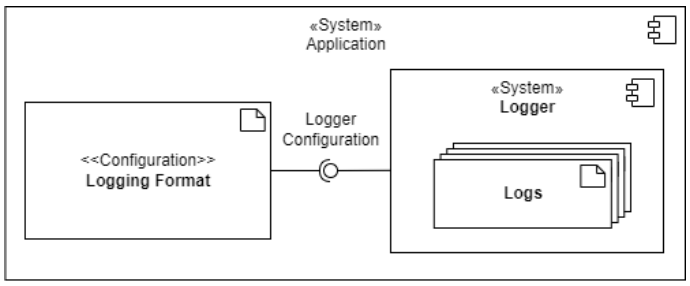


Figure L.1: Overview of the structure for the STANDARD LOGGING pattern. The logger generates logs based on a configurable logging format.

Consequences

This pattern has the following advantages:

- Logs' readability increases, so the team can extract relevant information from the logs more easily.
- A consistent format ensures that the fundamental properties of logs will be present across all system services.
- Logs become easier to parse and index, reducing the overhead of having other services processing the logs.
- Since logs become faster to process, the system's mean time to resolve (MTTR) is reduced.

This pattern also has the following drawbacks:

- A rigid logging format may not be versatile enough to accommodate the needs of all services and teams. This should not be a problem as long as there are open fields in the format.
- The initial overhead of adopting the standard format may be considerable depending on the number of teams and services that will need to adapt.
- Some logging structures (e.g. binary, XML) are harder to read by humans but better for machines. In that case, the team may consider having two different formats for each use case.

Example

This example is based on another example provided in [64]. There are two services in a system, one for handling orders and the other for payments. They are developed by different teams in different programming languages. On the one hand, the orders team decided to log incoming orders following the format:

```
<TraceID> <Date> <Time> <LogLevel> <ServiceName> <Body>
```

On the other hand, the payments team uses the following format for their service's logs:

```
<ServiceName> <LogLevel> <Date> <Time> <Body> <TraceID>
```

Both teams would like to be able to query their logs, so they decided to implement a query service to ingest the logs from the log files and index their fields for advanced queries. However, they soon realised they needed to configure two different rules to parse each log depending on the service that generated it. Even worse, the name of the service does not appear in the same place in both formats, meaning they would have to come up with some hack to know which parsing rule to use.

Instead, both teams agreed to follow the following logging format:

```
<Date> <Time> <ServiceName> <LogLevel> <TraceID> <Body>
```

Now they can parse and index the logs with a single rule. They do not need to differentiate between the source of logs, and both teams can easily read the logs when they show up together in a log file, as portrayed in the listing below.

```
1 15-02-2020 16:00:58 Order INFO [abc-123] Customer 2112 has placed order 988827
2 15-02-2020 16:01:01 Payment INFO [abc-123] Payment $20.99 for 988827 by cust 2112
```

Example of logs from the Orders and Payments services after adopting a common logging format. [64]

So far, finding the main fields of the logs is straightforward — the date is the first column, time is the second and so on. However, it is slightly more challenging to parse the *Customer ID* and *Order ID* from the logs' bodies. The strings used for those fields are easy to read by a person but harder for a machine to parse. In this scenario, both teams could use a JSON object for the body and agree on common property names for the customer and order IDs. It all comes down to the exact needs of both teams regarding query complexity.

Known uses

Waseem et al. conducted an industrial survey with ten different microservices systems from ten different companies [96]. Their study concluded that "the companies widely follow the log content and format defined by the OpenTracing specification" [96]. In the meantime, the OpenTracing specification was archived to make way for OpenTelemetry [22]. Nonetheless, the authors

noticed that even the interviewed companies that used customised formats were "migrating to the OpenTracing specification to benefit from its active open-source community" [96]. This tendency reveals that companies adopt a standard logging format and that an industry-standard may be on the horizon.

As part of this dissertation, we analysed and characterised a project from Konkconsulting. Section 6.3 describes the project and our main findings regarding its context and existing monitoring practices. We conclude that the team already adopts STANDARD LOGGING. They use Serilog to keep a consistent format across all their logs. They use text logs that get dumped into separate log files depending on the log's severity level. The logging format is configured in the application's configuration files and is as follows:

```
{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}]
{SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}
```

Finally, in a blog post, Praharaj [69] explains how Grab, a southeast Asia Online to Offline mobile platform, started using structured logging to solve some of its problems. The author states that they used "syslog-style key-value format logs, recognised as the most common format of logs for server-side applications due to its simplicity and ease for reading and writing" [69], for their Golang services. This worked for a long time, but as the system's complexity increased, their "logging bills started mounting to unprecedented levels" [69]. Hence, they changed their logging strategy and bootstrapped a Golang library that, between other features, ensures a consistent log structure. The library output logs in JSON format with a consistent structure that they built "on top of a schema in which [they] can add key-value pairs with a specific key name and type" [69]. Doing so enabled them to use an Elastic stack backend to index and query the system's logs.

Related patterns

This pattern integrates exceptionally well with LOG AGGREGATION. If the logs do not follow a consistent format, aggregating them makes them even harder to read. Thus, the team should consider adopting these two patterns together to help log readability and interchangeability across all other teams.

Moreover, if the team wants to adopt the QUERY ENGINE pattern, they should consider adopting STANDARD LOGGING first. Having a common and well-defined structure across all system services makes implementing a query system much more straightforward, reducing the initial overhead of adopting the QUERY ENGINE pattern.

Appendix M

DISTRIBUTED TRACING design pattern's initial description

Distributed Tracing

Cloud applications are usually a sum of modules that work together to reply to user requests. Therefore, a request may require many nested operations throughout the application's modules, both internal and external. When something breaks during these operations, it can become hard to figure out exactly where the failure is. The team should be able to record and visualise the end-to-end behaviour of the application to identify problems during the request lifetime. Therefore, they can assign each external request a unique ID and record how it flows through the system from one service to the next in a centralised server that provides visualisation and analysis, making troubleshooting the application faster and less complicated.

Context

Cloud applications are usually a sum of modules that work together to reply to user requests. Therefore, a request may require many nested operations throughout the application's modules, both internal and external. When something breaks during these operations, it can become hard to figure out exactly where the failure is. Moreover, the person troubleshooting the error may not even be familiar with all the modules, making the troubleshooting process longer and more complicated.

Sometimes, even if the system did not fail, it might still be underperforming. If a user needs to wait five or more minutes for a webpage to load, there might be an operation (among all executed for that request) that is taking too long. The cause of slowness may be hard to pinpoint when requests are complex and involve many operations.

Problem

How can developers record and visualise the end-to-end behaviour of the application to identify problems during the request lifetime?

Forces

- Monitoring metrics usually provide an aggregate view of the system (e.g. max response time in the last 15 minutes, the sum of request errors in the last hour), so they are not helpful in monitoring a single request.
- Log entries of a specific request, even when correlated through a unique ID, are stored across different locations on many individual logs.
- Logs usually record events or errors. Thus it is harder to visualise how much time each operation took to identify bottlenecks.
- Even when running outside the system's production environment, unit and synthetic tests may miss temporary events that impact the user (e.g. high CPU or RAM usage, network issues in the cloud infrastructure). Hence, developers need information from when the system processes the request to identify potential causes of failures.
- Any solution should impact the system's performance and maintainability as little as possible.

Solution

Assign each external request a unique ID and record how it flows through the system from one service to the next in a centralised server that provides visualisation and analysis, making troubleshooting the application faster and less complicated.

One way to achieve this is by instrumenting the source code as follows:

- An **Unique ID** is assigned to each incoming request (cf. CORRELATION ID [13]);
- The ID is sent to every **Service** that processes the request;
- Local thread activity is recorded in a **Span** of execution and tagged with the unique ID;
- The unique ID is included in all the logs generated while processing the request;
- The span information is sent to a central **Tracing Service** directly from the microservice or through a local **Forwarding Agent**.

The collector is then responsible for processing all spans and constructing **Execution Traces** by mashing together the related spans and logs. Usually, the collector also provides a UI to visualise request information and the ability to query the traces. The collector lets developers analyse the whole execution trace and quickly figure out performance bottlenecks, which operation originated a failure and the logs associated with the previous situations.

When recording local thread activity in the spans, developers should consider following the OpenTelemetry API. This standard is based on the existing OpenTracing and OpenConsensus APIs

and is widely used by the industry [64]. In addition, many existing libraries and tools already come with support for OpenTelemetry, making it much easier to adopt Distributed Tracing.

The biggest challenge when adopting this pattern is handling the information collected. The system will be impacted if the developing team collects and stores traces for every request. A possible solution to this issue is tail-based sampling, i.e. sampling the execution traces after they have been reconstructed in the collector service. The team can decide to drop some of the traces and only store the remaining ones, effectively reducing the overhead on the system. "The idea here is to capture enough information to understand what our system is doing but not capture so much information that the system itself cannot cope" [64]. The sampling can be random or dynamic. The former consists of randomly selecting which traces to sample, increasing the likelihood of losing relevant information or collecting insufficient information to reach relevant conclusions. The latter has different sampling rates that depend on certain conditions (e.g. if there was an error during execution or if a given operation has been slower than usual). Dynamic sampling is harder to implement and requires a bit more processing but is more likely to catch the information needed to troubleshoot the application. Balancing this trade-off is the responsibility of the team when adopting this pattern.

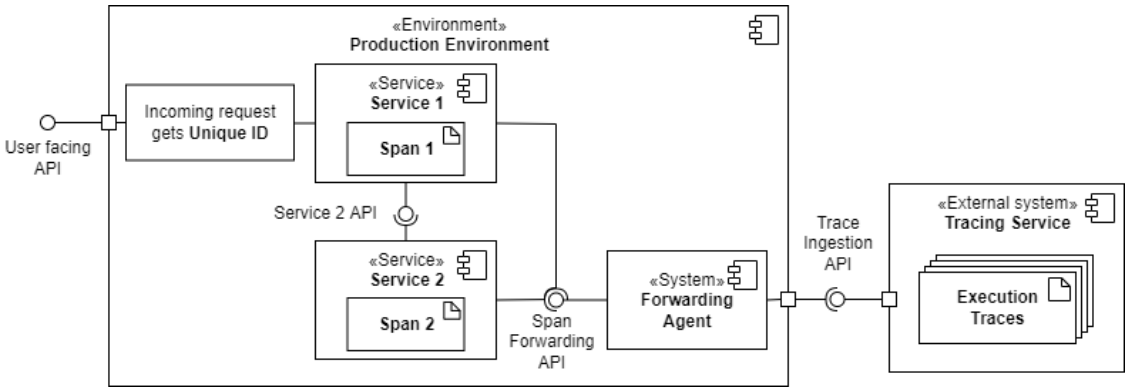


Figure M.1: Overview of the structure for the DISTRIBUTED TRACING pattern. The request is assigned a unique ID as it enters the system and the generated spans carry that ID.

Consequences

This pattern has the following advantages:

- Mean time to detect (MTTD) becomes shorter, making troubleshooting the system faster.
- User requests become observable from end to end, i.e. every operation performed since the request first arrives in the system until a response is sent to the user is tracked.
- Traces provide insight into individual operations, making it easier to identify system bottlenecks.
- Existing commercial solutions require little source code instrumentation and introduce very little overhead to the system.

This pattern also has the following drawbacks:

- Depending on the programming language used, implementing this pattern may require changes, sometimes complex, in the source code.
- Recording and aggregating traces introduces some overhead in the system and can impact its performance depending on the number of concurrent requests the application serves.
- Traces need to be stored, and as time goes by, they will start to occupy more and more space, demanding considerable infrastructure or increasing the storage costs.
- Existing commercial solutions are usually expensive; adopting an open-source solution and adapting it to the project may be cheaper but also introduce a larger overhead on the system.

Example

Meesho is an online shopping app for India. A blog post by Agarwal [3] explains that, as they transitioned from a monolithic architecture to a microservices-based framework, understanding the path of a single request became much more complicated. Because "a single request passes through multiple services" [3], tracing a request across all services became a challenge. The author explains in further detail their feed system's architecture. Most of their services composing the feed system were "Spring framework-based Java microservices" [3]. It followed a layered architecture, "where each layer has its responsibility and each domain service operates within its boundary context" [3]. The post had images to complement this description, but unfortunately, the post's images are no longer available.

To tackle the abovementioned issue, the team decided to implement distributed tracing in their system through Spring Cloud Sleuth. The author states that the decision was based on the tool's "auto-configuration capability and compatibility with other Spring libraries" [3]. The distributed tracing works as follows:

- The unique ID is assigned by the first service that does not find it in the request headers. They use two unique IDs for each request — the *Trace ID* and the *Span ID* — and the service attaches them to **ThreadLocal**, a Java class that provides thread-local variables.
- The team "implemented an interceptor that passes these IDs downstream as headers" [3].
- To include the unique IDs in the logs, the team used slf4j to fetch the IDs from the *Thread-Local* variables and include it in the logs.

They bumped into a problem with asynchronous processing that led to the loss of both unique IDs because the thread context changed in the middle of the execution. They wrote their code to copy the variables from one thread's context to the other to solve this issue. With that out of the way, Sleuth handles the aggregation and correlation of spans to construct the execution traces.

The author ends by stating, "we obtained Trace ID from User ID in one of our view layer services, then queried our centralised logging system. This showed us the whole request trace

across services and we instantly figured out that the A/B service was returning an unexpected response in one of our domain services which caused that issue" [3].

Known uses

Agarwal's [3] example described in the previous section is already a known use of this pattern that utilised Spring Cloud Sleuth to implement this pattern in a real-world microservices architecture.

FloQast is another company that uses distributed tracing to understand its application. In a blog post, Dinh [27] explains that FloQast is a fast-growing company, so they "add more and more business logic to the codebase every single day" [27]. The team noticed that bottlenecks started to appear in the system, but they did not have enough information to follow a data-driven decision. Thus, they started using AWS X-Ray service, which, as the author explains, "allows users to gain insight into requests that your application serves" [27]. Using that tool, they can "easily trace requests across AWS resources and other microservices" [27].

Finally, and on a much bigger scale, Netflix built its distributed tracing infrastructure to solve user issues better. In Netflix Technology Blog, Pandey [67] goes into some detail about how they did and evolved the infrastructure to cope with emerging standards like Open-Zipkin and OpenTracing. "Investigating a video streaming failure consists of inspecting all aspects of a member account" [67], so troubleshooting a user complaint can become very complicated on a system as complex as Netflix. They developed Edgar, an internal tool for troubleshooting streaming sessions. They started with a simple tool that, based on unique IDs for each streaming session, was able to "reconstruct session failure by providing service topology, retry and error tags, and latency measurements for all service calls" [67]. For their case in specific, the team had to apply a hybrid head-based sampling approach that enabled them "to record 100% traces in our mission-critical streaming microservices while collecting minimal traces from auxiliary systems like offline batch data processing" [67]. Even though this helped reduce the number of traces stored, the author states that they still had issues escalating their ElasticSearch clusters. So they eventually transitioned to Cassandra clusters to handle the high data ingestion rates.

Related patterns

This pattern somewhat encompasses the CORRELATION ID [13] pattern. In other words, to get DISTRIBUTED TRACING, the team must implement CORRELATION ID. Therefore, if that is not enough, the team can start with the latter to tackle the most basic correlation needs and move on to the former.

Logs are a common way to implement execution spans. In such cases, adopting LOG AGGREGATION becomes almost necessary to get DISTRIBUTED TRACING. In addition, logs can also be correlated and aggregated with the traces of execution. This provides the team with a complete report of the events and the time each operation took across the whole request lifetime.

However, generating too many traces can carry a very high overhead. The team should consider the LOG SAMPLING pattern to implement a sampling strategy that minimises the overhead of logged traces while maintaining the ones that truly matter for troubleshooting purposes.

Appendix N

DEPLOYMENT TRACKING design pattern's initial description

Deployment Tracking

Nowadays, code gets released faster and faster to meet customer demands, so system changes can be deployed many times a day. By introducing new features or tweaking existing ones, there is the risk that errors will start showing up and, even with a robust suite of tests, faults can make it into production, possibly degrading some quality attributes (e.g. performance, security). When metrics skew up or down, indicating system anomalies, the team should know where to start looking for a cause. Therefore, the team must track every deployment and change to the production environment, making it easy to relate effects observed in the system to the changes that caused them.

Also known as
Log deployments and changes [71], Release tracking [12]

Context
Nowadays, code gets released faster and faster to meet customer demands, so system changes can be deployed many times a day. Every change that gets released is an opportunity for failure. By introducing new features or tweaking existing ones, there is the risk that errors will start showing up and, even with a robust suite of tests, faults can make it into production, possibly degrading some quality attributes (e.g. performance, security).

By collecting metrics, engineers can notice when there is something wrong with the system. However, metrics represent effects and not root causes. In other words, they denote what failures or warnings are occurring in the system but do not correlate these to possible disruption causes.

Problem
When metrics skew up or down, indicating system anomalies, how the team know where to start looking for a cause?

Forces

- Issues in the system usually occur after changes, software-wise or not, hence the interest in tracking them.
- Changes affect end-users from both the technical (e.g. slow requests) and business (e.g. disappearing purchase history) point of view.
- Software release cycles can vary a lot, so any correlation strategy must be able to cope with just a few releases per year or many releases per day.

Solution

Track every deployment and change to the production environment [71], making it easy to relate effects observed in the system to the changes that caused them.

"Different companies [should] track change in ways that are reflective of their release cycle" [12]. So companies that release less frequently (e.g. once or twice a year) could make this a manual process, while companies that deploy more frequently (e.g. more than once per day) must automate this task into their **Deployment Tool** or pipeline. Nevertheless, automating deployment tracking handles both cases and eliminates possible human errors. Hence, it can be helpful for companies with infrequent releases as well.

By plotting deployments and changes together with existing metrics graphs, it becomes trivial to understand if a release caused the metrics' to skew up or down and, if so, at what time that happened. With this increase in visualisation, the team can detect the source of a failure and solve it much faster. Tracking Deployments can be achieved in many ways. **Deployment Records** can be logged or represented as events of the system and have an associated event handler to register them. However, to get the deployments shown in the metrics' dashboards, the best option is to directly insert the deployments as a binary metric into the **Metrics Service**.

Moreover, tracking deployments and changes and comparing them to business-related metrics can provide insight into how releases affect the end-users. Request rates or daily logins may increase if a major feature is introduced in a new system version. By plotting releases and business-oriented metrics together, managers can also benefit from the increased observability of the system.

Consequences

This pattern has the following advantages:

- Faulty versions of the system become easier and faster to detect, so the meantime to detect (MTTD) gets shorter.
- Releases can be correlated with other metrics, encouraging data-driven decision-making.
- When deployments are plotted with metrics, visualisation becomes richer, and a better system overview is made available to the stakeholders.

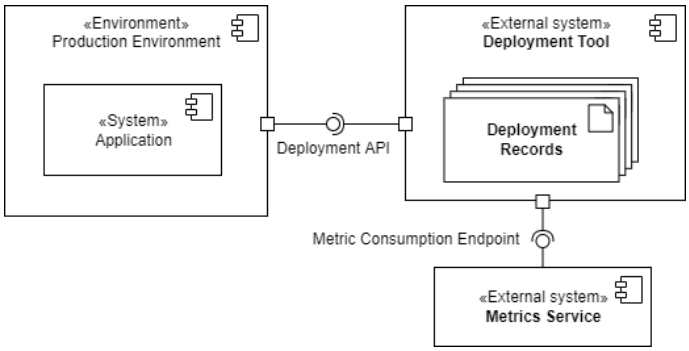


Figure N.1: Overview of the structure for the DEPLOYMENY TRACKING pattern. The production environment and application are shown just for better context.

- Teams can have more confidence when releasing a piece of software because, even if faults make it into production, they will be noticed quickly.

This pattern also has the following drawbacks:

- To take full advantage of this pattern's benefits, metrics should be already collected and visualised; implementing all of this requires considerable time and money.
- Plotting a binary metric, such as deployment logs, may not be supported by all tools; the team should ensure their **Metrics Service** supports drawing the releases as vertical lines.

Example

In an article written in Etsy's *Code as Craft* blog, Mike Brittain [12] explains how the company decided to adopt deployment tracking. Since they were already collecting many system metrics, they could correlate them and understand the effects of a change. Figure N.2 shows an example of an unusual increase in PHP warnings on the system.

However, they were missing the actual cause of the change - the deployments. Since issues in the system usually occur after changes, they started tracking deployments. To do so, they tweaked their code deployment tool to emit a deployment event every time a new release was made. The event consisted simply of a name field (*events.deploy.website*), a placeholder value (*1*, for example) and the timestamp (in their case, *1287106599* in Unix time). By sending this metric to their monitoring tool, Graphite, they plot the changes together with the metrics, as depicted in Figure N.3.

Given the new information, it became clear that the warnings were due to a release at 16h, and they were fixed with the two subsequent releases. The ability to visualise the changes makes it very easy to identify release trends. This example was focused on a technical metric. However, Brittain also provides an example of how the number of new posts in Etsy's forums increased after a product launch (see Figure N.4), revealing how tracking releases can also be helpful for business managers and customer support.

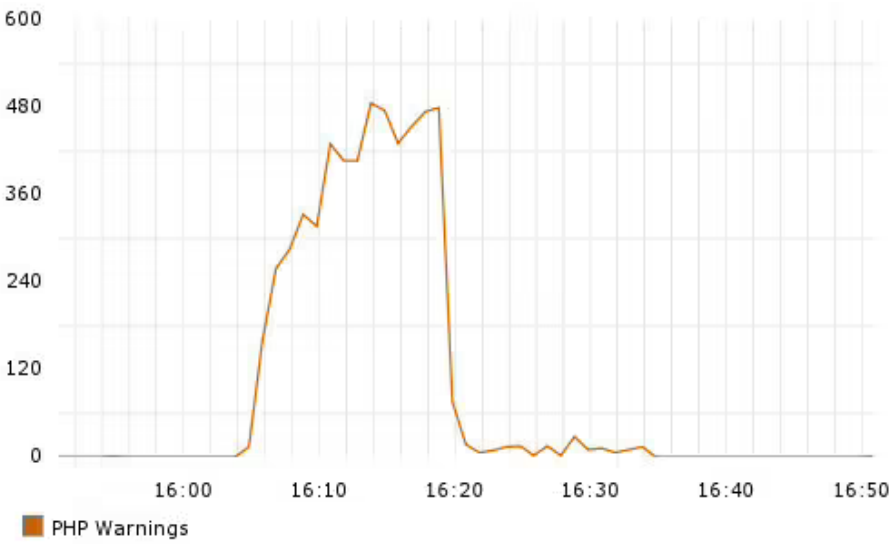


Figure N.2: Plot of the number of PHP warnings over time for Etsy's system.

Figure N.3: Plot of the number of PHP warnings over time and code deployments for Etsy's system.

Known uses

Brittain's [12] example described in the previous section is already a known use of this pattern that utilised Graphite to implement this pattern in a real-world system.

In a blog post, Will Sewell, a backend engineer at Monzo, explains how they deploy to production more than 100 times a day [80]. The author goes into detail to explain many things that contribute to the high frequency of deployments, including company culture and used tools. We focused on the fact that they track the number of deployments they do each day, which allows the company to plot, for example, the average weekly deployments per engineer. Moreover, their "deployment pipeline records events of what happened, and when " [80], so they can "easily find

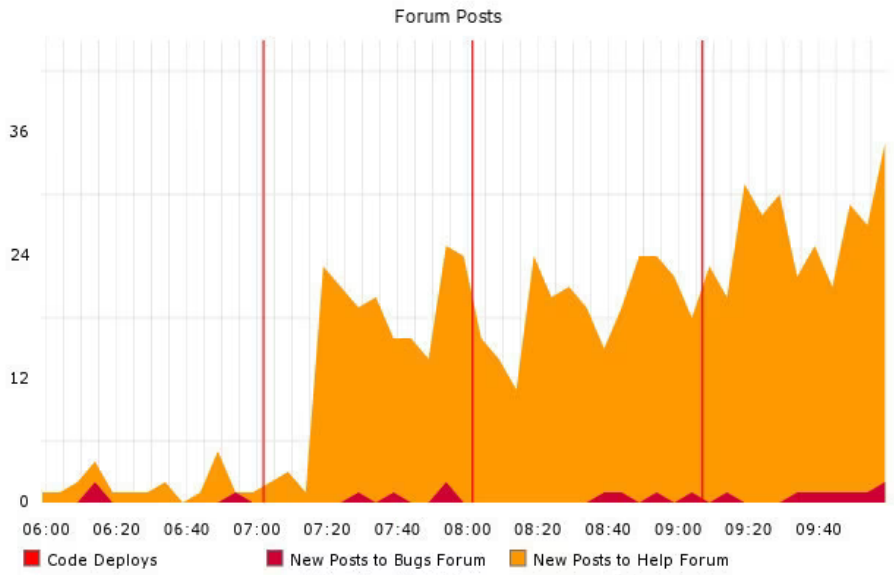


Figure N.4: Plot of the number of posts in Etsy's forums and code deployments.

out why things went wrong and stop it happening again" [80].

Finally, in a blog post about Raygun's Deployments feature, Freyja [37] reports that they interviewed some customers to understand how they use the feature to track deployments and version releases. According to the author, most teams find it "most helpful when investigating which errors have been introduced with the new release, which are still occurring, and which have been fixed " [37]. The tool provides visibility over the deployment pipeline so teams can avoid error propagation across the deployments. The author recommends configuring "Raygun with deployments so that you can correlate error spikes with releases " [37], which greatly resembles this pattern's solution.

Related patterns

Deployments and changes can be treated as events and monitored that way, so this pattern follows an event-based monitoring strategy. Similarly, deployments can be treated as binary metrics that are collected and correlated with other system metrics. Without metrics that illustrate system anomalies, the usefulness of DEPLOYMENT TRACKING is much lower. Hence, the team should consider adopting INFRASTRUCTURE METRICS, APPLICATION METRICS, or both before implementing this pattern.

Nevertheless, if the visualisation or collection of metrics is not yet developed in the project, merging the deployment logs with other system logs by adopting LOG AGGREGATION [83] may already be enough to correlate deployments with system anomalies. Logs are the simplest way to implement this pattern and may provide enough value for teams that do not have time or money to invest in more robust monitoring solutions.

Appendix O

SYNTHETIC TESTING design pattern's initial description

Synthetic Testing

Tests are usually run in a controlled environment before the application is deployed and made available to end-users. However, once it is deployed, many factors can influence the application's behaviour. The team should create or pick a subset of existing test cases and periodically run them against the production environment, ensuring the application behaves as expected and detecting issues before they affect end-users.

Also known as
Synthetic Monitoring [24], Proactive Monitoring [24], Synthetic Transactions [64]

Context
Following CI/CD practices, most teams run tests in multiple stages of their deployment pipeline [62]. These tests are usually run in a controlled environment (e.g. a testing or staging environment) before the application is deployed and made available to end-users [61]. This way, teams validate that the product about to be released is behaving as expected.

However, once an application gets deployed, many factors can influence its behaviour. Some are internal (e.g. a code fault, little disk space), and others are external (e.g. connectivity issues, network configurations). When so many things can go wrong, the system becomes more fragile.

Moreover, the more distributed the application gets, the harder it is to replicate the production environment. Components are usually released multiple times a day, creating further instability [61]. "This rapid rate of change to the production environment tends to make testing during CI/CD stages not hermetic and actually not representative of the end-user experience and how the production system actually behaves" [61].

Problem
How can the team ensure that the application running in production behaves as expected?

Forces

- Automated tests are usually run in a controlled environment, which is not representative of the production environment.
- Traditional monitoring techniques (e.g. metrics, logs and traces) are only available when users interact with the system.
- Health endpoints are helpful to understanding if a service is healthy but do not provide information on how it is behaving.
- Detecting failures only after user complaints are issued should be avoided because it degrades the user experience.

Solution

Create or pick a subset of existing test cases and periodically run them against the production environment, ensuring the application behaves as expected and detecting issues before they affect end-users.

Synthetic tests are essentially a form of testing. Therefore, when adopting this pattern, the team should start by defining the **Test Cases** they want to execute. These can be created specifically for testing the **Production System** or be the same ones used in an existing CI/CD pipeline. There are two main kinds of synthetic tests: "browser tests, which check whether users can complete important transactions such as account sign-up and checkout, and API tests, which allow [the team] to monitor key endpoints at every network layer" [24]. These kinds are comparable to system-level and integration tests, thus having similar advantages and limitations. Moreover, they can be applied to many use cases and be used to check actual behaviour and the performance and availability of the application.

So far, this solution is closely related to already established testing practices. This pattern is set apart because synthetic tests must be run against the **Production System**. "Testing in production [is] arguably a form of "monitoring" activity" [64] because the team monitors the behaviour of our application after it is deployed. They should set up a **Synthetic Testing Platform** to periodically execute the test cases against the application. The better the testing platform simulates a real user, the more accurate the test results will be. For the best possible coverage, the team should consider running their tests from different parts of the globe - at least where most of the application's users are located - and simulating different user environments (e.g. browser, OS and device). Obviously, the more the team tests, the more costly it becomes to maintain and execute the test suite. It is a trade-off that should be considered and balanced according to the services' monitoring needs.

The team must consider that they are effectively running tests in production. It is not as daunting as it may sound but still carries some risks. Microsoft's wiki reports the following potential issues with synthetic tests [61]:

- Corrupted or invalid data - Tests use invalid or corrupt test data. A schema may help enforce a proper structure on the data.

- Protected data leakage - Tests accidentally expose sensitive data.
- Overloaded systems - Tests crash or overload the system.
- Side effects - Tests trigger unintended side effects.
- Skewed analytics (traffic funnels, A/B test results, etc.)
- Auth/AuthZ - Tests are required to run in production where access to tokens and secrets may be restricted or more challenging to retrieve.

Finally, the **Test Results** can feed a **Metrics Service** that records the history of executions. The team can keep track of test evolution and correlate test failures with other system metrics. Detecting a failure and finding its cause are two distinct tasks. This pattern only helps with the former. In other words, a test failure will not provide much detail into the actual problem with the service. To find the problem, the team should have other artefacts (e.g. logs, traces and metrics) that they can use to troubleshoot the application.

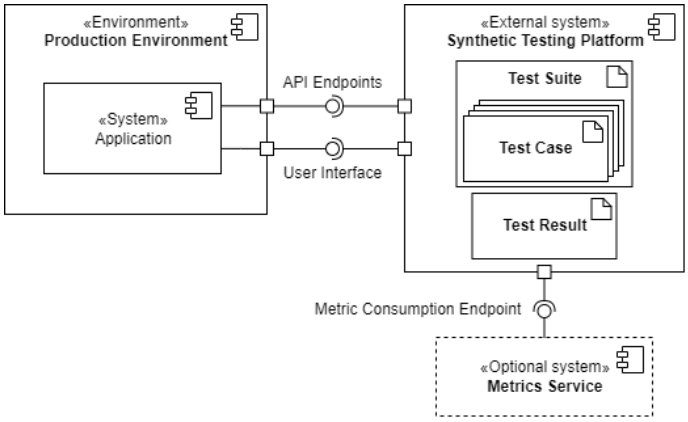


Figure O.1: Overview of the structure for the SYNTHETIC TESTING pattern. The dashed component is optional, while the remaining are strictly necessary to adopt the pattern.

Consequences

This pattern has the following advantages:

- When a feature is not working as expected, the team can detect the issue before it impacts end users [24], effectively reducing the mean time to recovery (MTTR).
- The team can check whether performance requirements are met.
- It checks the actual system behaviour in production, complementing other monitoring practices more focused on the system's state.
- Synthetic tests can be run anytime, providing a way to monitor the system even when there is no user activity.

- It increases the team's confidence to deploy because any fault that makes it into production can still be detected before impacting the users.
- It provides another metric to monitor to help identify regressions and correlate them to specific events (e.g. deployments, infrastructure anomalies).

This pattern also has the following drawbacks:

- When a test fails, it usually only provides information regarding the expected and actual results, lacking context to identify the cause of the failure [64, 24].
- The testing system needs to be updated every time the application changes, even if there are only slight changes (e.g. renaming a UI component) [24].
- The test cases are a limited representation of what a real user can do, so they might still miss complex user interaction scenarios [21].
- Synthetic tests can be hard to set up, especially when considering real and complex scenarios involving data changes.

Example

Newman [64] provides a great example of this pattern's adoption in his book *Building Microservices*. He explains that, some time ago, he was part of a team working on a system for an investment bank. They were responsible for reacting to trading market changes and evaluating the impact on the bank's portfolio. Market changes were constantly arriving in the system through events that it had to finish processing in under 10 seconds to meet the bank's requirements. "The system itself consisted of around five discrete services, at least one of which was running on a computing grid that, among other things, was scavenging unused CPU cycles on around 250 desktop hosts in the bank's disaster recovery centre" [64].

Due to the system's complexity, Newman reports that the low-level metrics the team was gathering were filled with noise. Moreover, they "didn't have the benefit of scaling gradually or having the system run for a few months to understand what «good» looked like in terms of low-level metrics like CPU rate or response time" [64].

Therefore, the team used synthetic tests to simulate events that would only affect a set of testing objects. They ensured the tests would not have any unintended side effects. Newman and his colleagues used a tool called Nagios to periodically run a command-line job that would generate a fake event and insert it into a queue. Their "system picked it up and ran all the various calculations just like any other job, except the results appeared in the «junk» book, which was used only for testing. If a repricing wasn't seen within a given time, Nagios reported this as an issue" [64].

Through synthetic tests, the team could guarantee they were meeting the system's performance requirements and that it was behaving as expected.

Known uses

Newman's [64] example described in the previous section is already a known use of this pattern that utilised Nagios to solve a real-world problem.

In a customer story, Splunk reports how Blue Apron improved its website load time by 30% [90]. Tom Wilson, a principal engineer at Blue Apron, states that "from testing new features to identifying easy performance wins, Splunk helps [them] integrate performance testing into our development life cycle" [90]. The company solved issues with its homepage through synthetic tests, reducing its load time by 30% and meeting its performance requirements.

Finally, a post by Carlbark on Meltwater's blog [18] explains how Meltwater's API was synthetically tested. The system's production environment was so complex that including everything in pipeline tests was slow (or sometimes unfeasible) to achieve. Therefore, Meltwater's team developed an in-house tool to carry test cases against the production environment. As a result, they found faults in their components and 3rd party dependencies, which allowed the team to fix many issues before going into the product's beta stage [18].

Related patterns

Ideally, the synthetic testing platform is running externally from the production environment. In other words, the team can adopt EXTERNAL MONITORING [83] alongside this pattern. Otherwise, the tests may not be affected by factors that influence the users, such as network latency, firewalls and localisation.

The test results are themselves something that should be monitored. Therefore, the team can adopt APPLICATION METRICS [72] and register the test results and metadata (e.g. execution time, service name) as system metrics. This way, alerts can be set when tests fail more than a specified number of times per time frame.

Finally, the LIVENESS HEALTH ENDPOINT pattern requires another service to perform the health checks periodically. Thus, that pattern can be implemented through SYNTHETIC TESTING. Each health check becomes a synthetic test whose expected outcome is just a response with a success code. If so, the team will only need to set up and maintain a single infrastructure, reducing the costs of adopting both patterns.

Appendix P

LOG SAMPLING design pattern's initial description

Log Sampling

The more services and servers a system has, the more monitoring data it will generate. All this data must be stored and processed to be useful to the team. Logs are part of this data, and the number of logs generated will also increase rapidly as the system grows. The burden to generate, store and process the logs is much bigger in complex systems. The team may need a way to reduce the number of logs generated by the system while maintaining enough information to troubleshoot the application effectively when an issue occurs. Therefore, they should sample and prioritise logs to reduce the number of logs that need to be stored and processed.

Context

Logging is one of the oldest monitoring practices due to its simplicity and broad support. Generating a log is very easy and can be as simple as a piece of text [91]. In addition, "most languages, application frameworks, and libraries come with support for logging" [91]. Therefore, it is safe to assume that almost every service in a system is logging something, one way or another. Logs are indeed quite helpful when troubleshooting an error. The description of the events in the system provides insightful information into its behaviour that is crucial to finding the root cause of failures.

The more services and servers a system has, the more monitoring data it will generate. All this data must be stored and processed to be useful to the team. Logs are part of this data, and the number of logs generated will also increase rapidly as the system grows. Logging excessively can already harm the system's performance [91]. The burden to generate, store and process the logs is much more significant in complex systems. Thus, logging too much introduces an enormous overhead in these systems. On the other hand, too few logs will hinder troubleshooting processes and affect the system's quality. The team must balance the number of logs generated to manage this trade-off.

Problem

How can the team reduce the amount of logs generated by the system while still maintaining enough information to effectively troubleshoot the application when an issue occurs?

Forces

- Collecting every log in a complex system is not feasible due to infrastructure constraints.
- Logs have different severity levels that can be configured in the logger service.
- Collecting an error log without the remaining execution logs usually does not provide enough information for troubleshooting.

Solution

Sample and prioritise logs to reduce the number of logs that need to be stored and processed while maintaining enough information for effective troubleshooting.

"*Sampling* is the technique of picking a small subset of the total population of event logs generated to be processed and stored" [91]. Sampling can be implemented mainly in two ways:

1. Fixed probability sampling — logs are recorded based on fixed probability value [96]; this is easier to implement but does not handle incoming traffic spikes.
2. Adaptive sampling — "dynamically adjusts the probability to ensure that the number of logs produced in a period of time keeps stable" [96]; this is slightly harder to implement but maintains a usual number of logs even when the application is under heavy traffic.

These two methods can be used interchangeably, i.e., the team can use fixed probability for some services and adaptive sampling for others without any problem. Moreover, consider that a **Logger** generates the logs. Afterwards, they are saved on a **Data Store** and are consumed by a **Log Consumer** (e.g. an aggregation service, an indexation service, a human being). In such a scenario, sampling can be classified into two categories: head-based and tail-based sampling. The first implies that sampling is done by the **Logger** as the logs are generated. This cuts on every aspect of the logging overhead but will degrade the correlation between logs (see CORRELATION ID [14]). Tail-based sampling implies that logs are sampled during a preprocessing phase, usually after being aggregated and correlated. This means there is still some overhead with generating and processing most of the logs. However, the sampling process can take into account log correlation and other criteria that were not available when the log was generated. The **Log Preprocessor** service should be the one performing tail-based sampling to prevent data propagation among many services. Note that head-based and tail-based sampling can be used together to create a robust sampling strategy.

Finally, sampling can be adjusted based on the log's criteria (e.g. severity level, origin). Sridharan argues that "the efficacy of the sampled dataset is contingent on the chosen keys or features of the dataset based on which the sampling decision is made" [91]. In other words, if the heuristic

used to sample the logs is poor, then the resulting log set will also be poor. The team should consider which aspects of the system's logs are more relevant and prioritise the logs based on them. For example, in a tail-based sampling scenario where the logs are correlated with a unique ID, a set of logs for a single request that contains at least one error-level log may bypass sampling altogether. Other possible criteria are the log source, the elapsed time, or custom attributes specific to a service. The team's goal when adopting this pattern is to ensure that the system collects the most important logs and discards the less helpful ones.

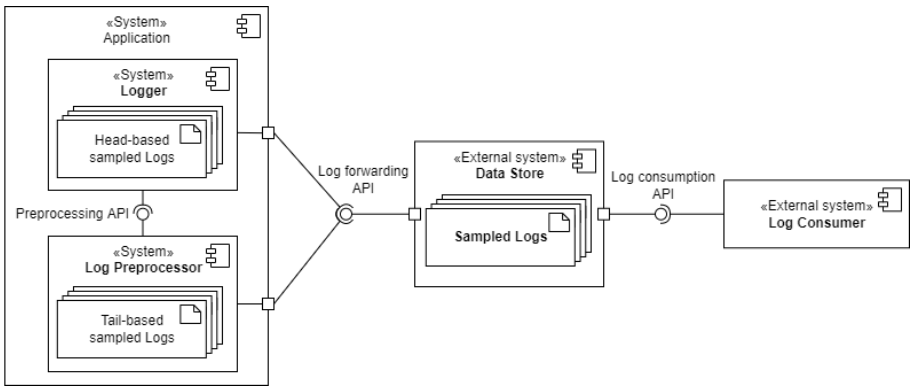


Figure P.1: Overview of the structure for the LOG SAMPLING pattern. Having both the logger and log preprocessor is not mandatory. The team may use just the one that provides more value to them.

Consequences

This pattern has the following advantages:

- Log overhead is reduced, so generating, storing and processing logs becomes cheaper.
- When using adaptive sampling, the number of logs generated per time unit is maintained somewhat constant.
- Logging remains feasible in complex systems, so the teams do not need to cut on logging practices as the system grows.

This pattern also has the following drawbacks:

- Important logs may be lost during the sampling process, which negatively impacts the team's troubleshooting capabilities.
- Trace analysis usually relies on logs, so sampling them may affect the team's DISTRIBUTED TRACING efforts.
- Implementing adaptive sampling and choosing the appropriate log criteria for sampling may require considerable effort.

Example

A common usage of log sampling is to control the overhead of DISTRIBUTED TRACING. Since spans are usually generated and saved as logs, sampling the spans mean sampling the logs they are stored in to avoid cluttering the system.

Imagine a system with 1000 microservices and around 1000 requests per second. An incoming request can easily trigger operations in around 100 of those services. In order to trace such a request, there must be 200 generated logs, one for when the request enters a service and another when it leaves. Suppose that the generated logs follow OpenTelemetry's format, like the one in the listing below.

```
{
  "Timestamp": "1586960586000000000",
  "Attributes": {
    "http.status_code": 500,
    "http.url": "http://example.com",
    "my.custom.application.tag": "hello",
  },
  "Resource": {
    "service.name": "donut_shop",
    "service.version": "2.0.0",
    "k8s.pod.uid": "1138528c-c36e-11e9-a1a7-42010a800198",
  },
  "TraceId": "f4dbb3edd765f620",
  "SpanId": "43222c2d51a7abe3",
  "SeverityText": "INFO",
  "SeverityNumber": 9,
  "Body": "20200415T072306-0700 INFO I like donuts"
}
```

Example of a trace log that follows the OpenTelemetry standard.

A *.txt* file with the example's content occupies around 0.5 kB. So, for every request, we need to store approximately 100 kB. With 1000 requests per second, the number grows to 100 MB. After one day, the system would have generated 8640 GB of logs. The data provided in this example may sound slightly far-fetched. However, considering that Netflix's API handles around 20000 requests per second [75] and that the three largest companies from the study in [96] had more than 1000 microservices, it is not as exaggerated as it may have looked initially.

The main takeaway is that logging storage becomes an issue very quickly. If we adopt an adaptive sampling strategy by configuring our log library to ensure a sampling rate of around five logged requests per second, at the end of the day, we only have to store 43.2 GB of logs. By adjusting the sampling rate, we can configure how many bytes of data we generate per day. To conclude, note that this is just an issue for complex and high traffic systems, like the one we portrayed in this example. The ideal scenario is to log everything; if the team can afford to do so, they should not use LOG SAMPLING.

Known uses

Waseem et al. conducted an industrial survey with ten different microservices systems from ten different companies [96]. According to their results, four companies used sampling as part of their tracing system. All four companies used head-based sampling and, depending on the service, either fixed probability or adaptive sampling. These companies had the most complex systems in the study, each with thousands of microservices to maintain. The authors concluded that "sampling can alleviate the overhead to services and log storage while losing important traces for trace analysis tasks such as root cause analysis " [96]. Nonetheless, in very complex systems, sampling remains a helpful method to control log overhead.

Sigelman et al. [82] wrote a white paper explaining the design of Dapper, "Google's production distributed systems tracing infrastructure " [82]. The tool was developed internally to tackle Google's production challenges. As one can imagine, problems at Google's scale demand unique solutions to handle the enormous amount of data generated. One of Dapper's design goals was low overhead. The authors concluded that sampling was the best way to achieve that goal [82]. The authors started by using a fixed probability strategy to collect approximately one sampled trace in 1024 candidates. This worked well for high-throughput services, but the number of traces collected per time unit was low for less loaded services. Therefore, the team implemented adaptive sampling in Dapper to ensure that the number of traces sampled per time unit was nearly constant. This tool reveals how large and complex systems like Google's eventually need to sample their data.

Finally, according to a post in Uber's engineering blog [81], the company explains how Jaeger, an open-source tool it created, supports sampling. According to Shkuro, the post's author, "most production tracing systems, especially those that have to deal with the scale of Uber, do not profile every single trace or record it in storage " [81]. Thus, they were forced to sample the trace data to be able to cope with the storage space required. Moreover, Uber created a feature for Jaeger, a polling feature in the tool's client libraries that allows the to poll for the sampling strategy to use. This allows the system's "backend to dynamically adjust the sampling rates as the traffic patterns change " [81].

Related patterns

This pattern and PREEMPTIVE LOGGING [83] have a slightly different goal, but both strive for having the maximum information possible within the system's resource capacity. Therefore, PREEMPTIVE LOGGING can be viewed as an alternative to the current pattern for less complex systems. However, for more complex systems, LOG SAMPLING can be used to handle any excessive logging that has not been tweaked after the adoption of PREEMPTIVE LOGGING.

If the team is adopting CORRELATION ID [13], they should consider using tail-based sampling to avoid losing the correlation between log entries. Moreover, if they already use LOG AGGREGATION [83], then the aggregation service can take the role of **Log Preprocessor** and be responsible for sampling the aggregated logs.