Universidad Politécnica de Madrid

Escuela Técnica Superior de ingenieros informáticos

Máster Universitario en Innovación digital: Ciencia de Datos
Master's Programme in Digital Innovation: Data Science

Trabajo Fin de Máster
Master's Thesis

Monitorización de Gateways en Arquitecturas de Microservicios Orientadas por Dominios utilizando técnicas de Big Data

Domain-Oriented Microservices Gateway Monitoring Using Big Data Techniques

Author:	**Cristian M. Abrante Dorta**
Supervisor (UPM):	**Marta Patiño**
Supervisor (Aalto University):	**Hong-Linh Truong**
Company advisor:	**Teemu Sidoroff**

Madrid, Marzo / March 2022

Abstract

Over the last decade, many companies and organizations have adopted microservices as the software architecture pattern to organize their software system. When the number of those microservices increases dramatically, there are some problems associated with tracing errors and managing common responsibilities. In that sense, many companies adopted a Domain-Oriented Microservices architecture pattern, where the microservices of a different business domain are separated and can be accessed only through a single gateway.The errors that are handled at the gateway level are interesting for other teams depending on it, but maybe they do not have direct visibility on them. This thesis has two outcomes; on the one hand, it introduces an extensible format for defining the high-level responsibilities that the gateway has to handle and the associated errors with those. On the other hand, it also describes the data pipeline introduced for gathering those errors at the gateway level and storing them efficiently to be visualized conveniently afterward.This tool is tested in the context of a multinational company on a gateway that handles thousands of requests per second. Apart from that, a comparison has been established with the most common open-source gateway technologies, showing that they do not provide by default enough cross-team visibility on the events that we are trying to gather.

Acknowledgement

Joining a master's program is challenging but rewarding work. In these two years, I have learned a lot, not only from an academic point of view but also about personal development, and this thesis is the culmination of all this effort.

First of all, I would like to thank Unity Technologies Finland for giving me the opportunity of joining such an amazing company, especially the Advertiser Public APIs team for making me part of the team from day one.

I also would like to thank my company advisor Teemu Sidoroff for helping me with this project unselfishly. In that regard, I would like also to thank my supervisor, Prof. Hong-Linh Truong, for his valuable feedback and continuous support.

I also think it is important to mention my utmost gratitude to the public institutions that participate in this Master's degree: EIT Digital, Aalto University, and the Technical University of Madrid. The existence of these institutions makes it possible for people with not so privileged backgrounds to strive for better opportunities.

Also, I would like to express my gratitude to all my friends here in Finland, without them life on campus won't be the same. In especial to Unna and to my flatmate and friend Stefano, for all the good moments lived here.

There is also a special place for my lifelong friends, the ones that had helped me to become the person that I am today: Rafa, Alba, Ángel, Alejandro Grillo, Reina, Alejandro Pérez, and Jorge. *¡Muchas gracias!*

Last but not least, I would like to thank my family for their love and support during all this time and for teaching me how to be the best version of myself. *¡Les quiero!*

Otaniemi, February 28, 2022

Cristian M. Abrante Dorta

Contents

Abbreviations

SOA	Service-oriented architecture
DOMA	Domain-oriented microservices architecture
WSDL	Web Service Deskcription Language
SOAP	Simple Object Acess Protocol
RPC	Remote Procedure Call
REP	Request Execution Path
RFC	Request for comments
BQ	BigQuery
VCS	Version Control System
JSON	JavaScript Object Notation
UI	User Interface
PR	Pull Request
GCP	Google Cloud Plattform
CI	Continuous Integration
CI/CD	Continuous Integration and Continuous Deployment
gRPC	Google Remote Procedure Call

1 Introduction

Over the past few years, there has been debate on how to properly organize and deploy service-oriented architectures. One of the software architecture patterns that has gained the most popularity today is microservices [13]. Many large, medium and even small software companies have adopted this architecture because of its enormous benefits compared to traditional, monolithic software applications.

Microservices are an extension of service-oriented architecture. The main objective is that each of these microservices provides a well-defined and clear scope of functionalities, which can be accessed by another microservice using *remote procedure calls* [51]. By using this pattern it can be established clear ownership of each microservice by different teams where they are responsible for the development, maintenance, and deployment. This is a clear advantage over the classic monolithic architecture, since in these, although the development tasks may be divided among different teams, the deployment of a single functionality involves the deployment of the entire system, which is then prone to cause major problems that can lead to the complete shutdown of the application.

Although the microservices architecture represents a step forward in the design of software systems, it also has some drawbacks compared to monolithic architectures, such as performance. In a traditional monolithic application subroutine calls are something that can be solved within the same process and machine. However, in microservices, the call has to be made over the network, which is something that may eventually introduce some execution delays. Apart from this, there is another problem faced by microservices architectures, which is the orchestration of thousands of different microservices, when dealing with enterprises operating on a large scale. In this case, any problem caused by one microservice can spread across many of them, and engineers have to debug calls coming from a huge stack of different services, which is something that can be challenging and time-consuming.

In order to provide an scope that would simplify the maintenance of the microservices set, many companies had adopted a **Domain Oriented Microservices Architecture** pattern, introduced by Gluck A. [11] in behalf on the Uber engineering team (Section 3.1.3). Using this architecture, many microservices can be grouped within a logical division, called Domain, that fits the needs of the enterprise. To isolate the domains and make them easier to maintain and debug, one approach which can be applied is to create a common gateway that exposes the domain's interface (usually as routes that can be called with different HTTP methods). Apart from being the single point of entry for specific domains, the gateway can also provide different high-level responsibilities, such as authentication, routing, rate limiting, load balancing, etc. This pattern has been widely adopted by companies operating on a large scale, such as Uber [2] or Unity Technologies (Figure 1).

The gateway is in itself another microservice. Consequently the error monitoring of it is a crucial task for maintenance the overall health of the system. About microservices logging and monitoring there is already a well established action framework [14] [67], which can be summarized as the storage of a log entry in a centralized server (for example Google Cloud or Amazon Web Services logging tools) for each request or error that it is produced in the system. Usually that is enough for tracing back each request that the system is receiving.

Even though there is a well-established framework for monitoring the requests, due to the

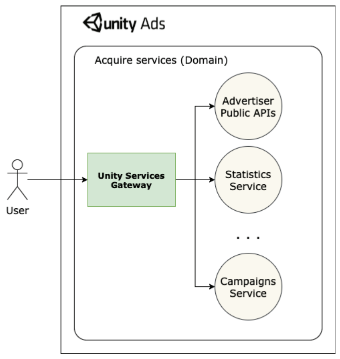


Figure 1: Domain-oriented microservices architecture structure in Unity Technologies

central position of the gateway on that type of architecture there are some problems on how to monitor and communicate the business logic of it (Section 2.2). In particular, this thesis address the case of the **Unity Services Gateway**, on the company Unity Technologies (Section 2.1.2). On that particular case, there are some questions raised about the high-level responsibilities of the gateway by project managers or development teams, such as: *what users or organizations are hitting our rate limits? What is the amount and frequency of authentication errors our system is having?*.

The answers to those questions are relevant because some actions can be taking against malicious users which are doing too many queries to the system or directly some organizations can be contacted to check if their integration with the APIs that the company provides are working properly. Nevertheless, the current monitoring setup of the company (Section 2.1.3), is not enough for answering those questions. This is mainly due to the fact that they only store the minimal piece of information required to trace back the request but do not contain complex fields, such as organization ids, paths for the request, rate limit quotas exceeded, etc.

One might think that simply by extending the logs entry to contain more information is enough for solving this problem; nevertheless this simple approach is not effective due to some reasons. The first one is that the data needed for solving those questions has to be extracted from different sources, which is something that it is reasonable to do when dealing with errors but can not be generalized to all logs entries as would dramatically increase the processing time of the requests in a system that is receiving thousands of request per seconds. The second reason is that the logging tool is shared among many microservices, so none of them has to worry in how to communicate with the cloud provider, this is why it is better to keep simple the amount of fields that the logs are storing. The final reason is the single responsibility principle, as it seems sensible to separate the data store that would keep complex error fields that would solve high-level questions from the actual log registry that would keep track of the requests. Taking this into account, the objective of this thesis can be defined as determine the best way to monitor the errors produced by the high-level responsibilities of a Domain Oriented Microservice Architecture gateway.

1.1 Contributions

This thesis proposes a monitoring framework for a Domain-Oriented Microservice Architecture gateway that allows stakeholders to observe the errors that the gateway is producing when performing its high-level operations. Particularly, the introduced framework offers the following characteristics:

- Definition of an extensible error format that matches the high-level responsibilities that the DOMA gateway has to fulfill for every endpoint it exposes.
- Creation of a cloud-agnostic data pipeline that transmits the errors from the emission on the gateway to the data storage.
- Creation of a visualization dashboard that will retrieve the error data and visualize it in a convenient structure.

A prototype is created for the particular case of the Unity Services Gateway, the DOMA gateway for Unity technologies, a highly-available service that handles thousands of requests per second, and that it is managed by a growing team of engineers located in different parts of the world. Additionally an evaluation step had been performed, first by doing unit testing on the implemented functions and then by creating an script that ensures that the data is flowing correctly through the pipeline.

1.2 Structure of the thesis

This thesis is divided into seven sections. The first one is the introduction, where the proposed contributions are highlighted. Section 2 is the background, where there is an overview of the company's architecture and a definition of the problem that will be tackled. Section 4 covers requirements engineering, which defines and establishes the project's scope according to its stakeholders. Subsequently, Section 3 will give a literature review, introducing the different software architectures used nowadays, primarily focusing on DOMA. Then, Section 5 will explain the technical solution and all the created artifacts for the proposed framework. Also, Section 6 will explain the followed evaluation process, especially compared with other industry-standard solutions. Finally, in Section 7, some conclusions to this work will be offered in addition to future work that can be done in this regard.

2 Background

This section describes all the necessary background to understand the problem that this thesis seeks to address. On the first section there is an introduction of the current software architecture where this thesis is developed, and in the second subsection, the problem is framed from different points of view, taking into account the various stakeholders involved.

2.1 Current sofware architecture description

To better understand the project, it is essential to present the company where it has been developed. This thesis is done in collaboration with Unity Technologies [21], a multinational company based in the United States. The company's main product is the Unity game engine [37], an industry standard for game and 3D content development [52].

Although the game engine is the most famous product offered by the company, it is not the only source of revenue, and it is not enough to understand its business model. This is mainly because the company's value proposition can be summarized as: *"Create a platform for creators to develop, execute and monetize their content"* [25]. To realize this value proposition, the company has established two organizational departments that offer various products to its users.

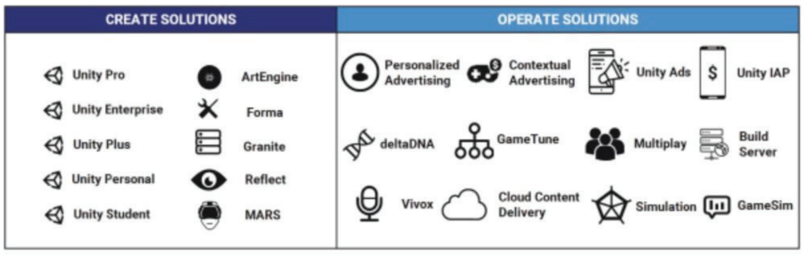


Figure 2: Products offered in the two organizational divisions of Unity Technologies

- **Create solutions**: The main functionality of this product suite is to offer a set of tools that allows users to produce all kinds of 3D content. The business model of this division consists of a subscription-based model for the different versions of the Unity Editor.
- **Operate solutions**: The goal of the products categorized in this section is to enable creators to run a successful business from their 3D content. This is the organizational section that contains UnityAds, the product on which this thesis project was developed.

2.1.1 UnityAds

To understand UnityAds correctly, first, it is essential to clarify the concept of **online advertising network** [33] [59]. An advertising network is a type of business or service whose purpose is to connect advertisers, who are individuals who want to display advertisements for the products they intend to sell, with publishers, who are individuals who can provide space to place such advertisements, and who usually reach a certain level of audience. The original definition of advertisement network was really tied to newspapers, magazines, and television, but in recent years, it has been extended to online services [36]. In that sense, UnityAds is an ad-hoc advertisement network for the game industry. In a nutshell, UnityAds allows game developers to show ads about their games in other games and place advertisements of other games in theirs [24]. As can be seen, by using UnityAds, the same game developer can be publisher and advertiser simultaneously, melting both roles.

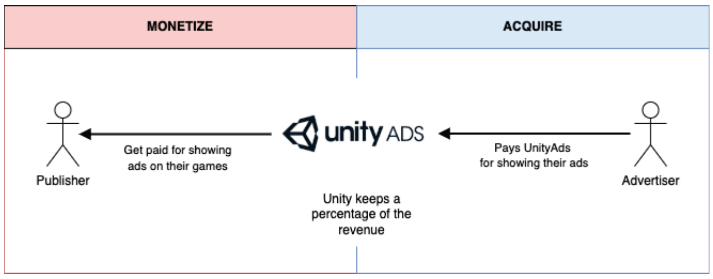

MONETIZE
ACQUIRE
Publisher
Get paid for showing ads on their games
unity ADS
Pays UnityAds for showing their ads
Advertiser
Unity keeps a percentage of the revenue

Figure 3: Simplified structure of UnityAds

Figure 3 shows the two aspects that make up the advertising network: **Acquire** and **Monetize**. The terms correspond to the classic definition of publisher and advertiser but translated into the gaming industry terminology.

- **Monetize unit**: This section includes all the tools provided to game developers to monetize their games, in addition to all the technical solutions needed to deliver an ad to the mobile game that has a publisher space.
- **Acquire unit**: This set of tools includes all the products that help advertisers acquire new users who can install the games they advertise on the network.

Although this overall context of UnityAds was essential to understand the big picture, only the Acquire unit is considered in the following sections, as the thesis focuses on it.

2.1.2 Acquire architecture overview

This section covers the architecture overview for the acquire unit of UnityAds. It is important to note that many details of the architecture will not be explained in-depth to respect the company's privacy.

The architecture of UnityAds was initially developed by a Finnish startup, Applifier, acquired by Unity Technologies in 2014 [66], and it can be seen in Figure 4.

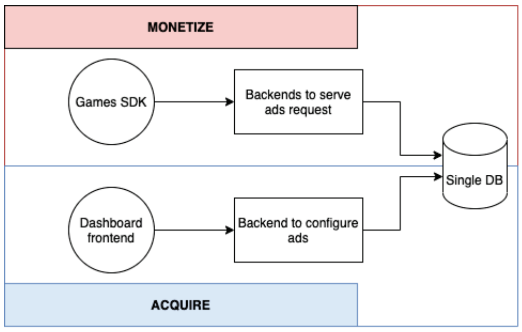


Figure 4: Diagram for the initial architecture of Applifier

This almost *monolithic architecture* [27] was convenient for a small engineering team to introduce changes to it, which allowed them to achieve a solid customer base prior to the Unity acquisition [72]. But when scalability started to become a fundamental requirement, there were some pain points they had to address. On the one hand, the database was the junction point between the services and the mechanism they used to communicate, which was limiting and error-prone, and on the other hand, this architecture was not suitable to be managed by a growing team of engineers. Due to this, migration started into a microservices-oriented architecture (Section 3.1.2). This means that now the database is not the single point of communication, and each of the services would need an independent database to store the data they needed to operate. The acquire and monetize units can easily be decoupled using that paradigm, and different teams could work independently. This initiative led the company to its current setup, which can be observed in figure 5.

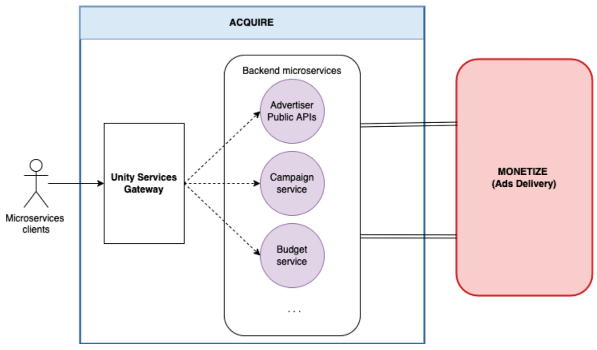


Figure 5: Current architecture of Acquire unit of UnityAds

On this new setup, Monetization and Acquire can be considered domains on the domain-oriented microservice architecture (DOMA) of UnityAds (Section 3.1.3), where different protocols are used for the intra-domain and extra-domain communication, such as NATS [64], Kafka [46], HTTP [29], or even gRPC [74]. Moreover, the Unity Services Gateway serves as the single entry point for the domain, and the backend microservices.

2.1.3 Microservices monitoring

The last step to understand UnityAds architecture is to explain how microservice monitoring is done. It is essential to state that all the architecture is deployed on Google Cloud [22], which already offers some pre-defined tools that simplify the operation of the product. Apart from that, on Figure 6 it can be seen the selected monitoring technologies for the Acquire architecture.

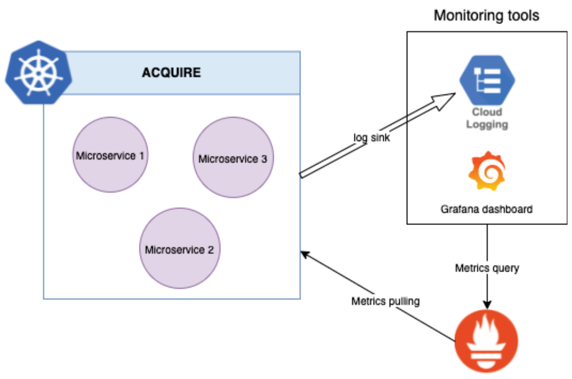


Figure 6: UnityAds microservices monitoring diagram

This architecture is running on a Kubernetes cluster that is deployed on Google Cloud [40]. For logging, the services are using the standard Google Cloud Logging tool [6], which simplifies the process for the storage and exploration of them. But logging is only one of the relevant parts for monitoring, as metrics are also essential to know the overall system's health.

In order to collect the metrics for the different microservices of the system, Prometheus [70] is the selected tool. It works by pulling relevant metrics from microservices and infrastructure, and afterward, those metrics can be accessed using its custom query language. On top of it, some Grafana dashboards [39] can be handcrafted to visualize those metrics on time series plots. This typical setup works well as it combines the possibility of exploring error logs and even tracing the endpoint calls for the microservices on a fine granularity, using standard Google Cloud tools such as the profiler [68], and at the same time presents a visual representation for the metrics of the system, such as error rate, uptime or throughput, using a time-series format.

2.2 Problem definition

The monitoring setup adopted by the Acquire unit has some limitations related to the monitoring of the Unity Services Gateway. First of all, it is noteworthy to say that Unity Services Gateway is another microservice, so the setup represented in figure 6 is enough for metrics and architecture monitoring. Nevertheless, it has some limitations for business logic monitoring [73].

A gateway of a Domain-Oriented microservices architecture (DOMA) is in charge of handling some high-level responsibilities (Section 3.2). The main advantage of that setup, is that those tasks are shared across microservices and there is a single place where this logic is implemented and maintained. But on the other hand, the teams responsible for the microservices that are directly dependant on the gateway might want some visibility on the events generated by handling that logic.

To give a concrete example for this problem, let's consider only rate-limiting responsibility [56], which is vital to prevent common attacks [43] such as DDoS [49] or unlimited API scraping [34]. The gateway's commitment corresponds to automatically rejecting the requests that had tried to access the server multiple times. To do that, it has to keep a temporal database that stores the count for how many requests the user had done on a concrete time window.

Considering this error type, different stakeholders would be interested in knowing more about the events associated with this error and that currently don't have that much information. The following subsections are will introduce them.

Public APIs team

This team is responsible for developing a set of public APIs that allow advertisers to manage their advertisement campaigns programmatically. Usually, those advertisers had built custom integrations with UnityAds and want to run adverts on different platforms simultaneously to maximize the visibility of their campaigns.

The engineering team is contacted when some advertiser is constantly receiving errors from the APIs because it has reached the established rate limit for the endpoint they are trying to consume. This can happen for a variety of reasons, maybe they want to fetch multiple resources at the same time, or there is an error in the implementation that causes the endpoint to consume the rate limit quota. With the current setup, it is tough to trace back the reasons as even if the engineering team can access the logs, as they might not contain all the information in a convenient format.

Product managers and client partners

Product managers and client partners are the intermediaries between the engineering team and the clients that consume the APIs or any other connected service through the gateway. This is also why they want to get some visibility in which clients are hitting the rate limits on the gateway level as they can contact them directly to ensure that their integration or their usage of the company's services is correct and give assistance in that regard if needed. In that way, customer satisfaction could be improved with respect to the current status, where is the client who usually contacts if they are experiencing some errors.

In that sense, this team is interested in having some visual tool to access that information because accessing the logs console using a series of queries requires some previous background that those professionals might not have.

Security team

The security team wants to know more when specific clients or users make several requests to the services. In case those calls are coming from a malicious user or a not-identified client, it could be banned for preventing any further attack [43].

3 Literature review

This section aims to provide a broad review of some topics relevant to the development of this thesis. The first subsection provides a review of some of the most widely-used service architectures, including DOMA. Then, the second subsection offers a review of the microservices gateway from the literature point of view, and the third section explores some of the most popular open-source gateways. Finally, the last section explores some previous work on the monitoring and observability fields.

3.1 Software architectures review

Domain Oriented Microservices Architecture (DOMA) is the paradigm that many large organizations are adopting to organize their software systems and the most suitable to describe the software architecture of the Acquire unit (Section 2.1.2). This section offers a review of the software architectures that led to the development of DOMA.

Software architecture is a fundamental topic in software engineering. Its main objective is to investigate the optimal way to organize software processes and artifacts to ensure that they can function properly and communicate with each other to accomplish their tasks [61]. According to E. Perry and L. Wolf [54], the term *"architecture"* was introduced into software engineering in the 1980s to evoke the notions of abstraction and coding. It is still relevant today to define the layout of software systems in different enterprises.

One of the initial paradigms that established a breakthrough in how software systems were organized was the introduction of object-oriented programming (OOP) [26], which was the first attempt to encapsulate data, offering a limited set of operations to access and modify it. It also introduced the concept of interfaces [62], which are the entry point that agents external to the object had to access. Although OOP established a solid foundation for creating more complex systems, it was still not sufficient to map business functionalities, which were later satisfied using more complex software paradigms. In the following subsections, those complex paradigms that are capable of mapping business functionality directly will be explained.

3.1.1 Service-oriented architecture

Service-Oriented Architecture (SOA) is a software paradigm that consists of packaging software systems into services that can be presented to the user in condensed packages that can be accessed through a well-defined protocol [63]. The rise in popularity of SOA coincides with the establishment of web protocols that allow software systems to communicate effectively.

The usage of those services can be done in two different ways. On the one hand, services can be consumed at run time by binding them into the code; thus, the packaging of the services is done in software libraries [23]. On the other hand, those services can be consumed using some web protocol; in this case, some standards can be used to establish a data format, such as Web Service Description Language (WSDL) or Simple Object Access Protocol (SOAP).

Even though SOA is a giant step forward in the design of software systems, mainly because it helps on the scalability and reusability of components effectively, it can also present some drawbacks:

- Services can contain many functionalities and a big codebase, which can be challenging to manage if the engineering team is constantly growing.
- If the business domain requires many functionalities that can be grouped logically in the same service, it can rapidly become a monolith.
- Adding new functionalities to the system usually implies that the whole service has to be deployed simultaneously, which can be error-prone.
- Data storage is a central piece of software systems. It can also be the communication point between services, which can cause inconsistencies as several services might get access to the same data.

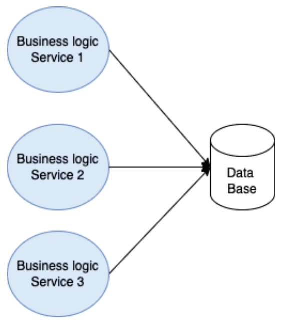


Figure 7: Diagram describing SOA

This software architecture might be helpful in the initial stage of companies or organizations, where services are minimal and the functionalities are not entirely developed. But the pain points might arise when operating on a different scale. The initial architecture of Applifier (figure 4) can be considered as SOA, as it shared the same structure and had to deal with similar drawbacks.

3.1.2 Microservices architecture

Microservices architecture is a novel approach that has gained a lot of interest both from the industrial and academic perspectives to solve the problems that traditional service architecture presents.

Many authors have tried to give precise definitions for microservices to establish a framework that can help both engineers and researchers develop applications that could tackle the original problems of software services.

One of the most relevant definitions is done by Dragoni et al. [28]: *"A microservice is a cohesive, independent process interacting via messages."* This definition is quite interesting because it is not centered on the size of the service, which is one of the debate points of the description due to the introduction of the word *"micro"* on it. It also emphasizes the concept of independence and remote communication, one of the critical points of modern microservices, which are usually deployed independently and communicate with each other using a well-known protocol.

In that regard, the definition of Johannes Thönes certainly expands some relevant aspects [69]: *"Microservice is a small application that can be deployed independently, scaled independently, and tested independently, and that has a single responsibility."* One of the emphasis points on this definition is the independence on the deployment and testing, which is one of the key advantages that make microservices useful when managed by a growing team of engineers. Apart from that, Thrones points out the single responsibility principle that should always be kept in mind when designing microservices.

Following the classification of Lewis et al. [42] microservices should share the following characteristics:

- **The modularity of services**. As in SOA, microservices offer a modular functionality that can be reused easily. The main difference between both is that microservices could be deployed independently, whereas, on SOA, each release implies a complete deployment of the service.
- **Organised around business capability**. On traditional software systems, teams are organized into different functional areas; for example, one group would be responsible for engineering one service and another for deployment and testing. In that schema, if there is a failing deployment, the responsibility is shared across the teams, which can cause some delays as the issue can not be handled within an organizational unit.
- **They are product not projects**. In SOA, each service is mapped into a business requirement; on the contrary, on microservices, one business requirement aims to be mapped to multiple microservices to reduce the amount of responsibilities of each one.
- **There are intelligent endpoints and dumb pipes**: The main logic has to be placed on the microservices that are accessed via endpoints. There have to be few communication middlewares (pipes) with actual logic.
- **Decentralized data management and governance**: One of the main characteristics of microservices is that data is no longer the common point between services because each microservice has its independent data store.

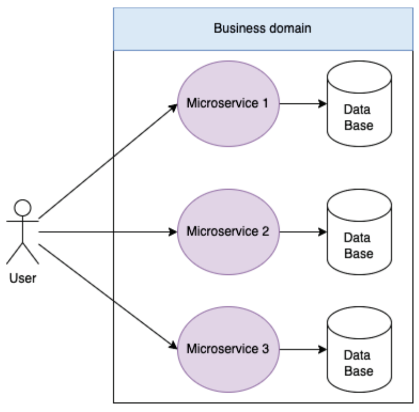


Figure 8: Diagram describing microservices architecture

Microservices represent a giant step forward in developing software architectures, as they present many *operational* benefits compared to SOA. Nevertheless, they are not a silver bullet as they also introduces some significant drawbacks, such as:

- The introduction of performance issues. The remote procedure calls (RPC) used between microservices to communicate can introduce some delays as they have to be resolved over the network.

- Problems on the data aggregation. As microservices remove a shared data storage in favor of a decoupled data governance if some complex data has to be retrieved that involves some integration between different sources.
- Difficulties in introducing new features. In a microservice architecture, introducing a new feature might involve changing several services, which can delay communication, validation, and code review.

3.1.3 Domain-oriented microservices architecture

The latest revised architecture will be DOMA, presented by Gluck A. in 2020, on behalf of Uber's engineering team [11]. The primary goal of DOMA is to mitigate the problems that microservices architecture presents in large enterprises that have hundreds of interconnected microservices.

In these companies, there are some additional problems to those discussed in Section 3.1.2:

- Knowledge sharing and architecture onboarding. When the number of microservices is significant, the understanding of the system becomes complex. That increases the onboarding time for new engineers, which causes an operational cost for the company.
- Difficulties to to debug and identify a problem. When considering hundreds of interconnected services, if a call in any of them introduces some delay or bug, engineers have to trace back the call stack, requiring substantial cross-team effort.

DOMA is a new software design pattern iteration that tries to tackle these problems. The main principles are extracted from Domain-Driven Design, a paradigm introduced by E. Evans and EJ. Evans [31], which defines software design based on the domain or business-specific needs. In that regard, DOMA is the direct application of Domain-Driven design to microservices design in an extensive network of distributed and interconnected services.

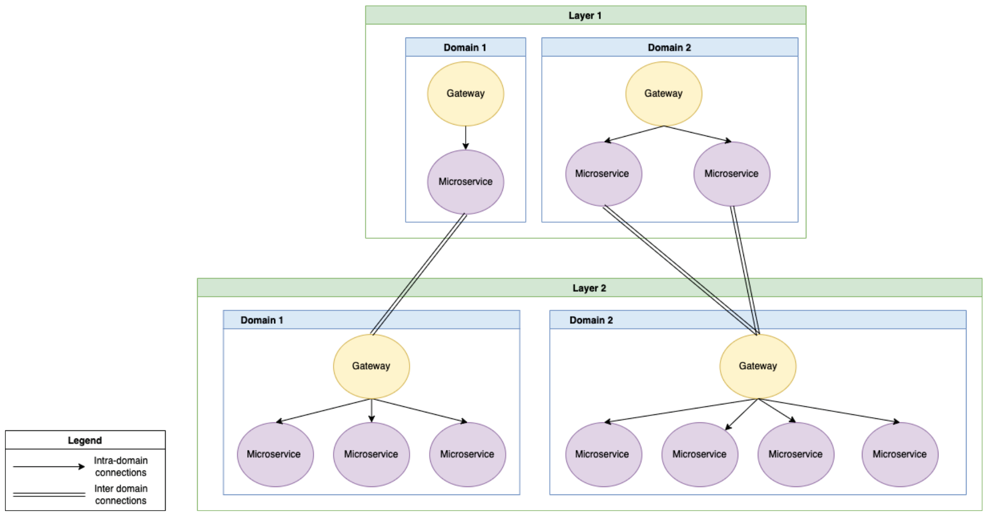


Figure 9: DOMA architecture diagram, with representation of the two communication types

DOMA is adopted on organizations as an iteration of a pre-existent microservices software architecture. Four main elements have to be introduced [11], to do that transition:

- **Domains**: A domain is a collection of related microservices. The main design challenge of this element is to decide how big the domain should be and how many microservices should contain.

- **Layers**: A layer is a collection of domains. The primary behavioral change is that we can consider that the layers are grouped into a stack, where the layer defines the dependencies between the domains that they belong to in the pyramid.
- **Gateway**: The gateway is the single point of access of each domain. It is the public interface of the domain, and it should prevent access to individual services and implement several high-level responsibilities.
- **Extension architecture**: the extension mechanism is a procedure for adding extra functionalities. This is important to respect the open-closed principle of the domain. It is remarkable to state that this can not be achieved on all domains as it depends on the nature of the handled data.

The introduction of DOMA is recent, so there is not much research on how to populate those four elements with concrete business requirements to transition a microservices architecture into a successful DOMA. The only available public example is offered in the article by Uber Engineering [11], where they provide an interesting view on how to distribute the layers pyramidally, where the base of the pyramid represents broad functionalities that can be used across the organization, such as storage or networking services. In contrast, concrete client-facing functionalities are at the top of the pyramid. This interesting approach guarantees that the layers are isolated, and dependencies flow from top to bottom.

3.2 Microservices gateway

This section is intended to extend the knowledge presented in the previous section about microservices gateways. As stated before, gateways are an essential element in DOMA, and their primary purpose is to isolate the domains of microservices and work as the single entry point. Even though they represent a crucial element in DOMA, they can also be used in a simple microservices architecture.

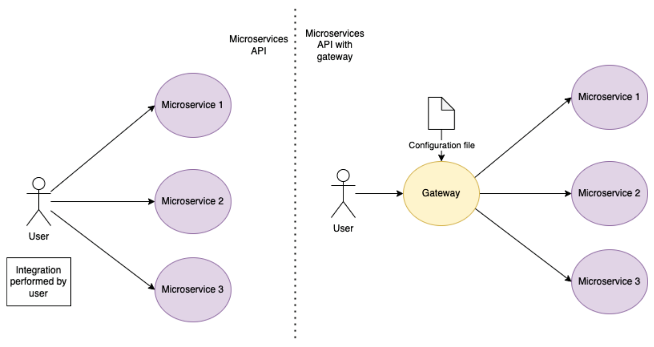


Figure 10: Diagram comparing the call of a microservice architecture with or without gateway

On Microservices Patterns, by C. Richardson [58], several reasons why creating an API gateway is vital on a microservices architecture are introduced.

- Each microservice exposes a minimal set of fine-grain API endpoints. To compose a more meaningful and high-level API call, the information of many microservices has to be integrated somehow. That responsibility should ideally not lie on the consumers of the services.

- The users do not need to know the internal details of microservices architecture to respect the company's privacy and internal structure. Due to this, they rather interact with some proxy instead of calling the specific microservices.
- The communication protocols between microservices can be very diverse. Clients would prefer only to use one communication protocol to request resources on the systems rather than preparing their custom data integration.
- Several high-level responsibilities are shared between microservices, and they would be better managed on a central point rather than re-implementing the logic for each service.

Each point of this list defines a valid reason for considering the creation of a microservices architecture gateway; nevertheless, the last one is the most relevant for this thesis and is the one that will be explored in-depth.

As this thesis aims to monitor the high-level errors of a microservices architecture gateway, it is necessary first to define the responsibilities it has to take. The high-level responsibilities stated by M. Thangavelu et al. [30] will be considered for this definition.

- **Auditing pipeline**. maintaining a registry of all the access and operations that the gateway performs. This thesis aims to improve the logging when referring to the error generated by the stated responsibilities.
- **Identity**. Each access to the service has to be authenticated and authorized, and that responsibility should lie on a central software piece like the gateway.
- **Rate limiting**. That refers to the capacity of the system to prevent malicious attacks caused by an uncontrolled number of recurrent requests.
- **Documentation**. Ideally, all API calls should be documented on a central repository with a standard format.
- **Response fields trimming**. As the gateway works as a proxy for the backend microservices, it would be helpful to have the possibility to specify the omission of specific fields on the user's responses.
- **Datacenter affinity**. There can be some logic on the gateway that could redirect user calls to certain physical regions where the data can be retrieved faster.
- **Short-term user bans**. Temporarily banning users who had malicious behavior on the system can be one of the solutions to deal with cyber-attacks. That responsibility should be delegated to a central software piece rather than be shared across the microservices.

Considering those high-level responsibilities, in Section 5.1 the errors related to each of them will be defined.

3.3 Open-source microservices gateways

This section will explain two of the main industry-relevant microservices architectures gateways that are open-sourced and can be incorporated into any company's current architecture. The main idea is to simplify the development of a gateway, having all the benefits already explained in Section 3.2, while also reducing time-to-market as the primary development effort can be centered on creating the end-product. The two gateways that will be relevant for this analysis are Kong and Zuul, both open-source solutions.

- **Kong Gateway**: Kong is the most popular open-source microservices gateway nowadays [12]. It is a containerized Lua application that can be extensible by custom plugins and executed within a Kubernetes deployment. Its main advantage is that it is fully configurable, so the different high-level responsibilities can be configured depending on the protocol or technology that is preferable to use.
- **Zuul Gateway**: The Netflix engineering team created Zuul gateway as part of their internal stack and then released it as open-source in 2018 [16]. Its main characteristic is that it is built for resisting a high volume of queries. It is the central entry point for all the cloud infrastructure at Netflix, having more than a million requests per second. The architecture of it works in layers. First of all, there is a module that sets all the correspondent network settings and connection protocols, and then there are different filters that are used for decorating the requests before being forwarded to the backends.

As it can be seen, both options offer a solution for companies that operate on a big scale and that have a considerable volume of traffic. In Section 6.2, those solutions will be examined in depth to be compared with the proposed solution in this project's scope.

3.4 Observability and Monitoring

Observability and monitoring are hot topics in software system maintenance, and it is especially relevant to review them applied to microservices architecture [67]. Hence, the objective of this section is to give a theoretical introduction to both topics, a review of the current *state-of-the-art*.

It is common to confuse both observability and monitoring as their objective is to check whether something in the system is not working as it should [67]. Nevertheless, monitoring is based on taking metrics of quantifiable elements such as CPU usage, memory usage, or network traffic [41]. The basic procedure for monitoring any system is by querying it directly to obtain those metrics and set up visual dashboards and alerts that will warn if the behavior is not the expected. This method is quite effective for maintaining traditional monolithic applications' health as the resources they consume are centralized, contrary to microservices or DOMA, where they are distributed over a network. Also, monitoring presents a clear disadvantage as the problems are usually identified when the metrics' values show a higher value than a certain threshold, which might be too late to react to an incident.

On the other hand, observability is similar on purpose but different on methodology. Rudolph E. Kalman introduced the term observability in 1960 on the control systems field [45]. The basic definition is: *"Observability is a measure of the wellness of the internal state of a system inferred from the knowledge of its external outputs."* Even if it was first invented for the control systems field, observability has been applied to modern software architectures, such as microservices. Attending to that classic definition, the inference of the internal state had to be done by producing a set of outputs from the highly-observable software service that it is being maintained, and that is done by generating a set of logs and traces [55].

It is also relevant to review the previous studies in the field of services monitoring. To narrow the scope of the research, only previous studies centered on microservices are considered. Furthermore, only the ones provide a novel approach to bridge the gap between the complexity of distributed microservices architecture and its monitoring [53]. First of all, Yang et al. [75] propose a new approach for capturing the request execution path (REP) transparently. This novel method categorizes the events generated at a microservice level, and the relationships between those events are identified to construct a monitoring tool that could effectively trace the REP. The way this study categorizes the events is similar to the work done for the categorization of the DOMA Gateway events done for this thesis (Section 5.1).

Sun et al. [65] presented an analysis of many microservices architectures on the complexity analysis topic, pointing out that they do not provide any valuable information about their internal state. Hence, the observability procedures were not working correctly, and some improvements have to be made.

Jongh et al.[44] formulate a different study where benchmarking and simulation are used on a combined approach in order to monitor execution times. This study is centered on monitoring execution performance, offering a concrete technical solution instead of concrete surveys and definitions.

In that regard, Lin et al. [47] present a study whose aim is to propose a new method for root cause detection, where the errors are identified using causal graphs. This work focuses on identifying and monitoring the errors caused in distributed systems and how they are propagated across the architecture.

Finally, Gupta et al. [35] proposed a new way for monitoring continuous deployment on rapidly changing companies (with the independence of the size of the product). On that approach, data mining techniques are applied to the logs of highly-observable microservices to discover the differences between deployed versions on microservices.

4 Requirements engineering

Having presented the background necessary to understand the thesis project in Section 2, this section will cover requirements engineering.

Requirements engineering is the process of establishing the desired requirements to be met by the project [38]. Section 4.1 will define the methodology to be followed to gather the requirements, and Section 4.4 will analyze these requirements. Finally, Section 4.5 will present the research questions that will serve as the guiding thread of this thesis.

4.1 Requirements engineering methodology

Before performing the requirements analysis, it is essential to clarify which methodology will be followed to identify the project requirements. In this case, the framework proposed by David C. Hay [38] for requirements engineering, which consists of three steps, is selected:

- **Eliciting requirements**: this phase is where all the requirements will be discovered following the selected gathering methodology.
- **Recording requirements**: The second phase consists of the documentation of the requirements previously collected.
- **Analyzing requirements**: Finally, those requirements have to be analyzed in order to see if they are clear, complete, concise, and unambiguous.

For the elicitation of requirements, the selected method is stakeholders identification [50], which consists of selecting individuals or groups interested in the product or function to be developed and asking them what the desired characteristics they expect from the product are. This method is the most appropriate for this thesis since it will only have an internal impact as external customers will not use it. The list of stakeholders for this project is given in Section 2.2.

4.2 Requirements recording and gathering

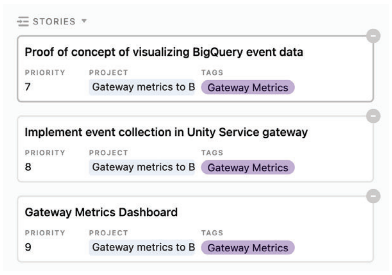
Once the methodology for the requirements is defined, and the relevant stakeholders identified, it is time to explain how those requirements were recorded.

To record and document the requirements expressed by the stakeholders, there were two methods: the first was to use message threads in Slack, which is the internal communication platform, where relevant stakeholders could set their expectations, and the second method was to organize some meetings, based on the framework of the Joint Requirements Development (JRD) Sessions [38], to discuss the prototype and the expected results. At these meetings, a log was drafted to keep track of the issues discussed.

Apart from that, a technical design document (TDD) was written about the implementation phase, where all the requirements gathered via slack messages could be stated down. That document was in review process for one week, and the relevant stakeholders could express their comments about it.

4.3 Prototipe implementation planning

Before diving into the thesis's technical implementation, it is relevant to give a brief overview of the planning. To develop the prototype, we used scrum as the agile methodology [60], dividing the tasks into specific user stories and tasks that were executed on different sprints. The tool used for the project planning was AirTable, which primary usage was to introduce the three stories that configure the implementation phase and give an estimation based on the workforce that will execute it.


STORIES

Proof of concept of visualizing BigQuery event data

PRIORITY	PROJECT	TAGS
7	Gateway metrics to B	Gateway Metrics

Implement event collection in Unity Service gateway

PRIORITY	PROJECT	TAGS
8	Gateway metrics to B	Gateway Metrics

Gateway Metrics Dashboard

PRIORITY	PROJECT	TAGS
9	Gateway metrics to B	Gateway Metrics

Figure 11: Project planning on AirTable

The three stories that configured the implementation phase were:

- **Proof of concept of visualizing event data**: As we were unsure how to use the selected technologies for development, first, it was needed to build a proof of concept with dummy data to configure all involved parts. Also, this story included the publishing of the RFC for the event structure described in Section 5.2.1.
- **Implement event collection in Unity Services Gateway**: This story includes the implementation of the events collection on Unity Services Gateway as well as the setup of the data pipelines, described in Section 5.2.4 and 5.2.5 respectively.
- **Gateway metrics dashboard**: This story covers the implementation of the custom plots used on the visualization step; this is explained in more detail in Section 5.3.

To fulfill the three stories, the total execution time was one month, which was close to the estimated time, only differing on a couple of days.

4.4 Requirements analysis

The last step of the chosen requirements engineering framework consists of performing an analysis of the requirements gathered from stakeholders. In that sense, the solution that is going to be presented for solving the problem stated in Section 2.2 needs to meet the following requirements:

- Clear definition on which are going to be the high-level responsibilities that the gateway will handle and which errors the gateway can generate associated with those.
- Establish a format for the generated errors. They need to store all necessary fields in order to be traced back and visualized afterward.
- Map possible errors and gateway responsibilities to the different endpoints that clients can consume.

- Select a suitable data store for the errors. Data storage needs to be easily integrated with the current cloud setup and must ensure the acceptance of a standard query language that allows the integration with third-party tools.
- Data ingestion has to be cloud-agnostic. This means that the error logging pipeline, from the event emission to the storage, has to be done with a technology that ensures compatibility with multiple cloud providers.
- Develop a visualization tool that allows stakeholders to access the errors generated at the gateway level easily.

4.5 Research questions

This thesis aims not only to satisfy the specific need for a technical feature but also to generate a set of tools and practices that can be used in the future by any other organization operating a DOMA. Therefore, it needs an extensive investigation and planning phase, which these three research questions will guide:

Q1 *What is the best way to represent the possible high-level errors that a DOMA gateway can produce?*

Q2 *What is the optimal technology for storing the errors that a DOMA gateway can produce?*

Q3 *How to correctly visualize and communicate with the stakeholders the errors a DOMA gateway can produce?*

The objective of the thesis is to produce a series of software artifacts and formats that will satisfy the research questions and, therefore, the technical requirements of the stakeholders.

5 Technical solution

This section will cover the technical solution developed on the scope of this thesis. The first subsection will introduce the gateway configuration file, the contract used to match the endpoints that the gateway exposes with the high-level responsibilities it will address. The second subsection will explain the error definition format and the data pipeline implemented to store the errors generated at the gateway level. Finally, the third subsection will introduce the visualization dashboard for the gateway errors.

5.1 Gateway configuration file

After having discussed the high-level responsibilities that the gateway of a DOMA should have (Section 3.2), in this chapter, one of the main outputs obtained from this thesis is introduced, which is the definition of a route-level configuration format for the gateway in order to solve **Q1**.

A route-level configuration format can be used to state down all the resources that the gateway of the domain exposes to the public and the associated responsibilities attached to each route. Having this standard defined will have many significant advantages, such as:

- It can be used as the primary source of truth of the resources that the gateway exposes.
- The generation of the endpoint's documentation can be automatized; therefore, all the teams involved in the microservices do not need to do it by themselves.
- This format defines a contract on what the endpoint handlers should do when managing the responsibilities. The engineers that are going to implement the proxy should fulfill the specifications of the standards, and the endpoints consumer should rely on those responsibilities.
- There are some literature examples of composable gateway logic [30]; this format can be the starting point of this automated gateway generator.

In the following subsections, there is going to be an explanation of the general structure of this configuration file, using YAML as the format, as well as each filter that can be specified to fulfill each gateway's responsibilities (Section 3.2).

General structure

The general structure of the configuration file is as follows:

```
- path: path/to/my/resource
  envs: ['prd', 'stg', ... ]
  methods: ['get', 'post', 'put' ... ]
  endpointFilters:
    ...
  transport: 'http'
```

There is one entry per resource on the gateway configuration file, and the following elements will specify each entry:

- `path`: This is the URL that will be used to do the operation on the system, and it is also the identifier of the resource.
- `env`: The environment where the gateway is executed. Usually, there are three different environments considered in software engineering: local, staging, and production.
- `methods`: The HTTP method to be used: put, post, get, patch... those method corresponds to CRUD operations [48].
- `endpointFilters`: Those filters are the responsibilities that the gateway will take. They are going to be defined in the following subsections.
- `transport`: The transport method used. Usually, it will correspond to HTTP, but other protocols like NATS can be specified.

Auditing filter

The first filter specified, which is going to be contained under the `endpointFilters` specification, is going to be the filter used for the auditing responsibility of the gateway. The structure is as follows:

```
  endpointFilters:
    - auditing: boolean
```

To keep this filter simple, there will be only a boolean flag that will determine if there has to be any kind of logging or auditing at the gateway level for the specified endpoint.

Identity filter

This filter specifies if there has to be any authentication or authorization for consuming the endpoint resources. The structure of the filter is:

```
  endpointFilters:
    - authentication: service-account | OAuth
    - authorization:
        create: permission-identifier
        read: permission-identifier
        update: permission-identifier
        delete: permission-identifier
```

First of all, the authentication section specifies how the user has to be identified on the system. In this example, two options are shown: service accounts and OAuth [20], but there can be many other methods and should be specified by the systems designers.

On the authorization side, the consideration has been to use Role-Based Access Control (RBAC) [32], which assigns one role to each user and specifies a set of permissions for each role. The specific CRUD operation can only be performed if the permission matches the one resolved for the user who tries to consume the resource.

Rate limiting filter

Rate limiting filter defines the number of access that the gateway will accept on a certain period of time. The structure of the filter is the following:

```
endpointFilters:
  - httpMethods: [get, put, patch, delete] // http methods to
 limit
  - perSecond: {prd: 6, stg: 100, local: 10000}
  - perThirtyMinutes: 4000
```

There are three fields that the filter accepts; the first one is the HTTP methods that are considered for the rate-limiting, the second is the number of accesses that the gateway accepts in one second, and the third one is the number of access that it accepts in thirty minutes. For the last two properties, it can be specified the number of accesses or an object with three properties referring to the three possible environments and a number for the rate-limiting on those.

Documentation filter

This simple filter specifies if any documentation for the endpoint at the gateway level has to be produced. There might be private endpoints where the consumption is done only within the organization, so creating external documentation might not be desired. The structure of it is as follows:

```
endpointFilters:
  - documentation: boolean
```

Response fields trimming

As the gateway should work as the proxy for all the microservices under a domain, we may want some fields not exposed outside of it, which is the purpose of the response field trimming. The structure is as follows:

```
endpointFilters:
  - responseTrimming: [field1, field2, ...]
```

The fields that are included in the array are not going to be forwarded to the endpoint consumers.

5.2 Events collection and storage

As stated in Section 4.4, one of the requirements for this thesis is to create a tool that stores and visualizes the high-level errors that the gateway is generating. This section covers the development of that tool, explaining the definition of the error format, the creation of the events gathering, the data pipeline for storing those events and finally the selection and setup of the data storage technology.

5.2.1 Error event definition

This subsection covers the definition of the error events that will be collected on the data storage to be visualized in the future. The main objective of this is that it should share many standard fields that can be used to identify the error for all the gateway high-level responsibilities (Section 3.2).

For doing that, the approach consists of having some general fields independent of the error event generated, which will be used to locate the endpoint that was called and that caused the error. Apart from that, some specific fields will be stored depending on the high-level responsibility that caused the error, and the purpose for this is to create meaningful visualizations afterward. It is essential to state that only the rate-limiting responsibility was considered for the initial implementation as it is the most important for the stakeholders. Nevertheless, it can be extended in the future as many fields can be added easily to the events.

The procedure for defining which are the meaningful fields that had to be stored was by presenting an RFC (Requests for comments) [57] internally on the gateway development team, so all the people involved on it could freely give comments and ideas. As the company is using Slack as the communication platform, the RFC was shared in a Slack thread. The number of people present on the slack channel was 294 when the RFC was published, and there were 139 replies to it from people from different teams, such as project managers and engineers.

After that internal discussion, the whole team agreed on a set of fields to be included in the error event:

- `source`: It is the type of event that will be stored. They should match the gateway responsibilities (Section 3.2), but as stated before, as only the rate-limiting responsibility is considered for this implementation, the only possible value is `RATE_LIMIT`. Some other values like `AUTHENTICATION` or `AUTHORIZATION` can be added in the future.
- `requestId`: This maps the request to a unique id that can be used to trace it back on the microservices calls. The gateway defines this unique id.
- `path`: The completed path of the resource requested on the query. In the context of the Unity public API an example of it can be:
`/advertise/v1/organizations/:organizationId/apps/:appId`.

- `organizationId`: It is the organization id extracted from the request's path. If that parameter is not present on the path, the value is `null`.
- `httpMethod`: This value is the HTTP method used in the request. This parameter is essential for distinguishing between different endpoints.
- `apiVersion`: As REST APIs are frequently versioned [71], this parameter aims to contain the different versions of the API. It is also extracted from the path, so it has a null value if it is not present on it.
- `apiType`: This parameter distinguishes between the two types of API endpoints, `private` and `public` ones. Private endpoints are used only internally, whereas public endpoints can be consumed by any external client.
- `apiNamespace`: In the context of the company, there are different types of APIs: monetization, advertising, cloud game..., and those categories are considered namespaces. This field represents the identifier of the namespace.
- `timestamp`: Timestamp on Unix standard time for when the event was triggered.

Apart from the previous fields, which are the common ones and shared across different errors, there is also the definition of specific fields about the rate limit responsibility:

- `rateLimitReason`: this is why the rate-limiting was activated and which of the two maximum rates was reached. It can be either `tooManyRequestPerSecond` or `tooManyRequestsPerThirtyM`
- `quota`: Number of requests that made the rate limit reach the maximum amount.
- `group`: Rate limiters can be grouped to be shared across different endpoints. For that, the parser of the gateway responsibilities file (Section 5.1) had to be able to apply shared rate-limiting responsibilities to other endpoints. This field stores the identifier of the rate limiter that was hit. In case it is not present, it should have a `null` value.

After having defined the data structure for the events, the following sections will cover its technical implementation, from the collection of the event on the gateway side to the actual storage on the databases.

5.2.2 Data storage technology selection

This section aims to explain why a particular storage technology is selected for storing the generated gateway errors, thus answering **Q2**. To define the optimal storage technology, three criteria have been considered.

C1 **Price**: The monthly price that the organization has to spend on the selected technology.

C2 **Data access**: This criterion sets how easy it is to query the data stored using an standard query language such as SQL.

C3 **Integration with the current cloud setup**: most modern organizations, especially those that are big enough to deal with a gateway for a microservices architecture, are operating using a cloud provider. In that sense, this criterion sets the easiness for integrating the storage technology with the current cloud provider.

The answers for this questions might differ depending on different organizations, but should be enough for deciding the best choice among the considered technologies. For the scope of this thesis and considering the events generated on the Unity Services Gateway, there are three storing technologies considered in the comparison:

- **BigQuery** [3]: This is a data warehouse designed by Google to store, efficiently access, and analyze enormous amounts of data. One of its most interesting characteristics is that it allows access to that data using standard SQL statements.
- **Amplitude** [1]: This is a tool used to store events produced on applications and analyze them. It is more common to use it in front-end applications to analyze user behavior.
- **PostgresSQL** [17]: PostgreSQL is an open-source relational database. It has to be used on an instance in the cloud provider of our choice.

After having defined those three options, they will be compared using the three criteria on separate sections, and finally, the results will be shown in table 1.

C1 - Price

It is essential to set some common values needed to compare different technologies' prices.

- **Requests per month** (N): It is considered that 50.000 events per month is the upper bound for the number of events for all endpoints.
- **Size of the insertion request (KB)** (w_i): Another consideration is that the upper bound for insertion is 100KB per event.
- **Maximum size per retrieve request (MB)**: One consideration is that there can be fetched 1000 records at most when retrieving the information. This limit is set as it is the limit of Grafana, the visualization platform.
- **Storage per year**: Considering the number of monthly requests and the size of each new record, the storage that we will consume is around 50GB per year.

After having those variables settled, the cost calculations are going to be done for the three different storage technologies considered.

BigQuery

First of all, the costs for BigQuery are going to be calculated based on the official documentation for the tool [15] and on the article by Welch Al [10]. On BigQuery, there are three different types of costs: storage costs, query costs, and insertion costs.

First, storage costs are going to be calculated. Storage costs are different depending on active storage (\$0.02/GB/month) and long-term storage (\$0.01/GB/month). The consideration of a record as active or long-term is decided by Google and depends on the last time accessing it. The calculation is that 15GB would be active storage, and there would be another additional 35GB as long-term storage.

$$\begin{aligned} C_{storage} &= \$0.02 \cdot S_{active} + \$0.01 \cdot s_{long-term} \\ C_{storage} &= \$0.02 \cdot 15GB + \$0.01 \cdot 35GB \\ C_{storage} &= \$0.65/month \end{aligned} \tag{1}$$

Secondly, the query costs are going to be estimated. Those costs are calculated per terabyte processed ($5 / TB), so the estimation has to be in that unit. The three variables involved in that calculation are the following:

- **The number of users per day** (n): The estimation is that 20 will be the upper bound for daily users, as it is an internal tool that is usually not consulted every day. Considering that they access five times, the total will be around 100 estimated daily individual accesses.
- **The number of queries per user** (q): The estimated number of queries is 25 for the initial implementation of the visualization tool.
- **Average data usage** (d): The data usage is the number of max records fetched per query. This number was set before to 1000 records, so the calculation in TB is the following: $100KB \cdot 1000 = 100000KB = 1e^{-4}TB$

$$\begin{aligned} C_{query} &= \$5 \cdot (n \cdot q \cdot d \cdot 30) \\ C_{query} &= \$5 \cdot (20 \cdot 25 \cdot 1e^{-4}TB \cdot 30) \\ C_{query} &= \$7.5/month \end{aligned} \tag{2}$$

Finally, the last type of cost being addressed is the insertion cost. This one is the easiest to calculate, as it only consists of $0.01 per 200MB inserted. So the total insertion cost will be:

$$\begin{aligned} C_{insertion} &= \frac{\$0.01}{200MB} \cdot N \cdot w_i \\ C_{insertion} &= \frac{\$0.01}{200MB} \cdot 50.000 \cdot 0.1MB \\ C_{insertion} &= \$0.25/month \end{aligned} \tag{3}$$

Considering those three costs, the total monthly cost (C_{total}) for BigQuery is estimated as it follows:

$$\begin{aligned} C_{total} &= C_{insertion} + C_{query} + C_{storage} \\ C_{insertion} &= \$0.25 + \$7.5 + \$0.65 \\ C_{insertion} &= \$8.4/month \end{aligned} \tag{4}$$

As it can be seen the total monthly cost for this technology is of $8.4 / month.

Amplitude

Amplitude's costs are divided into three categories: starter, growth, and Enterprise. On the starter version, they offer the storage and processing of 10 million monthly events, which is more than enough for covering the expected 50.0000 monthly actions. The price for this version is free, so as it can be seen Amplitude is a good option cost-wise.

PostgresSQL

Here the costs are not dependent on the technology itself, as PostgresSQL is an open-source database technology, but on the expenses related to the cloud provider, which in the case for this project is Google Cloud. To calculate the costs, the official documentation for CloudSQL [5] is considered.

In this case, the are three variables to decide that will determine the costs, which are also dependent on the zone. In this case, we are considering London (Europe west-2) for the cost calculation in order to cover European clients:

- **Number of CPUs**: It is the number of virtual CPUs allocated for processing the events, the insertions, and the queries. As the application is not customer-facing, the estimation is that only 2 CPUs will be enough. The price for each CPU is $27.13 per month.
- **Memory**: The amount of memory allocated for the processing. The estimation is to have 8GBs assigned. The price is $6.13 per GB per month.
- **Storage**: For the storage, same numbers as for the BigQuery section it is considered. The price is $0.176 per GB on SSD storage and $0.09 per GB on HDD storage.

Once those variables had been established, the total cost for this technology will be:

$$\begin{aligned} C_{total} &= C_{CPU} + C_{memory} + C_{storage} \\ C_{total} &= \$27.13 \cdot 2 + \$6.13 \cdot 8 + \$0.176 \cdot 15 + \$0.09 \cdot 35 \\ C_{total} &= \$54.26 + \$49.04 + \$5.79 C_{total} = \$109.9/month \end{aligned} \tag{5}$$

C2 - Data access

The second criterion considered is the tool's easiness for accessing data, especially if we consider using a standard query language such as SQL. In that regard, the characteristics for each datasource are:

- **BigQuery**: One of the main characteristics of BigQuery is that even though it is a data warehouse, standard SQL can be used for accessing the data on it, so that makes it convenient.
- **Amplitude**: One of the weak points of this tool is that the data can only be accessed via their application using click-ops, which makes the integration with external tools difficult.
- **PostgresSQL**: As it is a traditional relational database, it can be accessed using standard SQL.

C3 - integration with the current cloud setup

This third criterion refers to the tool's capacity to be used in integration with the current cloud setup for its purpose. The comparison between them is as follows:

- **BigQuery**: In the case of using Google Cloud Platform, BigQuery is integrated into the environment out of the box. Nevertheless, if the company already uses another cloud platform, BigQuery is not a suitable option.

- **Amplitude**: As it is an external platform, it can be easily used with any cloud provider to store and analyze events.
- **PostgresSQL**: All modern cloud providers provide access to a database instance, so it should be effortless to integrate with any of them.

Data storage selection based on the criteria

Based on the three criteria, table 1 contains a summary of the comparison between the three technologies. Based on this table and for the particular case of the Unity Services Gateway, the selected technology to continue with is BigQuery. The main factor for making that decision was that it is already integrated into the cloud environment, and it is already being used in other parts of the technical stack.

	C1 - Price	C2 - Data Access	C3 - Integration
BigQuery	\$8.4/month	Easy	High
Amplitude	Free	Medium	Medium
PostgreSQL	\$109.9/month	Medium	High

Table 1: Comparison between different data storage and data analysis technologies

For the particular case of any other company trying to accommodate this solution for their specific case, it is vital to do an analysis depending on the business conditions and then decide accordingly using those three criteria.

5.2.3 Datastorage implementation with BigQuery

Once it was decided that BigQuery was the technology to go with this implementation, the next step was to create the table containing the data and configuration. This can be easily done using the Google Cloud Console on the BigQuery section.

First of all, the database has to be created under the target project on Google cloud. In our case, it is the `unity-services-foundation-stg` project for staging deployment and `unity-services-foundation-prd` for production deployment (figure 12). In this case, two databases will be created with the exact schema both for staging and production to have a separated development environment from production.

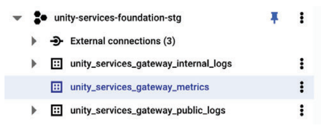

unity-services-foundation-stg
External connections (3)
unity_services_gateway_internal_logs
unity_services_gateway_metrics
unity_services_gateway_public_logs

Figure 12: Table creation on staging project, exactly the same setup is reproduced on production

The schema of those databases will follow the events structure already explained in Section 5.2.1. As can be seen on Figure 13, most of the variables are set to string as data type except for the required timestamp, used for data partitioning. It is also noticeable that all the columns set to `REQUIRED` on the database correspond to the common fields on the events previously explained in Section 5.2.1.

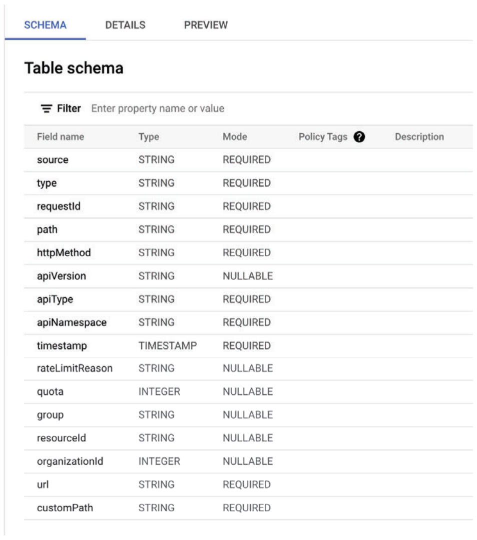


Table schema

Field name	Type	Mode	Policy Tags	Description
source	STRING	REQUIRED		
type	STRING	REQUIRED		
requestId	STRING	REQUIRED		
path	STRING	REQUIRED		
httpMethod	STRING	REQUIRED		
apiVersion	STRING	NULLABLE		
apiType	STRING	REQUIRED		
apiNamespace	STRING	REQUIRED		
timestamp	TIMESTAMP	REQUIRED		
rateLimitReason	STRING	NULLABLE		
quota	INTEGER	NULLABLE		
group	STRING	NULLABLE		
resourceId	STRING	NULLABLE		
organizationId	INTEGER	NULLABLE		
url	STRING	REQUIRED		
customPath	STRING	REQUIRED		

Figure 13: Schema of the created BigQuery table

The next steps will be to implement the data gathering on the gateway microservice and create a log sink (Section 5.2.5) for starting storing data on the warehouse.

5.2.4 Event gathering implementation on Unity Services gateway

This section explains which concrete changes had been introduced on the Unity Services Gateway codebase to support the new events gathering. A diagram for this process can be observed on figure 14.

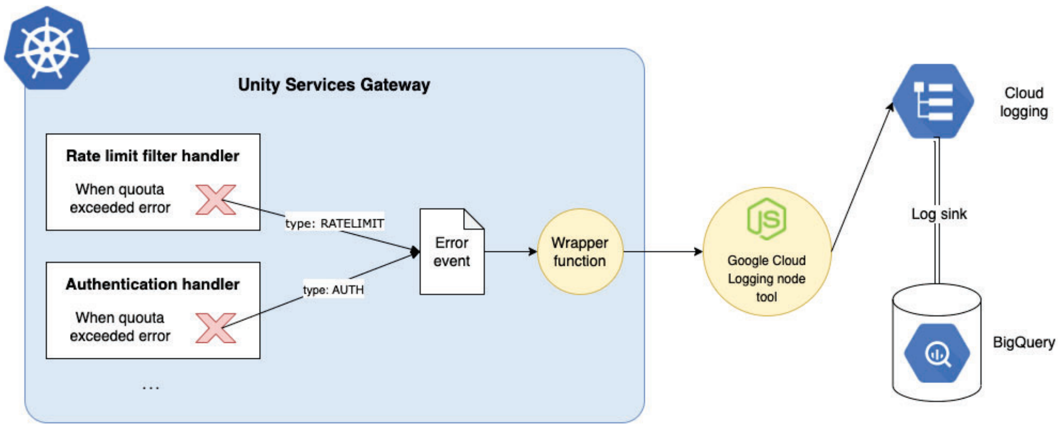


Figure 14: Diagram that shows how the events gathering work on the Unity Services Gateway and how the log sink works

The first change introduced on the gateway consists of adding a wrapper JavaScript function to the Google Cloud Logger that will construct the event structure that BigQuery expects (explained in Section 5.2.1). That function will be used instead of the current logger in the codebase where the events are triggered.The logger function is straightforward. It mainly matches the object that is given as a parameter, and then it enriches some fields from the request that it is receiving. Those parameters are the Organization ID obtained from the path parameter and the Request ID, injected by the gateway in an intermediate middleware. The code for the function can be seen at Appendix A.

As stated before, only the rate-limiting events are considered; thus, only those events have to be logged for this iteration. This means the function that performs the rate limiter has to be modified to include the logging using the previous utility. It is first essential to explain how the rate limit quota is calculated, for doing that, a Redis database is set up with the Organization ID and a numeric value that increments when a new request is resolved. When the quota is exceeded, a new event is triggered containing the information if there were too many requests per 30 minutes or second.

Finally, it is relevant to state how the extensibility for this module will be done. If an additional field needs to be added to the database, then the logging function can be modified accordingly to contain it, and in the part of the codebase where that event is triggered, that logging function should be invoked. Apart from this, a default value should be set on this new field to ensure retro compatibility.

5.2.5 Log sink configuration

Section 5.2.4 explains that the events are collected using the standard Google Cloud logging tool on the Unity Services Gateway side. Still, for this project, it is not only needed to have the logs on the Google Cloud Logging, but it also needed to be persisted on a big data tool (in this case, BigQuery) for future analysis. GCP uses Sinks [7] to do this log routing and redirection, and this section explains how they were configured in this project.

As Unity is using Terraform for configuring all the infrastructure, for creating this sink, a new Terraform resource (with type `google_logging_project_sink` [8]) has to be added. Then the configuration has to be applied to take effect. On Terraform, the structure for creating the log sink is as follows:

```
resource "google_logging_project_sink" "log_sink" {
  name                        = var.sink_name
  destination                 = "bigquery.googleapis.com/projects/${
   google_bigquery_dataset.dataset.project}/datasets/${
   google_bigquery_dataset.dataset.dataset_id}"
  filter                      = var.filter
  unique_writer_identity = true

  bigquery_options {
    use_partitioned_tables = true
  }
}
```

As it can be seen, this code is parametrized, as many BigQuery log sinks are used across the organization. To create a new sink, a new module has to be created that extends this base code:

```
module "unity_services_gateway_metrics_log_sink" {
  source = "bigquery-logsink"

  dataset_id = "unity_services_gateway_metrics"
  sink_name  = "unity-services-gateway-metrics-stg-bigquery-log-
   sink"
  filter      = "resource.type=\"k8s_container\" AND (labels.k8s-
   pod/app=\"unity-services-gateway-public\" OR labels.k8s-pod/
   app=\"unity-services-gateway-internal\") AND jsonPayload.key
   =\"METRICS\""
}
```

This module extends the original log sink configuration, specifying a name for the log sink and the identifier for the BigQuery table where the log is going to be stored. One noticeable thing is that a filter is specified for distinguishing the logs containing the expected metrics. That is done by filtering only the records from the `unity-services-gateway-internal` or `unity-services-gateway-public` app, including the `"METRICS"` identifier.

5.3 Error visualization dashboard

This section covers the creation of a visualization tool for the high-level errors generated by the Unity Services Gateway and, therefore, how **Q3** is solved. It is composed of three subsections; the first one explains the integration tool used to connect Grafana with BigQuery, the second one explains the panels created for the dashboard, and the final one describes how the persistence of the continuous deployment had been done for it.

5.3.1 BigQuery and Grafana connection

One of the main characteristics of Grafana is that it is agnostic to the storage technology that is being used for the time series data that is going to be shown on the generated plots. This means that for adding different data sources, a connection has to be established between Grafana and the selected data storage technology, which in the case of this prototype is BigQuery. In that regard, two steps had to be followed for establishing that connection; the first one consists of installing the official Grafana plugin (`bigquery-grafana` [4]) on the current instance of the company, and the second step is to add BigQuery as a data source so Grafana can access its data by using SQL statements.

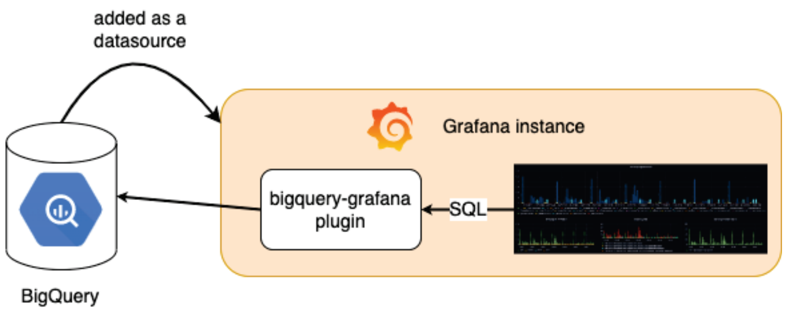


Figure 15: Diagram showing the connection between Grafana and Big Query

In order to fulfill the first step, thus installing the plugin on the Grafana instance, the procedure consists of adding the URL and the variable that sets how the plugin should be loaded on the Helm chart of the Grafana instance. For doing that, the file `values.yaml` has to be modified in two sections, `plugins` and `grafana.ini`, adding the following values:

```
 plugins:
    - https://github.com/doitintl/bigquery-grafana/releases/download/2.0.3/doitintl-bigquery-datasource-2.0.3.zip;doit-bigquery-datasource

 grafana.ini:
   plugins:
     allow_loading_unsigned_plugins: doitintl-bigquery-datasource
```

On the **plugins** section, the URL of the official repository is specified alongside the desired version, so when loaded, the Grafana application will know where to download the plugin from. Additionally, the `grafana.ini` section specifies the name of the plugin to be loaded, in this case, `doitintl-bigquery-datasource`. After this configuration is applied and the Helm application is deployed, the plugin should be accessible from the Grafana instance.

The second step for establishing a connection between Grafana and BigQuery is adding it as a data source. As the infrastructure management of the organization is using Terraform, adding a new data source for Grafana consists of adding the already existent BigQuery table as a Terraform resource [19].

There are two steps for adding this resource on Terraform; first, the different configurations for BigQuery, are defined as Terraform locals on the terraform data sources configuration file. The objective is to avoid repetition on the creation of new resources while simplifying the addition of a new BigQuery data source (or even other types of data sources, such as Prometheus) to the infrastructure. The definition of the locals is as follows:

```
locals {
  // some other datasources ...
  bigquery = {
    // some other BigQuery datasources ...
    "bigquery-unity-services-foundation-prd" = {
      "project"         = "gateway-metrics-ds"
      "service_account" = "gateway-metrics@iam.gserviceaccount.com"
    }
  }
}
```

After the local variables are added is time to create the BigQuery data source resource, which is specified as follows:

```
resource "grafana_data_source" "bigquery" {
 for_each = local.bigquery

 type = "doitintl-bigquery-datasource"
 name = each.key
```

```

  json_data {
      token_uri = "https://oauth2.googleapis.com/token"
      authentication_type = "jwt"
      default_project = each.value.project
      client_email = each.value.service_account
  }
}
```

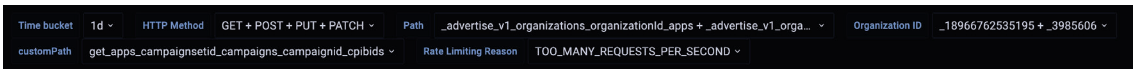
The resource generation is generalized for each data source specified as locals (using a `for_each`). Apart from that, there are two schema fields, `type`, and `name`. The first one defines the data source name extracted from the key of the locale, and the second one specifies the type determined by the Grafana plugin previously installed. Additionally, one meta-argument [18] (`json_data`) defines the data used when the current resource is created once the Terraform configuration is applied.

For managing the access to the BigQuery data source, the official JSON configuration accepts a `client_mail` field to check the access rights. Nevertheless, this is matched to the `service_account` field on the locale definition; this happens because the organization uses Role-Based Access Control to check the required permissions. It is relevant to mention that there is an additional process for creating a new service account for granting reading rights to the BigQuery data source, which is also done by creating another Terraform resource.

5.3.2 Grafana dashboard

In this section, it is going to be explained the handcrafted Grafana dashboard used for the visualization of the high-level errors that the gateway emits. This dashboard's objective is to effectively communicate those errors with the relevant stakeholders and establish the optimal way for visually representing those and consequently answering **Q3**.

One of the essential parts of this dashboard is the variables that will control the information that will be displayed on the panels. Variables work as follows: depending on the values selected, the events that match those values will be visualized on the boards. On this dashboard, six different variables are created (Figure 16).



Time bucket: 1d
HTTP Method: GET + POST + PUT + PATCH
Path: _advertise_v1_organizations_organizationId_apps + _advertise_v1_orga...
Organization ID: _18966762535195 + _3985606
customPath: get_apps_campaignsetid_campaigns_campaignid_cpibids
Rate Limiting Reason: TOO_MANY_REQUESTS_PER_SECOND

Figure 16: Overview of the six Grafana variables created

- **Time interval**: this variable is not explicitly created for this dashboard, but it is a default variable on Grafana. It defines the time range that will be selected for showing the time series data.
- **Time bucket**: It can be more understandable to show bucketed data instead of individual events when representing plots. This variable defines which is the time interval considered for those buckets.
- **HTTP method**: this variable defines the HTTP method of the gateway event that will be shown on the visualizations.

- **Path**: The path of the resource consumed which caused the gateway event to be triggered.
- **Organization ID**: The ID of the organization that triggered the gateway event.
- **Custom path**: this is a custom string that combines HTTP method and path, and it is used for visualization purposes.
- **Rate Limit reason**: it is essential to mention that at the moment, this dashboard is only visualizing rate limit events, even though it can be easily extended in the future. This variable represents which of the rate limit reasons is surpassed on the considered event.

It would not have been an extensible approach to manually insert the possible values for those variables, mainly because some of them, like the organization ID, could contain hundreds of possible values. In that regard, SQL was used for setting the possible values for them. The general structure for the SQL queries is the following:

```sql
SELECT DISTINCT jsonPayload.variable_name
FROM ‘gateway_metrics_ds‘

-- delimiter of the time range selected
WHERE ‘timestamp‘ BETWEEN TIMESTAMP_MILLIS($__from) and
   TIMESTAMP_MILLIS($__to)

-- set of common where clauses for variables
AND jsonPayload.source = "RATELIMITER"
AND jsonPayload.api_type = "public"
AND jsonPayload.api_namespace="advertise"

-- aditional WHERE for previous variables
-- e.g. for path variable
AND jsonPayload.http_method IN ($http_method)

ORDER BY jsonPayload.variable
```

On that query, the `SELECT DISTINCT` clause generates a list of possible values that the variables section of the Grafana dashboard can interpret. Furthermore, the variables are concatenated from left to right, so an extra `WHERE` clause is added in each query to set the previous variables' values.

In continuation with the explanation of the dashboard, the different panels that were created are going to be reviewed. The Grafana dashboard is divided into separate rows, and for each row, there is going to be a set of panels that share some characteristics.

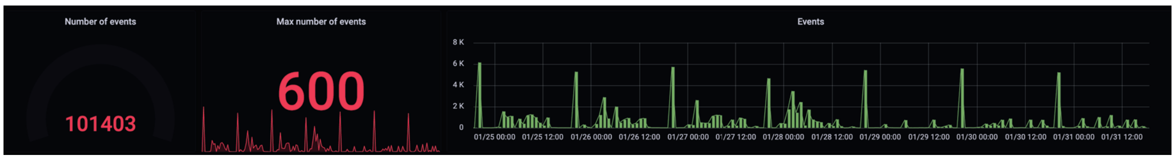
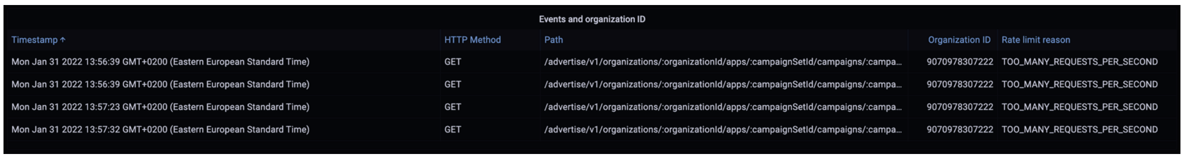


Figure 17: First row of the Grafana dashboard

The panels of the first row (Figure 17) give general information about the number of events generated by the gateway. The left panel is a gauge type of plot that shows the total number of events considering all the selected variables. The central panel is a stat type of plot that offers the most significant spike in events, showing the maximum number registered on the specified bucket. Finally, the left one shows the total number of events disposed by the selected bucked on the chosen time range. To gather the data for those panels the same SQL statement is shared across them:

```sql
SELECT
  $__timeGroup(timestamp, $time_bucket),
  count(*) AS n_events
FROM
  `gateway_metrics_ds`
WHERE
  $__timeFilter(timestamp)
  -- ...
  -- all the additional WHERE clauses for each variable
GROUP BY 1
```



Events and organization ID

Timestamp ↑	HTTP Method	Path	Organization ID	Rate limit reason
Mon Jan 31 2022 13:56:39 GMT+0200 (Eastern European Standard Time)	GET	/advertise/v1/organizations/:organizationId/apps/:campaignSetId/campaigns/:campa...	9070978307222	TOO_MANY_REQUESTS_PER_SECOND
Mon Jan 31 2022 13:56:39 GMT+0200 (Eastern European Standard Time)	GET	/advertise/v1/organizations/:organizationId/apps/:campaignSetId/campaigns/:campa...	9070978307222	TOO_MANY_REQUESTS_PER_SECOND
Mon Jan 31 2022 13:57:23 GMT+0200 (Eastern European Standard Time)	GET	/advertise/v1/organizations/:organizationId/apps/:campaignSetId/campaigns/:campa...	9070978307222	TOO_MANY_REQUESTS_PER_SECOND
Mon Jan 31 2022 13:57:32 GMT+0200 (Eastern European Standard Time)	GET	/advertise/v1/organizations/:organizationId/apps/:campaignSetId/campaigns/:campa...	9070978307222	TOO_MANY_REQUESTS_PER_SECOND

Figure 18: Panel of the dashboard showing the detailed information of the events

The second row (Figure 18) is a table that offers more detailed information about the events in the selected period. The five columns of the table are timestamp, HTTP Method, path, organization ID, and rate limit reason. The primary purpose of this table is to offer a more detailed view that would make the identification of specific events faster.

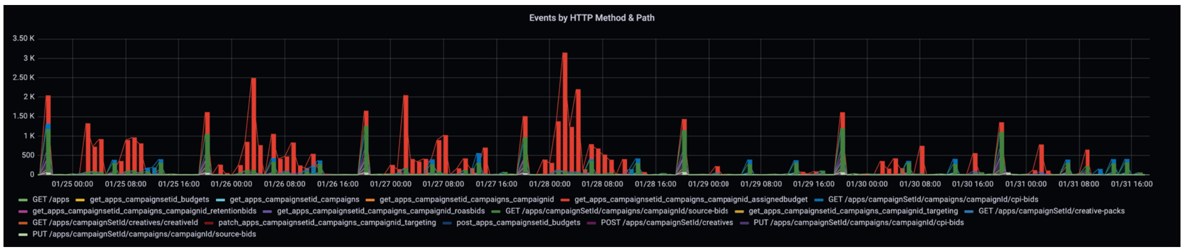


Figure 19: Panel showing the time series layout of the combination of HTTP method and path

The third row (Figure 19) contains a time series panel showing the combination of HTTP method and path that triggered the gateway event. Previously, it was stated that the variable **Custom path** combines HTTP method and path into a single variable, and the primary purpose is to build this panel. The SQL statement to gather the data is the following:

```sql
select *
from
(
  SELECT
```

```
    $__timeGroup(timestamp, $time_bucket),
    jsonPayload.custom_path
  FROM
    `gateway_metrics_ds`
  WHERE
    $__timeFilter(timestamp)
    -- ...
    -- all the additional WHERE clauses for each variable
) d
pivot
(
  count(custom_path)
  for custom_path in (
    $custom_path
  )
) piv
order by time
```

As can be seen, the PIVOT clause transforms the different rows into columns and builds the time series data displayed on the plot. Unfortunately, in this process, the two variables can not be combined, so this is why a variable that contains both values has to be created.

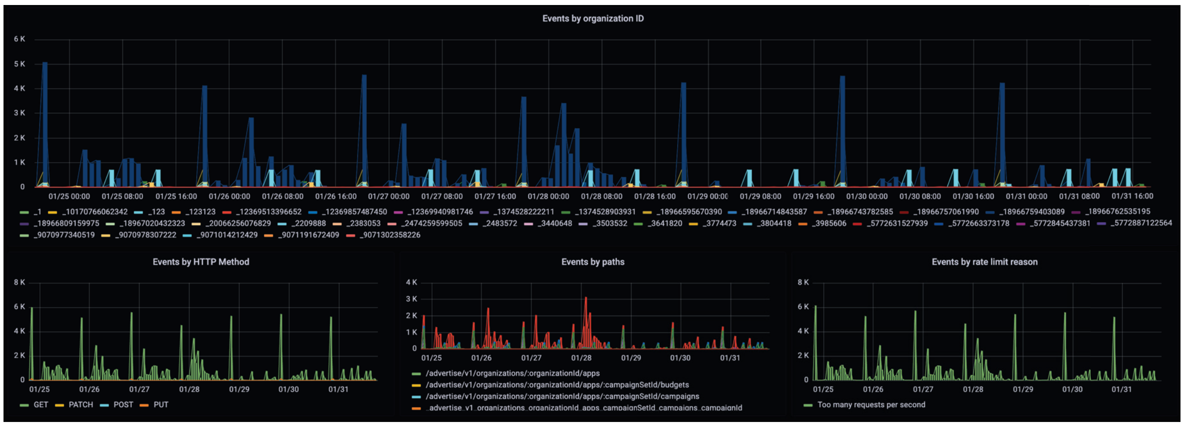


Figure 20: Panel showing the time series layout divided into the different possible values of the variables

Finally, the fourth and fifth rows (Figure 20) contain different panels showing the distribution of previously explained variables, like the events divided by other Organization IDs, HTTP Method, or path. One thing to notice is that the last panel shows the events by different rate limit reasons, and this is the only panel specific to rate limit events. The example SQL statement used for building those panels is the one used for the variable rate_limit_reason, and it is equivalent to the others, but only changing the variable name.

```
select *
from
(
  SELECT
```

```
    $__timeGroup(timestamp, $time_bucket),
    jsonPayload.rate_limit_reason
  FROM
    `gateway_metrics_ds`
  WHERE
    $__timeFilter(timestamp)
    -- ...
    -- all the additional WHERE clauses for each variable
) d
pivot
(
  count(rate_limit_reason)
  for rate_limit_reason in (
    $rate_limiting_reason
  )
) piv
order by time
```

Finally, it is essential to notice that the extensibility of these panels is straightforward. If a new field is introduced on the gateway event (Section 5.2.1), the only change on the dashboard side would be to duplicate one of the last panels and change the variable name on the SQL statement.

5.3.3 Grafana dashboard persistence

In this last section, it is going to be explained how the Grafana dashboard is persisted and correctly versioned. The objective is to have a way to recover the dashboard if something happens and not only rely on click-opts to create and save it. Additionally, the dashboard will be stored on a version control system (VSC) so it can be versioned; thus, team members can review its new changes before deploying it.

Once the dashboard is created, the next step is to generate a JSON file in the Grafana format [9] through the UI. That JSON Document will be persisted in the private repository of the project. For doing that, a pull request (PR) has to be created containing the copied file under the `monitoring/dashboard` directory. That particular file location is chosen because a company-wide automation deploys the dashboards located under that folder for each repository.

After this first version is created, future changes or maintenance will follow this procedure: import the JSON file into Grafana, make the changes that are required, export the new JSON file replacing the old one in the repository, and finally create a new pull request (PR) that explains the differences and that the team that owns the service has to review before being merged and deployed.

6 Evaluation

This section will cover the evaluation of the proposed framework. Two sub-sections are introduced for that purpose; the first will cover the internal evaluation techniques used to verify that it fulfilled the expectations; the second will compare the proposed framework against two of the most relevant tools already present in the industry.

6.1 Internal evaluation of the proposed framework

This section will cover the different steps used internally for verifying the proper functioning of the proposed framework, not only from the technical point of view on the different phases but also from the expected requirements point of view.

6.1.1 Event gathering phase evaluation

This first subsection covers the evaluation of the event gathering phase, which was initially introduced in Section 5.2.4. As explained before, this phase consisted of creating a wrapper logging function on the Unity Services Gateway codebase that will construct the event with the expected format, emitted to BigQuery afterward.

For evaluating this logging function and correctly testing if the event mapping was done as expected, the selected methodology was doing a battery of unit tests that will be part of a CI pipeline that ensures the correct functioning of the service. The chosen library for this purpose was Jest, as it is already used on other parts of the service. The complete code of the test can be observed in Appendix B.

6.1.2 Data pipeline, storage and visualization dashboard evaluation

The data pipeline, data storage, and error visualization dashboard were evaluated in several steps. One of the main characteristics of this evaluation is that for having a safe sandbox where to do testing for the development of this prototype and in the future, two different environments, staging, and production, were created. Production will contain the data and instances that will be delivered to the clients, and staging will be an exact clone where testing procedures should be executed.

One of the first relevant tests was if the data was stored correctly in BigQuery. The correctness criteria that was considered is that if an error is generated on purpose on the gateway side it should be able to flow across all the dependency chain and be stored on BigQuery and be visualized in Grafana. For doing that, a script that generated fake events was placed and connected to the data store in staging. That script inserted 10.000 events with the structure introduced in Section 5.2.1 on the staging instance of the dataset. This method allowed us to test that the event´s format was correct, and also, the fake data introduced can be used for testing the dashboards visualization afterward.

Finally, for evaluating the visualization dashboard, there were two instances of it created in Grafana. One was extracting the data from the production database and the other from the staging database. The manual test done in that regard was checking that the data inserted before on the staging data source could be visualized as expected on the staging dashboard. Apart from that, the query structure was confronted with the BigQuery console results.

6.1.3 End to end evaluation

The end-to-end evaluation consisted of deliberately generating error events on the Gateway service to check if there are correctly stored on the data source and visualized in the dashboard afterward. This step can only be done after deploying the event-gathering code on the staging environment and applying the Terraform configuration for the log sinks and Grafana.

As it was stated in Section 5.2.1, the high-level events will only include rate-limiting in this first iteration of the framework. In that sense, to test if the errors are correctly stored and visualized, they must first be generated. A new bash script was created that will call different endpoints in a loop a certain number of times over the expected quota, thus causing the expected errors. The script can be observed in Appendix C.

After creating and calling that script, the resulting events were observed on BigQuery and correctly visualized in the dashboard; thus, the framework could be safely deployed in the production environment.

6.2 Comparison with state-of-the-art gateway frameworks

In this section, the proposed framework will be compared to the two integral gateway solutions that are standards in the industry: The Gateway by Kong technologies and the Zuul gateway by Netflix, already introduced in Section 3.3. Those two examples are out-of-the-box integral gateway solutions, so they present more functionalities than the proposed framework; therefore, they will be compared by the expected criteria exposed in Section 4.4. The primary purpose of this comparison is to evaluate if the industry solutions already fulfill these criteria and to assess this proposed framework in conjunction with the Unity Services Gateway.

6.2.1 Frameworks comparison

The three solutions will be compared using four criteria to determine if the expected outcomes defined in Section 4.4 are achieved.

C1 - High-level responsibilities matching to individual routes

In this criterion, the frameworks will be evaluated concerning if they have a way to match the high-level responsibilities that they have to the routes that the gateway is exposing. The comparison is the following:

- **Kong**: This gateway has a minimalistic approach when defining the responsibilities it will take, as it is plugin-based. The plugins must be installed per different instances, and many of them are premium features. There are two options for defining the responsibilities that each route will take: by using the administrator UI, relying on click-opts for introducing the data, or doing a REST API call once the service is deployed.
- **Zuul**: There is a clear definition of the responsibilities on this gateway, but matching them to the routes is done by using code, as the routes are defined using Java classes.
- **Proposed framework**: The routes definition format is explained in Section 5.1.

As it can be seen, the only gateway that provides file-based configuration is the proposed framework. It might seem a disadvantage as the format is more straightforward if done through a UI. Nevertheless, using a file adds the option of introducing new changes to the structure and CI/CD capabilities.

C2 - Error format definition

This criterion refers to the possibility of adding enriched fields to the events gathered on the gateway side, defining our format. The comparison is the following:

- **Kong**: As this framework is an out-of-the-box solution, the time needed for the configuration is effortless, but the configuration possibilities are limited. In this case, there is no way to enrich the error fields and emit them to a secondary stream.
- **Zuul**: This framework allows error configuration and custom handling by catching the error on the codebase and adding an extra mapping and emission step.
- **Proposed framework**: Events are defined with the relevant fields to persist and be visualized afterward (Section 5.2.1) and gathered on the codebase (Section 5.2.4).

C3 - Data storage selection

This criterion refers to the existence of data pipelines and data storage technology for the persistence of the event that will ensure the proper visualization and discovery afterward.

- **Kong**: This solution provides Cassandra and PostgreSQL for storing rate-limiting and authentication events that the gateway emits.
- **Zuul**: This framework does not offer any custom integration to store data events. As it is a library module, all the integration has to be custom-built, which can be equivalent effort as the proposed framework, but in a language (Java) that does not fit directly on the company's technical stack.
- **Proposed framework**: Events are stored on BigQuery, and the data pipeline is defined as a GCP sink (Section 5.2.5).

C4 - Visualization of high-level errors

This criterion sets the existence of a visualization tool that allows the stakeholders to visualize the high-level error events of the gateway simply. In that regard, of the compared frameworks, only the proposed solution includes this visualization tool, explained in Section 5.3.

6.2.2 Frameworks comparison conclusions

	C1	C2	C3	C4
Kong	Yes - with UI	No	Yes - Cassandra and PostgreSQL	No
Zuul	Yes - with code	Yes	No	No
Proposed solution	Yes - with file	Yes	Yes - BigQuery	Yes

Table 2: Framework comparison table

Table 2 summarizes the comparison between the different frameworks. The conclusion that can be extracted is that even though Kong or Zuul are considered industry standards and are very competent products, they do not provide an automatic solution for the visualization and storing of the high-level events of the gateway.

7 Conclusions

In recent years, there has been an increase in the adoption of microservices architectures by extensive and well-established companies and small startups. On many of them, the amount and complexity of their architecture are so significant that they had to adopt patterns like DOMA, creating a gateway that allows them to simplify the operations that they had spread across many microservices. This thesis aims to offer a framework for developing and monitoring the high-level errors that those gateways are producing and giving procedures on how to increase their visibility of them across different teams.

From the point of view of the expected outcomes, especially from the stakeholder's point of view, it can be stated that this thesis matches the expectations. Before this research thesis and the implementation of the prototype, they were not able to easily access the gateway's information. After it, they could fulfill all the requirements gathered in Section 4. This represents a critical operational benefit for the company.

Apart from that, the work developed in this thesis gave valuable insights that can be applied to any organization facing the same challenges that Unity Technologies. But it is remarkable to mention that this thesis is done within a multinational organization. Thus, the knowledge might not be applied in the same way for organizations of different sizes. This is why these conclusions might be divided to give insights depending on the company's size, from the technical point of view.

Small number of microservices (1 - 20)

In this type of organization, the main goal is to develop the business's value proposition and give worth to the final customers. In those organizations, the number of employees is usually small, so there are not enough resources for implementing internal tools like the one proposed in this thesis.

The software architecture is usually monolithic or just a few microservices on these businesses. Therefore, as they are not running a DOMA, they are not the target of this project. Nevertheless, in the initial stages of software design is always beneficial to take into consideration that the architecture might change and scale in the future, and they might find the necessity of adopting a gateway after that initial stage.

Medium number of microservices (20 - 50)

In the case of a software-based organization that is running from 20 to 50 microservices, it is expected that the need for a standard gateway for the microservices has already arisen.

In that case, the most crucial step is defining the format for determining the routes that the gateway will expose to the public and how to implement the high-level responsibilities. Considering this, one sensible approach will be to select one of the existing industry-standard solutions for the gateway (Section 6.2), as it will dramatically reduce the implementation and maintenance time. One possible solution to monitoring it will be to adapt the knowledge exposed in the technical solution of this thesis (Section 5) to the particular case of the gateway technology that they are running.

Big number of microservices (More than 50)

For companies or organizations with more than 50 microservices running, the most sensible approach will be to adopt a DOMA architecture while maintaining a gateway that will be the starting point of one or several domains. This is the case with Unity technologies or Uber.

This thesis is mainly focused on this type of organizations, as different teams probably depend on others, working in various locations and times zones. In that case, visibility of the business logic is crucial, and monitoring the high-level gateway errors becomes very significant.

7.1 Future work

Different improvement ideas or future work that can be done on this topic are introduced in this section.

The first one is about the different types of errors that were mapped. As it was stated in section 5.2.1, for this project, only rate-limiting responsibility is considered for the monitoring; this is because it was the error that project managers and teams were more worried about. Nevertheless, other types of errors can be handled, and that can be done by extending the error type and using the custom function introduced in section 5.2.1.

The second improvement idea will be to create a gateway composable logic tool. Having the standard defined in section 5.1, the next step can be to create a solution that injects the handling of those high-level responsibilities on the gateway at execution time to minimize the effort for defining new routes and resources.

Finally, the last improvement idea will be to pack this knowledge into a fully-extendable tool that can be used in standard gateway solutions like Kong. For doing that, some pieces of this software should be re-written to ensure compatibility with the gateway solution that it is intended to be extended.

References

[1] Amplitude. https://amplitude.com/. Accessed: 2022-01-20.

[2] The architecture of uber's api gateway. https://eng.uber.com/architecture-api-gateway. Accessed: 2021-08-08.

[3] BigQuery. https://cloud.google.com/bigquery. Accessed: 2022-01-20.

[4] BigQuery DataSource for Grafana. https://github.com/doitintl/bigquery-grafana. Accessed: 2022-01-30.

[5] Cloud SQL for PostgreSQL pricing. https://cloud.google.com/sql/docs/postgres/pricing. Accessed: 2022-01-20.

[6] Configure and manage sinks. https://cloud.google.com/logging. Accessed: 2022-02-07.

[7] Configure and manage sinks. https://cloud.google.com/logging/docs/export/configure_export_v2#api. Accessed: 2022-02-02.

[8] Configure and manage sinks. https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/logging_project_sink. Accessed: 2022-02-02.

[9] Dashboard JSON model. https://grafana.com/docs/grafana/latest/dashboards/json-model/. Accessed: 2022-02-01.

[10] How to Estimate Google BigQuery Pricing. https://chartio.com/resources/tutorials/how-to-estimate-google-bigquery-pricing/. Accessed: 2022-01-19.

[11] Introducing domain-oriented microservice architecture. https://eng.uber.com/microservice-architecture. Accessed: 2021-08-08.

[12] Kong gateway. https://konghq.com/faqs/. Accessed: 2022-02-24.

[13] Microservices adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/. Accessed: 2021-09-03.

[14] Microservices logging best-practices. https://www.scalyr.com/blog/microservices-logging-best-practices. Accessed: 2021-09-03.

[15] Official BigQuery pricing. https://cloud.google.com/bigquery/pricing. Accessed: 2022-01-19.

[16] Open Sourcing Zuul 2. https://netflixtechblog.com/open-sourcing-zuul-2-82ea476cb2b3. Accessed: 2022-02-24.

[17] PostgreSQL. https://www.postgresql.org/. Accessed: 2022-01-20.

[18] Terraform meta arguments. https://www.terraform.io/language/meta-arguments/lifecycle. Accessed: 2022-01-30.

[19] Terraform resources. https://www.terraform.io/language/resources. Accessed: 2022-01-30.

[20] The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749. Accessed: 2022-01-04.

[21] Unity Technologies website. https://unity.com/. Accessed: 2021-11-17.

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

[23] Saad Alahmari, Ed Zaluska, and David De Roure. A service identification framework for legacy system migration into soa. In *2010 IEEE International Conference on Services Computing*, pages 614–617. IEEE, 2010.

[24] Philip Chu. iad: Banner ads and interstitial ads. In *Learn Unity 4 for iOS Game Development*, pages 439–451. Springer, 2013.

[25] Gennaro Cuofano. How Does Unity Work And Make Money? Unity Business Model Explained. https://fourweekmba.com/unity-business-model/. Accessed: 2021-11-17.

[26] Ole-Johan Dahl and Charles Antony Richard Hoare. Chapter iii: Hierarchical program structures. In *Structured programming*, pages 175–220. 1972.

[27] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE, 2019.

[28] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[29] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 291–304, 2013.

[30] M. Thangavelu et al. The Architecture of Uber's API gateway. https://eng.uber.com/architecture-api-gateway/. Accessed: 2022-01-01.

[31] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[32] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech house, 2003.

[33] Avi Goldfarb and Catherine E Tucker. Online advertising, behavioral targeting, and privacy. *Communications of the ACM*, 54(5):25–27, 2011.

[34] A. Greenberg. An Absurdly Basic Bug Let Anyone Grab All of Parler's Data.

[35] Monika Gupta, Atri Mandal, Gargi Dasgupta, and Alexander Serebrenik. Runtime monitoring in continuous deployment by differencing execution behavior model. In *International Conference on Service-Oriented Computing*, pages 812–827. Springer, 2018.

[36] Louisa Ha. Online advertising research in advertising journals: A review. *Journal of Current Issues & Research in Advertising*, 30(1):31–48, 2008.

[37] John Haas. A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*, 2014.

[38] David C Hay. *Requirements analysis: from business views to architecture*. Prentice Hall Professional, 2003.

[39] Trong Thang Hoang, Minh Thong Tao, and Phu Hiep Au. *Research and implementation of monitoring systems Prometheus and Grafana*. PhD thesis, FPTU HCM, 2020.

[40] Ted Hunter and Steven Porter. *Google Cloud Platform for developers: build highly scalable cloud solutions with the power of Google Cloud Platform*. Packt Publishing Ltd, 2018.

[41] IEEE. IEEE Standard Glossary of Software Engineering Terminology. https://ieeexplore.ieee.org/document/159342. Accessed: 2022-01-13.

[42] M. Fowler J. Lewis. Microservices a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. Accessed: 2021-12-31.

[43] Rutvij H Jhaveri, Sankita J Patel, and Devesh C Jinwala. Dos attacks in mobile ad hoc networks: A survey. In *2012 second international conference on advanced computing & communication technologies*, pages 535–541. IEEE, 2012.

[44] Haan Johng, Doohwan Kim, Tom Hill, and Lawrence Chung. Estimating the performance of cloud-based systems using benchmarking and simulation in a complementary manner. In *International Conference on Service-Oriented Computing*, pages 576–591. Springer, 2018.

[45] Rudolf E Kalman. On the general theory of control systems. In *Proceedings First International Conference on Automatic Control, Moscow, USSR*, pages 481–492, 1960.

[46] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[47] JinJin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*, pages 3–20. Springer, 2018.

[48] James Martin. *Managing the data base environment*. Prentice Hall PTR, 1983.

[49] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[50] Ronald K Mitchell, Bradley R Agle, and Donna J Wood. Toward a theory of stakeholder identification and salience: Defining the principle of who and what really counts. *Academy of management review*, 22(4):853–886, 1997.

[51] Bruce Jay Nelson. *Remote procedure call*. Carnegie Mellon University, 1981.

[52] Benjamin Nicoll and Brendan Keogh. The unity game engine and the circuits of cultural software. In *The Unity game engine and the circuits of cultural software*, pages 1–21. Springer, 2019.

[53] Sina Niedermaier, Falko Koetter, Andreas Freymann, and Stefan Wagner. On observability and monitoring of distributed systems–an industry interview study. In *International Conference on Service-Oriented Computing*, pages 36–52. Springer, 2019.

[54] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52, 1992.

[55] Rodolfo Picoreti, Alexandre Pereira do Carmo, Felippe Mendonca de Queiroz, Anilton Salles Garcia, Raquel Frizera Vassallo, and Dimitra Simeonidou. Multilevel observability in cloud orchestration. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 776–784. IEEE, 2018.

[56] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review*, 37(4):337–348, 2007.

[57] Joyce Reynolds and Jonathan Postel. The request for comments reference guide. *Request for Comments*, 1000, 1987.

[58] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[59] Steven Schmeiser. Online advertising networks. *Manuscript*, 2016.

[60] Ken Schwaber. Scrum development process. In *Business object design and implementation*, pages 117–134. Springer, 1997.

[61] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pages 1–6. IEEE, 2017.

[62] Alan Snyder. The essence of objects: Common concepts and terminology. *IEEE Software*, 1993.

[63] David Sprott and Lawrence Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17, 2004.

[64] Pyda Srisuresh, Bryan Ford, and Dan Kegel. State of peer-to-peer (p2p) communication across network address translators (nats). *Internet Engineering Task Force, Request For Comments*, 2008.

[65] Chang-ai Sun, Meng Li, Jingting Jia, and Jun Han. Constraint-based model-driven testing of web services for behavior conformance. In *International Conference on Service-Oriented Computing*, pages 543–559. Springer, 2018.

[66] Dean Takahashi. Unity acquires Applifier as mobile game replays take off. https://venturebeat.com/2014/03/13/unity-acquires-applifier-as-mobile-game-replays-take-off/. Accessed: 2021-11-30.

[67] Loggly team. Observability vs. Monitoring – What's the Difference? https://www.loggly.com/blog/observability-vs-monitoring-whats-the-difference/. Accessed: 2022-01-13.

[68] Ranjit Singh Thakurratan. *Google Cloud Platform Administration: Design highly available, scalable, and secure cloud solutions on GCP*. Packt Publishing Ltd, 2018.

[69] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[70] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[71] Ervin Varga. Versioning rest apis. In *Creating Maintainable APIs*, pages 109–118. Springer, 2016.

[72] Ville Vesterinen. Cross-Promotion Network Applifier Hit 55 Million Users In 4 Months. https://arcticstartup.com/cross-promotion-network-applifier-hit-55-million-users-in-4-months/. Accessed: 2021-11-30.

[73] Barbara Von Halle and Larry Goldberg. *The decision model: a business logic framework linking business and technology*. CRC Press, 2009.

[74] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86, 1993.

[75] Yong Yang, Long Wang, Jing Gu, and Ying Li. Transparently capturing execution path of service/job request processing. In *International Conference on Service-Oriented Computing*, pages 879–887. Springer, 2018.

A Appendix 1

This appendix contains the code for the event gathering on the Unity Services Gateway:

```
const { logger } = require('@unity/node-monitoring');
function log(req,
  {
    quota,
    source,
    type,
    apiVersion,
    apiNamespace,
    rateLimitReason,
    group,
    resourceId,
  },
) {
  const organizationId = req.params?.organizationId ?? null;
  const requestId = req?.context?.id ?? '';
  const apiType = process.env.API_MODE ?? '';
  const event = {
    key: 'METRICS',
    path: req.route.path,
    url: req.url,
    httpMethod: req.method,
    customPath: `${req.method}_${req.route.path}`,
    source,
    type,
    apiVersion,
    apiNamespace,
    rateLimitReason,
    group,
    resourceId,
    apiType,
    requestId,
    organizationId,
    quota,
  };

  logger.log(event);
}
module.exports = { log };
```

B Appendix 2

This appendix contains the code for the test case written in JavaScript using the framework Jest for evaluating the event gathering function on the Unity Services Gateway (Section 6.1.1).

```
const { logger } = require('../../monitor');
const { log } = require('./metrics');

describe('metrics', () => {
  let req;

  beforeEach(() => {
    jest.clearAllMocks();
    jest.spyOn(logger, 'log');
    req = {
      route: {
        path: '/url/:organizationId',
      },
      url: '/url/123',
      method: 'get',
      params: {
        organizationId: 123,
      },
      context: {
        id: 'abc-123',
      },
    };
  });

  it('should call logger one time', async () => {
    log(req, {});
    expect(logger.log).toHaveBeenCalledTimes(1);
  });

  it('should call logger with correct data', async () => {
    log(req, {
      quota: 10,
      source: 'RATELIMITER',
      type: 'QUOTA_EXCEEDED',
      apiVersion: 1,
      apiNamespace: 'advertise',
      rateLimitReason: 'TOO_MANY_REQUESTS_PER_SECOND',
      group: 'group-123',
      resourceId: '*',
      customPath: 'A-01:{#\\.}a/b',
    });

    expect(logger.log).toHaveBeenCalledWith({
      key: 'METRICS',
      path: '/url/:organizationId',
      url: '/url/123',
      httpMethod: 'get',
      time: expect.any(String),
      source: 'RATELIMITER',
      type: 'QUOTA_EXCEEDED',
      apiVersion: 1,
      apiNamespace: 'advertise',
      rateLimitReason: 'TOO_MANY_REQUESTS_PER_SECOND',
      group: 'group-123',
      resourceId: '*',
      apiType: '',
      requestId: 'abc-123',
      organizationId: 123,
      quota: 10,
      severity: 'info',
      customPath: 'a01a_b',
    });
  });
});
```

Two test cases had been included, the first one tests if the Google Logger function had been called only once and the second one tests if the mapped event matches the expected structure.

C Appendix 3

This appendix contains the script used for triggering the rate limiting events on several endpoints, introduced on Section 6.1.3.

```
for i in {1..4010}
do
    curl --request GET \
   --url https://services.api.unity.com/advertise/v1/organizations
    /:organizationId/apps/ \
   --header 'Authorization: Basic <service-account>'

   curl --request GET \
   --url https://services.api.unity.com/advertise/v1/organizations
    /:organizationId/apps/:appId/campaigns \
   --header 'Authorization: Basic <service-account>'

   curl --request GET \
   --url https://services.api.unity.com/advertise/v1/organizations
    /:organizationId/apps/:appId/campaigns/:campaignId/budget \
   --header 'Authorization: Basic <service-account>'
done
```

Inside a loop that intends to exceed the established quota of 4000 calls per 30 minutes there are three calls to different endpoints. The concrete parameters for the endpoint are not included for privacy concerns.