Universidade de Aveiro
2022

Carlos Manuel
Basílio Oliveira

Arquitectura de Software Escalável para Sistemas de Apoio à Decisão para Entidades Gestoras de Água

Towards a scalable Software Architecture for Water Utilities' Decision Support Systems

Carlos Manuel Basílio Oliveira

Arquitectura de Software Escalável para Sistemas de Apoio à Decisão para Entidades Gestoras de Água

Towards a scalable Software Architecture for Water Utilities' Decision Support Systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor André Zúquete, auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor António Gil D'Orey Andrade Campos (co-orientador), Professor auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro.

o júri / the jury

presidente / president	**ABC** Professor Catedrático da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)
vogais / examiners committee	**DEF** Professor Catedrático da Universidade de Aveiro (orientador)
	GHI Professor associado da Universidade J (co-orientador)
	KLM Professor Catedrático da Universidade N

agradecimentos

Agradeço o apoio da minha família, amigos e colegas da SCUBIC, e ao prof. Zúquete pela paciência e disponibilidade estes últimos anos

acknowledgments

I wish to thank my family, friends and coworkers at SCUBIC for the support, as well as prof. Zúquete for the availability and patience through these past years

Palavras-chave

Água, Arquitectura de Software, Sistemas de Apoio à Decisão, Entidades Gestoras de Água

Resumo

O fornecimento de água às populações é um serviço de qualquer grande sociedade, desde o início da Civilização. Hoje em dia, enormes quantidades de água são fornecidas constantemente a residências e indústrias variadas utilizando motores eléctricos acopolados a bombas de água que consomem vastas quantidades de energia eléctrica. Com o recurso a tarifas de electricidade variáveis e dinâmicas, dados em tempo real de sensores nas empresas de fornecimento de água e a modelos da rede de distribuição de água, o software da SCUBIC consegue monitorizar e prever consumos de água e assim optimizar a operação destas bombas por forma a baixar os custos operacionais das empresas gestoras de água.
O software fornecido pela SCUBIC é um conjunto de serviços construídos numa fase embrionária da empresa que, por se manterem inalterados ao longo dos anos, não se adequam ao plano de negócios e aumento de requisitos por parte dos *stakeholders*. Daqui surge então a necessidade de construir uma nova arquitectura de software capaz de responder aos novos desafios numa indústria cada vez mais instrumentalizada e evoluída como a da Gestão de Água.
Recorrendo a métodos de engenharia de software, migração de arquitecturas de software e planeamento cuidadoso, sugere-se neste trabalho uma nova arquitectura de software baseada em micro-serviços e *serverless*.Esta arquitectura foi então avaliada de acordo com os índices ((indicar quais)) e comparada com a solução antiga. Após rever os resultados gerados pelos indicadores de performance, conclui-se que a migração foi um sucesso.

Keywords Key, word.

Abstract

Water Supply is a staple of all civilizations throughout History. Nowadays, huge amounts of water are constantly supplied to homes and businesses, requiring the use of electric pumps which consume vast amounts of electric energy.

By using variable and dynamic electric tariffs, multiple real-time sensor date from Water Utilities and Water Network Modelling, the SCUBIC software is able to monitor the water networks, predict water consumption and optimize pump operation allowing the Water Utilities to lower operational costs.

Built during an earlier phase of the company, the SCUBIC software is a monolithic amalgamation of services, full of compromises that cannot fulfill the latest requirements from the *stakeholders* and business plan.

Therefore, a need to build a more modular and scalable software architecture for this software becomes apparent. Using careful planning, software engineering knowledge and literature regarding software architecture migration, a new software architecture was implemented. Results from comparisons between the older and newer architectures prove that the migration was a success and complies with the requirements set at the beginning of the project.

Table of contents

Table of contents i

List of figures iii

List of tables v

List of abbreviations vii

1 Introduction 1

1.1 Water Supply Systems 1
1.2 Existing Decision Support System 1
1.3 Objectives 2
1.4 Structure of the Document 3

2 State-of-the-Art 5

2.1 Cloud Computing 5
2.1.1 Deployment models for cloud computing 5
2.2 Software-as-a-Service 6
2.2.1 Security 7
2.3 Software Engineering 7
2.3.1 Defining Requirements 8
2.4 Software Architecture 8
2.4.1 Service-Oriented Architecture 8
2.4.2 Modularity 9
2.4.3 Reusing Software Components 9
2.4.4 Cohesion and Coupling 9
2.4.5 Monolithic and Modular Monolithic 10
2.4.6 Microservices 11
2.5 DevOps 14
2.5.1 Deployment Frequency 14
2.5.2 Lead Time for Change 14
2.5.3 Time to Restore Service 14
2.5.4 Change Failure Rate 14

2.6 Observability 15
2.6.1 OpenTelemetry 15
2.6.2 Telemetry 16

3 Methodology **17**
3.1 The Application 17
3.2 The Old Architecture 18
3.2.1 Overview 19
3.2.2 Issues 23
3.2.3 Observability 29
3.3 Proposed New Architecture 30
3.3.1 Solving the Deployment issues 30
3.3.2 Solving the Scalability issues 32
3.3.3 Replaceability 34
3.3.4 Resiliency 34
3.3.5 Temporal Coupling 35
3.3.6 Implementation Coupling 36
3.3.7 Testing 36
3.3.8 New components 38
3.3.9 Workers 40
3.3.10 VPN and DNS 41
3.3.11 Observability 41
3.3.12 Finalized Proposed Version 41

4 Results and Discussion **45**
4.1 Scalability Improvements 45
4.1.1 Cost Rundown 45
4.1.2 Development and Deployment Issues 47
4.2 DevOps Improvements 48
4.3 Observability Improvements 48

5 Conclusion **49**
5.1 Final Considerations 49
5.2 Future Work 49

References **51**

Appendices **55**

A Appendix example **57**
A.1 A section example 57

B A second example of an appendix **59**

List of figures

1.1 DSS example. 2

2.1 Single-Process Monolith 10
2.2 Modular Monolith 11
2.3 Netflix Microservice Architecture Example 11
2.4 Example Microservice Architecture 12

3.1 AWS VPC Overview 19
3.2 Old Architecture's Containers 20
3.3 old-arch-nginx listing 22
3.4 Client CPU Usage Example 27
3.5 Initially Proposed Architecture 31
3.6 new-arch-scalable listing 33
3.7 new-arch-queues listing 35
3.8 Load Balancing Example 36
3.9 VPN Usage Example 37
3.10 Service Naming Scheme 38
3.11 Serverless Data Intake 39
3.12 Main VPC Diagram 42
3.13 Elastic File System 43

4.1 Infrastructure Cost Per Client 46

List of tables

4.1 Individual Cost of AWS Infrastructure used with the old architecture. Each Month corresponds to 30 days. 45
4.2 Individual Cost of AWS ECS Fargate Containers used in the new architecture. Each Month corresponds to 30 days. 46

List of abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CAPEX	Capital Expenditure
CI/CD	Continuous Integration/Continuous Deployment
CSV	Comma-Separated Values
CTO	Chief Technology Officer
DNS	Domain Name System
DSS	Decision Support System
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
EFS	Elastic File System
EIP	Elastic IP
ENI	Elastic Network Interface
FKM	Four Key Metrics
HTTPS	Secure Hypertext Transfer Protocol
KPI	Key Performance Index
NIST	National Institute of Standards and Technology
OPEX	Operational Expenditure
ORM	Object-Relational Mapper
OTel	OpenTelemetry
SaaS	Software-as-a-Service
SCADA	Supervisory Control And Data Acquisition
SDK	Software Development Kit
SFTP	SSH File Transfer Protocol
SOA	Service-Oriented Architecture
SRP	Single-responsibility Principle

SSH	Secure Shell
TLS	Transport Layer Security
vCPU	Virtual CPU
VPC	Virtual Private Cloud
VPN	Virtual Private Network(s)
VPS	Virtual Private Server
VSD	Variable-Frequency Drive
WSS	Water Supply Systems
WU	Water Utilities

Chapter 1

Introduction

1.1 Water Supply Systems

The water supply systems that are prevalent in modern society play a very important role in daily life, distributing water throughout the country from water reservoirs or water treatment plants to the citizen's houses and industries. These Water Supply Systems (WSS) can be quite complex and difficult to manage without proper processes that ensure the efficient operation of such networks including its environmental and economical sustainability. For this reason nowadays, the use of specialized software to aid operators or even automatically control the operation of these WSS is of uttermost importance. It must be highlighted that water has been a staple of all major human civilizations throughout History, from ancient roman aqueducts to the current era.

Moving large quantities of water through large WSS requires the use of large quantities of mechanical work, which in turn requires high levels of electric energy. With the ever-growing political, economic and environmental pressure to improve and optimize the use of energy, and with the current geopolitical issues, the access to energy is getting more expensive and regulated. This means that the need for the optimization of pumping operations to reduce costs and, potentially reduce the energy use as well, is growing within Water Utilities (WU).

1.2 Existing Decision Support System

In order to the WU's optimally operate their water pumps, a Decision Support System (DSS) is used by the WU's pump operators and/or by automatic Supervisory Control And Data Acquisition (SCADA) systems. Generally, this DSS is a web platform designed to suggest *which* pumps to operate, *when* to operate, for *how* long to operate and in some cases what *speed* their Variable-Frequency Drive (VSD)'s should operate, as shown in Figure 1.1

The existing software's architecture can be summarized as a "Monolithic Modular" software architecture (Newman, 2019). This architecture is composed of a set of Virtual

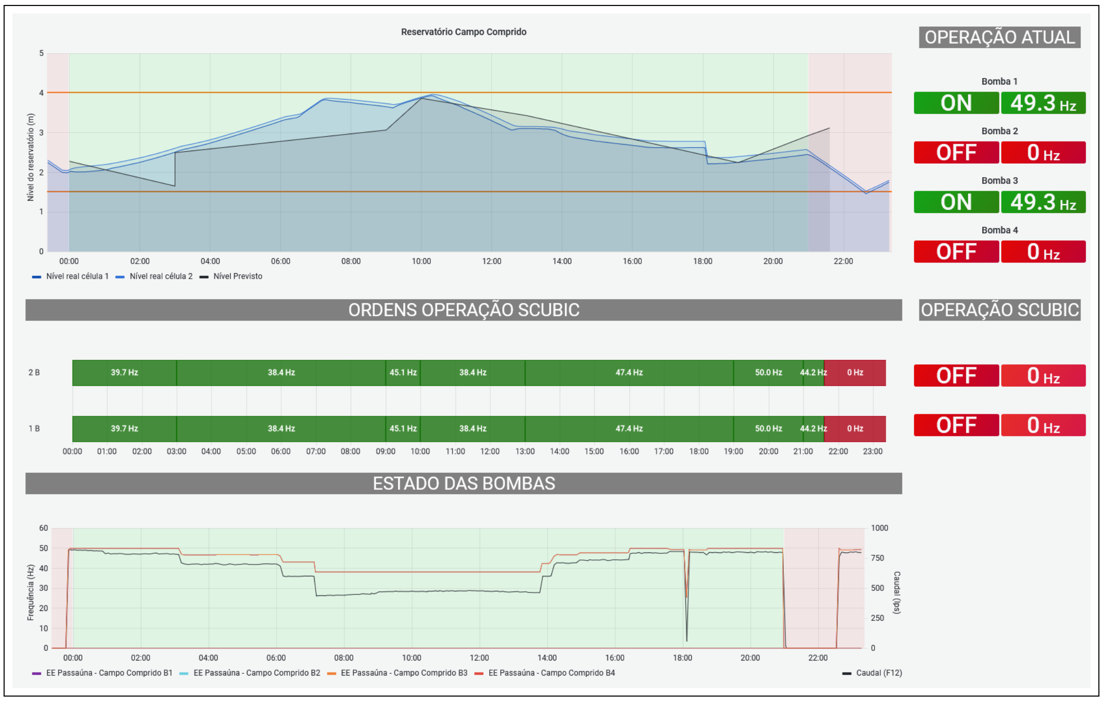

Figure 1.1: Example of a DSS interface from one of SCUBIC's[1] Clients.

Private Server (VPS), one for each Client, where a set of Docker containers enclose all the services needed for running the software for that Client. These services are also configured and developed separately, each in a different code repository. This fact results in an unsurmountable amount of *code drift* between the same services of the different clients. *Code drift* happens when, despite being based on the same code, the codebases for each Client follow different paths during software development. When there is a need to implement a new feature or fix a bug common to both codebases, these differences increase the amount of work. Apparently, this structure is not even remotely manageable for any software development team. On **??**, a complete analysis of this architecture is provided and explained in detail. [1]

1.3 Objectives

The main goal of this work is to make the migration from the old software architecture of the DSS to a more efficient, improved software, considering the requirements from the *stakeholders* while also improving the cost-performance ratio of the software without compromising the software's functionality. This new architecture improves the performance, reliability, resilience, security, scalability and observability in comparison to the old DSS Architecture. The new software architecture brings improvements not just for the software itself but also for the development team, allowing them to improve and maintain the software easier and faster than ever before. By reducing the amount of work and time

[1] https://scubic.tech

the software development team spends on each maintenance action or new functionality, it reduces cost to the software company as well. Infrastructure costs are also an important aspect of this new architecture, where the adoption of more modular and independent services means a more optimal use of compute resources, resulting in lowering such costs. This new architecture also improves the Observability of the entire system, allowing for quicker failure detection and to anticipate possible future problems with the system.

As such, the objectives can be summarized as three goals: Enable scalability of the software (through multi-tenancy), improve DevOps' Key Performance Index (KPI) and improve the Observability of the systems.

1.4 Structure of the Document

This document is composed by a total of X chapters.

In Chapter 1, the chapter presents the overall theme of this body of work. Firstly, some context is given about the overall theme of this body of work and the motivation behind it. Then, the objectives for dissertation are presented to the reader. Finally, at the end of the chapter, some information regarding the content of each chapter is presented.

In Chapter 2, a bibliographical analysis is presented, divided into three parts. Firstly, it's presented a summary of the state-of-the-art on software architecture, cloud-based software solutions, scalability and containerization of services. Secondly, some concepts regarding DevOps' origins and it's influence in today's software development paradigm are presented as well as what KPI are regarded as important for DevOps. Thirdly, Observability is studied and presented, exploring how it can improve software development in general and how it enables the developers and maintainers to gain insight into the system internal state. Additionally, some text regarding the general technologies used throughout the work is also analyzed whenever relevant to the topic in question.

Chapter 3 is divided into two sections. Firstly, a more detailed explanation of the old architecture and its inherent flaws is presented. These flaws, which end up showcasing the need for a new and improved software architecture, are related to the objectives established in section 1.3. In this section, it's explained how the old software architecture is flawed and has problems with scalability, poor DevOps performance and low-to-non-existent Observability. Then, in a second section, the author proposes a new software architecture that attempts to solve the problems aforementioned. In this later section, it's shown how the new architecture works, it's key components and given multiple diagrams that help explain said architecture. For each one of the flaws presented in the section before, it's presented how each aspect of the new architecture solves those flaws and the reasoning behind the choices that lead to this new architecture. The procedures taken, the challenges and decisions made throughout the implementation are shown and contextualized in this section.

Chapter 4 analyzes how the new architecture manages to achieve the objectives set in the introductory chapter of this document. Firstly, a cost-rundown report and simulated costs are shown that prove the cost-effectiveness improvement. Secondly, it shows how, by using the new architecture, the DevOps' KPIs have improved. Lastly, an overview of the result of implementing changes to the architecture to increase system observability and how it relates to better error detection and increases the system's behavior awareness.

Chapter 5 discusses the previous results and presents some conclusions from what has been demonstrated on previous chapters.

Furthermore, attached to this document, is an appendix that contains some extra results generated from the monitoring interface used internally to evaluate the new architecture.

Chapter 2

State-of-the-Art

2.1 Cloud Computing

Cloud Computing is a robust and scalable dynamic platform where configurable compute resources are made available as a service, over regular Internet access (Alnumay, 2020). It can also be understood, as the U.S. National Institute of Standards and Technology (NIST) indicates as: *(...) a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* (Mell and Grance, 2011).

According to the predictions in (IDC, 2021), *'by the end of 2021, 80% of enterprises will put a mechanism in place to shift to cloud-centric infrastructure and applications twice as fast as before the [Sars-Cov2] pandemic.'*

Incorrectly used as a synonym of on-demand computing, grid computing or even Software-as-a-Service (SaaS) (Kim, 2009), Cloud Computing has become prevalent nowadays in academic, household and business environments, from small business to large enterprises (Rezaei *et al.*, 2014) with multiple deployment models:

2.1.1 Deployment models for cloud computing

2.1.1.1 Private Cloud

A Private Cloud is managed by a single entity, within a single organization. The uses for Private Cloud can be for data privacy reasons, academic reasons, testing reasons or even to utilize existing in-house resources of an organization. This deployment model has the advantage of also allowing local data transfers, which are usually paid for when using other deployment models.

2.1.1.2 Community Cloud

A Community Cloud is a Private Cloud where several organizations democratically manage, construct, maintain and share the same cloud infrastructure. This allows for a more economically stable experience.

2.1.1.3 Public Cloud

In the Public Cloud, the dominant form of Cloud Computing (Dillon *et al.*, 2010), the users are the general public and the owner and maintainer of the underlying infrastructure and services is the cloud service provider. With this cloud, users are provided access to cloud computing services and don't have to worry about the infrastructure.

2.1.1.4 Virtual Private Cloud

A newer type of cloud deployment model has emerged in the last decade where users can experience a mixture of Public and Private Cloud. A Virtual Private Cloud (VPC) enables users to manage virtual infrastructure on top of public infrastructure. Cloud providers such as Amazon, Google and Microsoft are the main providers of these VPCs (Aljamal *et al.*, 2018), where users can stipulate the amount and configuration of resources like they were in a Private Cloud, ensuring low to non-existent data transfer limits and cost, more privacy and personalized cloud experiences, while relying on the service provider's public cloud infrastructure.

2.1.1.5 Hybrid Cloud

This deployment model is a combination of one or more of the previous deployment models, where data transfer or task handling can occur seamlessly between the different clouds.

2.2 Software-as-a-Service

Nowadays, with the proliferation of faster Internet connections and the ever-growing landscape of Cloud Computing (Dillon *et al.*, 2010), software has become more accessible to companies than before. By hosting and serving software through the use of Cloud Computing, that software's clients reduce both Capital Expenditure (CAPEX) and Operational Expenditure (OPEX) by eliminating the need to buy and maintain the software and underlying infrastructure (Alnumay, 2020).

SaaS allows users to access software and its data, usually hosted on cloud computing services, through thin clients and/or web browsers (Mell and Grance, 2011; Ali *et al.*, 2017). The *Multi-tenancy* design structure of SaaS enables the software to serve multiple users (tenants), from a central server. This design allows for more efficient use of both computer resources and the human resources needed to maintain and manage them, which lowers expenditures and is therefore an imperative for businesses. Through the use of SaaS instead of traditional software, the Clients no longer require the arduous task of deploying software to each one of the users, no longer dealing with varied user endpoint hardware configurations. Cloud Computing enables ubiquitous access to SaaS, which in turn makes its adoption by businesses more enticing to them. The use of SaaS allows not only for lower CAPEX and OPEX for the Client, but also enables faster and more frequent updates of the SaaS (since the software manufacturer has control over it), which increases safety and security for the Client's day-to-day operations (Cavusoglu *et al.*, 2008).

2.2.1 Security

There have been multiple occasions where security breaches could be prevented had the victims been using up-to-date software (Glenn, 2018). The amount of time and resources needed for patching security vulnerabilities varies from company to company but overall, can reach averages of 38 days (Rapid7, 2018). By relying on the SaaS provider to patch vulnerabilities in a timely manner, Clients no longer need to allocate costly human resources to this task, which lowers expenses and human error (Glenn, 2018).

The use of a SaaS solution enables the users to access the software and its data without the use of complex networking such as Virtual Private Network(s) (VPN) and locked-down user endpoints, which have shown its faults when not properly managed, during the SARS-CoV-2 pandemic (Adams *et al.*, 2022). By moving the majority of the responsibility for the system's security to the SaaS provider who are more likely to employ security best-practices, it eliminates security threats posed by the Client's deficient security measures.

2.3 Software Engineering

Software Engineering is the application of engineering to software. It's the *'application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software'*. Developing software is a process by which *'user needs are translated into a software product'* ("ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary" 2017).

According to the IEEE, this software development process involves the following steps:

- Translating user needs into software requirements
- Transforming the software requirements into design
- Implementing the design in code
- Testing the code
- Optionally, installing and checking out software for operational use

By following these steps, proper software development can be done in a timely and cost-effective manner.

2.3.1 Defining Requirements

Deciding on what and how to develop software is a difficult part of the software development cycle (Pacheco *et al.*, 2018). By properly defining what the software product requirements are, many software problems can be avoided after the development of the software finishes. Of all defects that software products can have, forty to fifty percent of those defects arise from errors during the first phases of software development: Translating user needs into software requirements and then transforming those into software design (Eugene Wiegers and Beatty, 2013).

Requirements specify implementation objectives. They are specifications of the system's behavior, attributes or properties or constraints during the development process (Sommerville and Sawyer, 1997). Requirements elicitation should be performed during the early stages of the software planning phase in order to prevent major refactoring of code before it's released, which would lead to significant costs for the software development company.

2.3.1.1 Identifying Key Stakeholders

Before requirements elicitation, one of the most important steps is asking from whom should such requirements be elicited from. This step is crucial to prevent functional (and financial) success of the project about to be started (Lewellen, 2020).

2.4 Software Architecture

Designing the overall system structure, how the software systems integrate with the infrastructure, how these systems interact with each other and with the users is a considerable challenge for large, complex software systems. Defining the software system, in terms of components and connections between these, is describing the system's software architecture (Hasselbring, 2018). Starting from the elicited requirements, and before putting into action the development of the software's code, the next step in software engineering is *Transforming the software requirements into design*. This step allows for easier iteration, since it's a design phase of the software development process, and thus can be modified in a easier and quicker way than performing those changes in an already implemented design. By producing documentation regarding the software architecture, someone unfamiliar with the whole project can be quickly introduced to the overall structure of the project, without needing to delve deep into the codebase.

2.4.1 Service-Oriented Architecture

There are multiple definitions of SOA (Niknejad *et al.*, 2020), but for the context of this piece of work, the definition given in (Marks and Bell, 2008) is the most adequate: *'SOA is a conceptual business architecture where business functionality, or application logic, is made available to SOA users, or consumers, as shared, reusable services on an*

IT network. Services in an SOA are modules of business or application functionality with exposed interfaces, and are invoked by messages'

2.4.2 Modularity

Since the early years of software development, there existed the concept of dividing software into modules to improve its comprehensibility and flexibility, while carefully decomposing software in order not to generate too many inefficiencies with this approach (Parnas, 1972). Not only are *comprehensibility* and *flexibility* good qualities for the software to possess, but also *dependability*, *maintainability*. These qualities derive from proper software architecture design and implementation and must be contemplated in this phase of software development, as they are too integral to the core software design to be thought of during the software implementation phase. By dividing the software into multiple components and as long as they are not tightly coupled, different developer teams can independently create those components, leading to increased flexibility in software development as well.

2.4.3 Reusing Software Components

By using modular software architectures, the modular software components are no longer required to be built by the same development team nor being built specifically for a single project. Code reuse is a common practice nowadays and, similarly, reusing whole software components in a software architecture introduces the same advantages, by reducing work to be done in order to create a new architecture and implement it and easier and more cost-effective maintenance (Hasselbring, 2018). Additionally, it increases system stability, security and reliability when reusing components that have been publicly published and evaluated by the software development community. However, when dealing with 3rd-party components, additional care must be taken to ensure that the evaluation of that software is done on a constant basis and that any changes to that component only reflect on the system after thorough review of said changes. Otherwise, external actors may introduce malicious code in the component before publication and compromise the security of those who rely on that component (Tal, 2022).

2.4.4 Cohesion and Coupling

In software development there two relevant goals during development: *cohesion* and *coupling*, where the first is related to how code is grouped together in a logical unit, the latter is related to how that code is dependent on each other. Ideally, developers nowadays try to develop in order to obtain high cohesion and low coupling (Candela *et al.*, 2016). With high cohesion, developers need not perform code changes in multiple different places to implement new features or fix a bug. With low coupling, those changes are less likely to impact related code.

2.4.4.1 Types of Coupling

Coupling can happen in multiple, different, manners. Given two components, A and B, if changing the implementation of A requires re-implementation of B, then there is ***implementation coupling***. This is a regular occurence when two components share the same database: if one components changes its database-schema structure, then those changes need to be implemented simultaneously in all other components that rely on the same database-schema structure. This is usually easy to fix, by requiring all components to access the database through a single service, like an API. If the database-schema needs change, then only the service that serves the database information needs to be changed.

Take component A and B, where A sends a syncronous message to B for it to perform a task. If these components rely on synchronous calls to each other to operate, then to perform those calls between each other they need to be up and running and reachable. This is called ***temporal coupling***. If something occurs that delays a message between A and B or if one of them is unavailable, then calls that A makes to B will not be delivered and will be lost. To avoid this, *caching* or *asynchronous* messages sent through a *message broker* can be used. *Caching* would function for serving cached static content and using a *message broker* would allow messages to be picked-up downstream as soon as the service downstream was available.

2.4.5 Monolithic and Modular Monolithic

There are multiple, design patterns for software architecture. The most common software architecture, that is generally the first architecture used when developing a new product in a prototype or academic environment is a monolithic architecture. In this kind of architecture, all (or a majority) of the functions of the software are encapsulated in a single software application (Chen *et al.*, 2017). This monolithic approach makes it easy to develop, test and deploy software, until the amount of functions it encompasses stops being small. When that happens, when the number of functions grows, evolving and updating the software becomes unwieldy. This is due to the high coupling that accompanies monolithic software architectures. In monolithic software architectures, the application's modules cannot be executed independently, changes to a function require the restart of the entire application, usually requiring considerable *downtime*.[1]

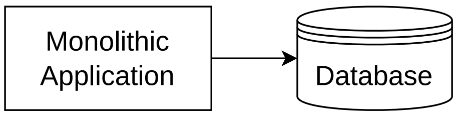

Figure 2.1: A single-process monolith: all code is packaged into a single process (Newman, 2019)

When a monolithic application is composed of multiple modules, where these can

[1] Time during which the Client doesn't have access to the application

be developed independently, but still need to be combined for deployment, it's called a modular monolith.

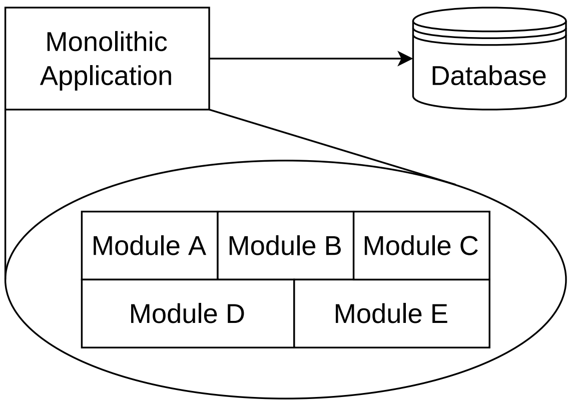

Figure 2.2: Modular Monolith: application code broken down into modules (Newman, 2019)

2.4.6 Microservices

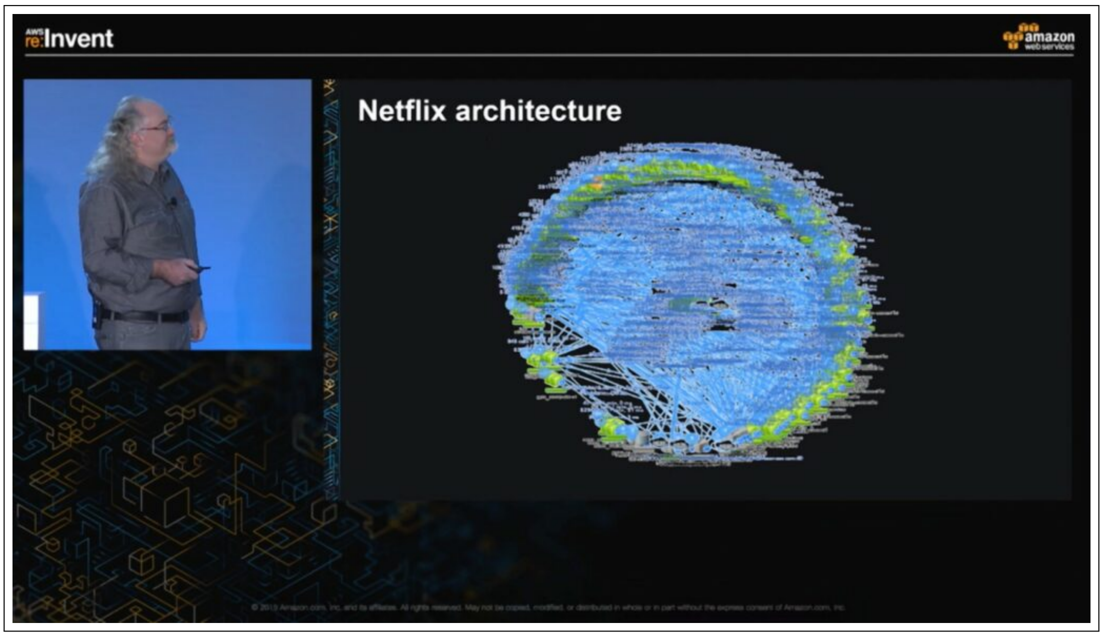

Figure 2.3: Netflix Senior Engineer Dave Hahn showing Netflix's microservice architecture at AWS re:Invent event in 2015.[2]

Splitting the software system into multiple, independently maintained, fine-grained components, allows developers to deliver software faster and embracing new technologies even faster.

[2] Available on https://www.youtube.com/watch?v=-mL3zT1iIKw

By resorting to microservices, new technologies can be tested and prototypes developed in a blink of an eye. According to (Newman, 2015), *'microservices are small, autonomous services that work together'*. While simplistic, this definition sums up quite well what microservices are. Transporting the Single-responsibility Principle (SRP) in (Martin, 2014) from classes to microservices, having these fine-grained, independent services allows for more cohesion. As Newman interprets Martin, *'Gather together those things that change for the same reason, and separate those things that change for different reasons.'*.

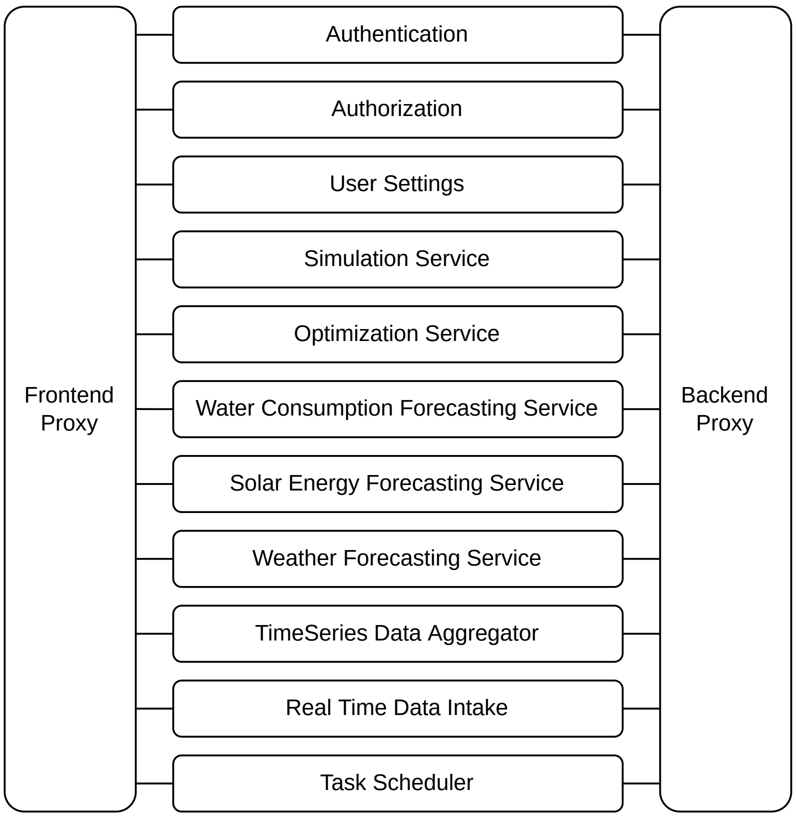

Figure 2.4: A partial example of a Microservice Architecture, with self-contained microservices. Based on Fig.3 from (Newman, 2015)

According to IDC's predictions (IDC, 2019), *'by 2022, 90% of all new apps will feature microservices architectures that improve the ability to design, debug, update, and leverage third-party code; 35% of all production apps will be cloud-native'*.

2.4.6.1 Key Benefits

In a monolithic architecture, the technologies used throughout the application need to be compatible with each other. Developing an application in Java and C++ and Python is not possible in such a monolithic architecture. By using microservices, developers can now take advantage of ***Technology Heterogeneity***. If an application is made up of several, different services that can communicate using shared communication protocols, then there is no reason they can't use different technologies. The application can have services providing, for example, real-time video encoding using CUDA[3] and C, have services performing data science tasks in Python, built for academic research in a production environment, Java-based services providing text analysis, among other examples. The goal with *Technology Heterogeneity* is to allow the development team to choose a technology stack that best suits a particular problem to be solved. If, after implementation, the technology stack is no longer ideal and needs to be replaced, it can happen much more easily and quickly. When transitioning from a monolithic architecture to a microservices architecture, this is ideal since it allows for the old technology stack to be transposed first, while the team readjusts to the new architectural rules. By doing this, the development team doesn't require training in new technologies during the migration since it can be done at a later time, and can implement the new architecture using the previous technology stack. Then, when the team has more availability, individual services can be targeted for upgrade to a better suited technology stack. By using microservices architecture, the amount of work required to replace legacy code or to remove unused code is smaller since the services are more granular and independent. Thus, the cost of replacing code is small and ***Replaceability*** is optimized.

Unlike monolithic applications, a component failure doesn't require the whole application to be unavailable to the clients. Since the services are plenty and independent, failures don't propagate as easily and thus, service to the client can be somewhat maintained, increasing the ***Resiliency*** of the application. This can also be transposed to deployment. *Deployment* for a monolithic application is difficult, as it requires the entire application to be deployed and thus, is highly impactful and prone to causing failure if just one component fails. Since the microservice application is resilient, *Deployments* present a lower risk of complete failure. Having less complex and more granular components, the microservice application has ***Easier Deployments***. This same low complexity and granularity also makes ***Testing*** easier and able to be performed more regularly, decreasing the chances of failure.

Microservices also allow for independence between development teams, as the *coupling* between services is very low. This means that backend development teams don't need to meet and engage with frontend development teams as often since the impact of changes to a backend service is also low.

During high usage spikes, a monolithic application would need to be scaled up in order to respond to the inrush of user requests. This means that all the components of the application would be scaled up, regardless of their importance to the high usage spike. This creates two problems, one being the computing resources that would need to be allocated to scale an entire application is much more challenging and the second relates to the inefficient use of resources, since not all components of the application would require the ***Scaling*** operation. By using microservices, individual services can be scaled up (or down) independently, quickly and possibly automatically in order to better suit the demands posed by the application's users.

[3] https://developer.nvidia.com/cuda-toolkit

2.5 DevOps

The *portmanteau* of Development and Operations — DevOps — is '(. . .) a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality', according to (Bass *et al.*, 2015). In (Sallin *et al.*, 2021), the authors start by analyzing the use of the Four Key Metrics (FKM) (Forsgren *et al.*, 2019) for measuring developer performance and DevOps adoption in software companies. Then, the authors evaluate how can these metrics be measured automatically.

2.5.1 Deployment Frequency

Deployment frequency relates to the frequency at which code is pushed into production. Generally, this is performed by making small changes and placing them in a production environment regularly. This is usually achieved through good Continuous Integration/Continuous Deployment (CI/CD) practices. The smaller changes lead to fewer potential issues arising during deployment to production and allow for more control over changes.

2.5.2 Lead Time for Change

This is defined in (Forsgren *et al.*, 2019) as the *'the time it takes to go from code committed to code successfully running in production'*. Having the goal to minimize this time leads to faster feedback after changes have been deployed, which in turn allow for faster correction in case of failure. This also applies not only to changes that introduce new features or modify them but also for potential bug fixing deployments. In some of the articles mentioned in the analysis performed by the authors in (Sallin *et al.*, 2021) it's mentioned that certain companies have used the *commit* into the version control system as the start of the change procedure.

2.5.3 Time to Restore Service

The time it takes to restore service, equivalent to the time between the start of *downtime* and its end, is an essential metric. With ever more complex systems and newer technologies, system failure is bound to happen. How, and how quickly a situation is resolved, is an essential metric for the day-to-day operations of a software company providing a product such as a SaaS.

2.5.4 Change Failure Rate

The rate at which deployments to production lead to system failure or impairment, requiring remediation. This remediation can be in the form of a hot fix, a patch, a rollback, among others. Some articles analyzed in (Sallin *et al.*, 2021) also suggest different interpretations to this definition such as using the percentage of releases that were performed as *fixes* or *hotfixes*, detecting failure through observability tools or manually marking deployments as failed or successful.

2.6 Observability

Virtual Cloud Computing allows users to configure near limitless services, spanning multiple types of resources and with a dynamic range of options for infrastructure. As such, software that relies on VPS's elasticity can also become more complex than when using private clouds. With the adoption of modular software design and the increase in the use of micro-services and *serverless* compute services, the complexity of software architecture has increased greatly (Niedermaier *et al.*, 2019). Such complexity comes at a cost however: Lower Observability. Taking a page from modern control system theory (Gopal, 1993), Observability refers to *the degree to which a system's internal state can be determined from its output.* The ability to closely monitor a system's internal state is beneficial during software deployments as well as after them, as it allows stating whether the system is running according to plan. While a few services can be easily monitored by a single human resource inside a company, complex systems require huge efforts to keep in check on a regular basis. Erratic or unexpected system behavior can be spotted when certain patterns make themselves apparent through monitoring. These patterns become difficult to spot when the amount of monitored metrics are inadequate for the system's complexity. As a means to increase transparency in these new distributed and complex systems, observability tools have emerged that allow for *traces*, *metrics* and *logs* to be generated, collected and further analyzed so that insightful information towards the system's internal state can be attested.

2.6.1 OpenTelemetry

An open-source project, OpenTelemetry (OTel) is a framework born from the software industry's interest on open-source tools, Software Development Kit (SDK) and Application Programming Interface (API) for sending this monitoring data to a Observability back-end in a standardized, vendor-agnostic way (OpenTelemetry, 2022). Before this open-source solution became available, Observability software required the software developers to use that specific Observability back-end's libraries and agents to emit the required data. From a technical and business point-of-view, this was harmful to a software company, since it greatly reduces the ability to quickly and easily change Observability back-end, locking the company in using the same observability software for long periods of time, regardless of the adequacy of it. With this solution, open-source and innovative add-ons and custom tools to enhance Observability of a system can be made and implemented in much less time, while allowing changing the tools to interact with each other, generating even more insightful knowledge about the systems internal state.

2.6.2 Telemetry

Telemetry, in the context of this document, refers to the data a system sends regarding its internal state. For software, this data can be in the form of traces, logs and metrics. ***Logs*** are timestamped messages emitted by a service or component of a system, which inform about a specific occurrence, such as a request being made to a service or logging the time it took for a function to perform. ***Traces*** are data that informs about the path that a request took while it propagates through a service or component. If it traverses more than one service, it is called a ***distributed trace***. Distributed traces keep software developers and maintainers informed about the entire path that a request might make, which is a hard task to perform when dealing with multi-service software architectures, like microservice or serverless software architectures. These kinds of architectures are usually complex and non-deterministic, which make debugging quite an endeavor. Individually, these traces and logs provide information about a specific event or set of sub-events that are related to an event. However, in order to ensure system reliability, a system needs to be monitored not just for a single instant but throughout time. This gives an additional dimension to the data emitted by the system's components. By aggregating numeric data over a set period of time, ***metrics*** can be obtained that give more insightful knowledge regarding the system's internal state. For cases when the information is not numeric in nature, for example a *log* informing that there has been an error, this information can be transformed to inform of the frequency of the event that created that message. Thus, this quantification of data allows for metrics to be recorded and shown graphically, where patterns can be detected. By quantifying telemetry data and generating metrics, it becomes possible to evaluate the system's behavior before, during and after a software deployment so that it's success can be ascertained (Mills, 1988)

Chapter 3

Methodology

Each Water Utility is a Client, and the Product is the Application (the DSS) that the Company (SCUBIC) provides as a cloud-based service. In this chapter, the old architecture of the Application is explained, its flaws are exposed and possible solutions are analyzed. From these possible solutions, a new architecture is proposed, one that corrects those flaws.

3.1 The Application

The software that the Company provides to each Client allows the Client's water operators and managers — the Users — to access multiple application modules:

- Monitoring Module, where data from the Client's sensors can be consulted using charts and other visualization methods. This is the base module, necessary for using the other modules.
- Forecasting Module, which performs machine learning operations using the Client's historical sensor data and forecasted weather data to forecast water consumption for a pre-specified period after execution.
- Forecasting Model Training Module, which trains the machine learning models used in the Forecasting Module.
- Optimization Module, which relies on the Client's sensor data, forecasted water consumption data and the Simulation submodule to optimize the pump operation schedule for lower operational cost. By optimizing the pumping operation, water and energy usage efficiency increase, lowering CO^2 emissions and reducing the cost to operate the pumps.
- Simulation Module, where a *Smart Digital Twin* of the Client's water network is created and its water pump operations simulated.

- KPI Module, which performs arithmetic calculations to generate KPI regarding the Client's operation.
- Solar Forecasting Module, which forecasts photovoltaic solar panel power Production for use in conjunction with the Optimization Module.

By using the Application's Monitoring Module, each User can access the data generated by the modules.

For these modules to work, each Client is required to send their sensor data, with adequate frequency. Depending on the sensors, this data can be collected by the Client's sensors from every minute up to every hour, which is not synonymous with the data intake frequency. There are Clients who have sensors in remote locations which log data in 15 minute intervals, but due to power constraints this data is only sent once or twice a day to the Client's central monitoring system. Thus, the distinction between data intake frequency and data frequency must be made: the first is related to the frequency with which the sensor data packages arrive at the Application and the latter, the frequency or the time interval between each point of data in the set of sensor data. While the first is important for proper scheduling of the forecasting and optimization tasks performed by the Application, the latter is crucial to those tasks and requires a frequency of up to 1 hour.

Once data is sent from the Client's databases and to the Application, the data is pre-processed and stored in a Timeseries[1] database. The modules access this data in order to perform their tasks.

These Forecasting, Optimization and Simulation, Solar Forecasting and KPI calculation tasks are performed with a frequency ranging from 8 to 24 hours, every day. For most clients, these tasks are performed once a day, at midnight, in order to prepare the next day's operations. The duration of said tasks can vary, depending on the amount of water consumption points to forecast, the complexity of the water network or on the amount of sensor data to process. These tasks perform calculations using medium-to-large Timeseries datasets and utilize machine learning algorithms or complex optimization algorithms in conjunction with water network simulation algorithms. Thus, these tasks are run asynchronously, in *Python* workers.

3.2 The Old Architecture

The old software architecture is still in use as of the date of publication of this body of work, alongside the proposed new architecture. They are both in Production, with older clients using the old architecture and new clients using the new architecture.

[1] Series of data points indexed in time order

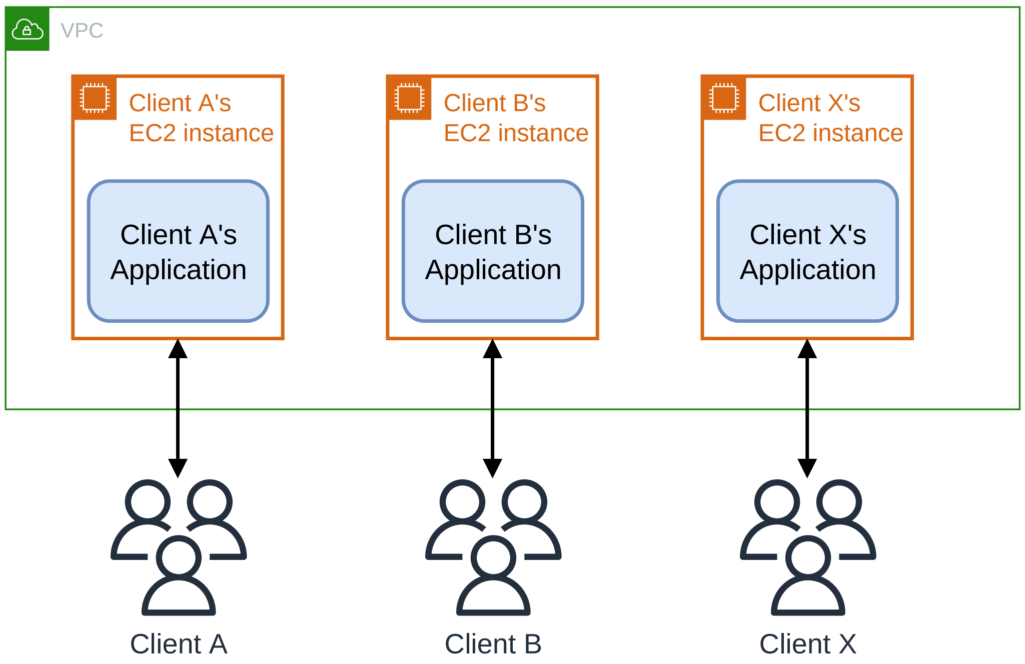

Figure 3.1: The AWS VPC used, hosting the old architecture's EC2 VPS

3.2.1 Overview

The old architecture is composed of a group of Elastic Compute Cloud (EC2) instances, one for each Client, inside the Company's VPC. Inside each EC2 instance, using Docker[2] for container orchestration, an Application is executed for that specific Client. Each Application consists of the joint deployment of a set of services and databases, which run inside Docker containers. As can be seen in figure 3.1, this architecture requires an EC2 instance for each Client, which is not scalable for reasons explained later in this document.

3.2.1.1 VPC

As can be seen on the diagram presented on Figure 3.1, these instances are deployed to the same VPC, sharing a private network between them.

3.2.1.2 Services and Components

Each EC2 instance runs a Docker container for each one of the following services:

- **InfluxDB**[3] (Timeseries Database)
- **Telegraf**[4] (Data collecting service)
- **MongoDB**[5] (General use, no-SQL, Document Database)

[2] https://docker.com
[3] https://www.influxdata.com/products/influxdb-overview/
[4] https://www.influxdata.com/Timeseries-platform/telegraf/
[5] https://www.mongodb.com/

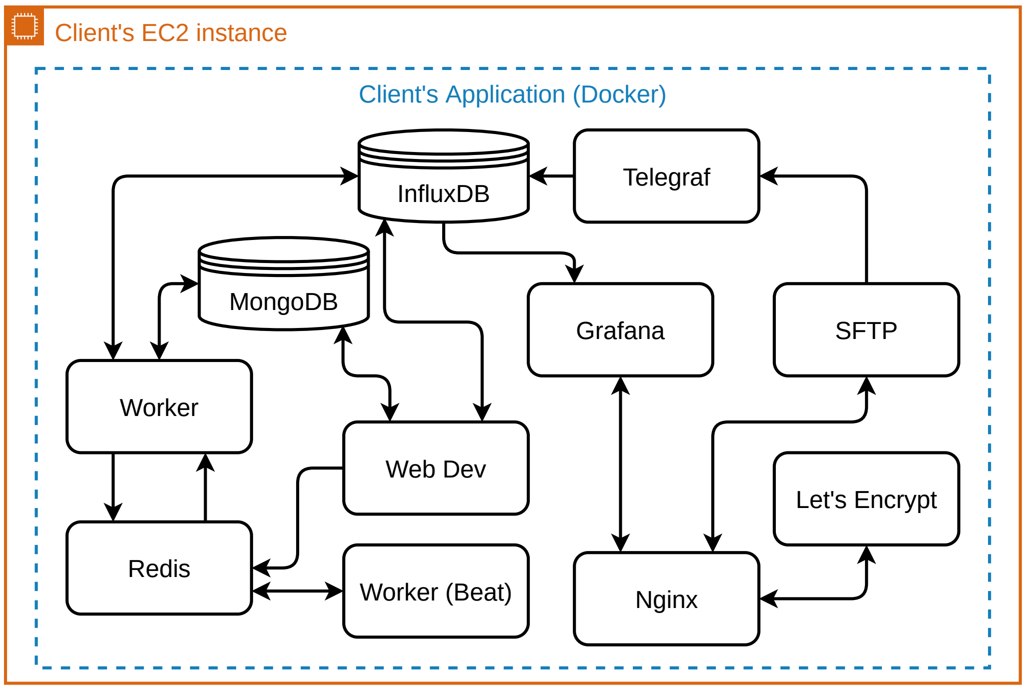

Figure 3.2: Old Architecture: the containers and their connections inside the Application

- **Grafana**[6] (Web platform for data visualization, the front end of the DSS)
- **Nginx**[7] (Reverse proxy with Secure Hypertext Transfer Protocol (HTTPS) capabilities)
- **Let's Encrypt**[8] (Automatic Transport Layer Security (TLS) Certificate installer, companion for the Nginx container)
- **Web Dev** (Flask[9] Web platform / API for managing Workers' settings)
- **Redis**[10] (Message Queue System for queuing Worker's jobs)
- **OpenSSH**[11] (*atmoz/sftp*) (Secure Shell (SSH) Server for receiving client data through SSH File Transfer Protocol (SFTP))
- **Workers** (Celery[12] Container running the Forecast, Simulation and Optimization *Python* Algorithms as well as the KPI Algorithms.)
- **Workers** (*Beat*) (Celery Container that periodically *triggers* jobs in the Workers container)

[6] https://grafana.com/
[7] https://www.nginx.com/
[8] https://letsencrypt.org/
[9] https://flask.palletsprojects.com/en/2.1.x/
[10] https://redis.io/
[11] https://www.openssh.com/
[12] https://docs.celeryq.dev/en/stable/

3.2.1.3 Service's Connections

In Figure 3.2 the relations between these containers can be schematically seen. Starting on the right side, with the Let's Encrypt and Nginx containers, these provide outside access to the Grafana and SFTP services inside the respective containers. Data from the InfluxDB database is read by the Grafana service which allows the Client's users and the company's developers to query the database and at the same time generate charts with such information. Client sensor data is sent to the SFTP server that shares the incoming files with the Telegraf service and allows it to pre-process that sensor data and proceed to the data intake into the InfluxDB database. Then, either through remote access to the Web Dev container or automatically through the Worker Beat service, tasks are sent to the celery queue (using the Redis service) and picked up by the Worker service. This Worker service then accesses the MongoDB Database to load algorithm and device configurations and the required client sensor data from the InfluxDB database before running the tasked algorithm. Data resulting from the execution of the algorithms is then sent to the InfluxDB database, to be read by the Grafana service. There are some connections that are bidirectional, such as the Web Dev to the MongoDB database which is the service used to manipulate the MongoDB database's algorithm and device configurations.

3.2.1.4 Databases

There are two types of databases being used by this architecture: A Timeseries Database, in this case **InfluxDB**, and an additional general-purpose Document Database: **MongoDB**. Each type of database has a different role, the first one stores the Client's timeseries data such as sensor information, pump orders, predicted tank levels, etc. The second one, the Document Database, is responsible for storing configuration settings for each worker service (optimization, simulation and forecasting), for storing electrical tariffs data and to store sensor device's configurations.

3.2.1.5 Grafana

This web platform allows the visualization of the Timeseries data from the **InfluxDB** database. This is an open-source platform that runs on a docker container with little to no modifications necessary. The dashboards are built using the built-in tools and allow for complex and very informative data visualization.

3.2.1.6 Telegraf

The **Telegraf** container is used to gather the files containing the raw sensor data sent from the Client to the SFTP server. Since this container shares the file upload location folder with the SFTP, through a convoluted process of storing the filename of the last file uploaded, periodically checking for the next file and file handling *spaghetti* code that spans multiple files and has an enormous codebase that weighs the docker image's file size considerably.

3.2.1.7 SFTP

The SFTP service here provides a secure method for the Clients to send files containing the Timeseries data to our servers, where they can be processed and turned into actionable insights by the algorithms running in the Workers container. The Client sends their public key (from a cryptographic key pair) when the project start to authenticate against this SFTP service and uploads the files to a pre-designated folder. These files are then accessed by the Telegraf container which performs the file intake process.

3.2.1.8 Nginx + Let's Encrypt

These two containers allow secure Internet access from the EC2 instance into the correct docker container IP address and port. The Client-facing services Grafana and SFTP which, respectively, provide the web interface for the DSS and client file input service are inside containers which themselves can change their internal IP inside the Docker environment. To keep the dynamic IPs in check and allow for these services to be accessed from outside the Docker environment, the Nginx container keeps track of this dynamic IP and updates its route table accordingly. This allows for any of these two containers to restart, change their IP address and still not break the routing back to the host EC2 instance, which has an Elastic Network Interface (ENI) associated to it exclusively. This ENI is then connected, exclusively, to a single Elastic IP (EIP) to which the Clients connect, like Figure 3.3 implies.

As for the Let's Encrypt container, this container shares a docker volume with the Nginx container and automatically and periodically maintains the TLS certificate files that the Nginx requires in order to serve the Grafana interface through HTTPS.

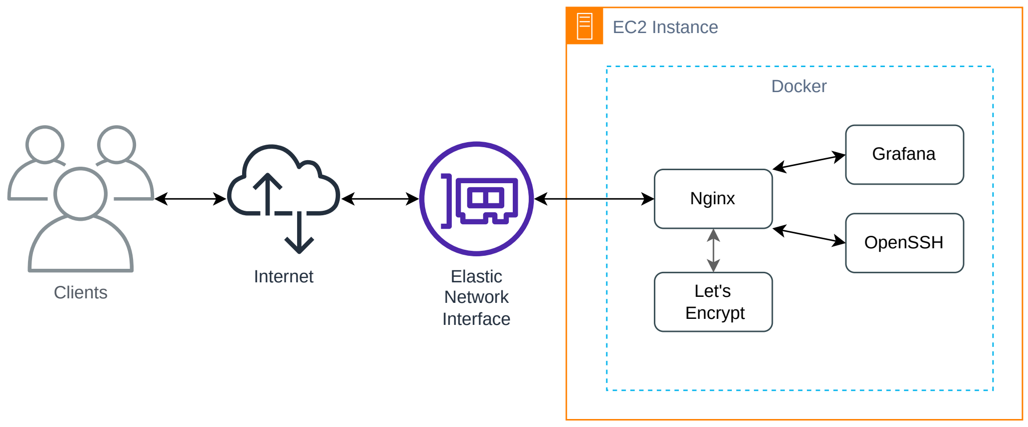

Figure 3.3: Internet access to the Client-facing services

3.2.1.9 Redis

Redis is used as a message queue backend for Celery, enabling other services to send Celery tasks to a queue for asynchronous execution by the Workers.

3.2.1.10 Web Dev

Based on Flask, this web application serves an API as well as serving a web page that gives developers access to algorithm configurations and the ability to push Celery tasks to the queue. This application connects directly to both databases.

3.2.1.11 Workers

The Workers' container image is built *in-house* by the development team, using a *Python* Docker image as the base image, wherein all the company's algorithms lay. The *forecast*, *optimization* and *performance analysis*/KPI algorithms are individually linked in a Celery configuration file, which defines how each algorithm is executed in a Celery task and how that task is called. This container executes a Celery Worker that executes all Celery Tasks in the Celery task queue.

When a task is sent to the task queue, this Celery Worker who polls the task queue, picks the task up and starts executing the task as soon as possible.

There are two Workers images, the first one contains the code for all algorithms and is the one which starts the Celery worker. The other one, which is internally called Celery Beat, executes a Celery instance in *Beat* mode which sends pre-configured Celery tasks to the queue. This is used to run the algorithms periodically in order to process the Client data and generate actionable insights for the Client.

These algorithms require decent amounts of computer resources, namely CPU power and RAM capacity, in order to be able to run effectively. This is a direct contrast to the remaining components of this old architecture, which see minimal Client use and are therefore less resource intensive. In terms of storage, the situation is the opposite since these algorithms use data stored within the other services: the database services.

3.2.2 Issues

Besides an individual EC2 instance, each Client also has an individual GitLab[13] project, which is composed of several different Git[14] code repositories. Each GitLab project contains the following repositories:

- **dbs** (Databases configurations, build files for databases' docker images, deployment scripts)
- **Workers** (Build files for the Workers' docker images)

[13] https://gitlab.com/
[14] https://git-scm.com/

- **DBconnectors** (Standardized code for database access)
- **forecast_optimization_api** (Code and build files for the Web Dev docker image)

In the **dbs** repository, build scripts for custom docker images for InfluxDB, Nginx and Telegraf can be found. Also, here reside the scripts that are used to remotely deploy docker containers to the EC2 instances as well as the *docker-compose* configuration files. The GitLab CI/CD pipeline that deploys the old architecture to the instances also resides here.

As for the **DBconnectors** repository, database connectors can be found here. These allow offloading the code that connects to the databases from the algorithms to a separate module, which can be reused throughout the same GitLab Project and, in theory, keep the query methods consistent for both the **Workers** and **Web Dev** codebases.

In the **Workers** repository, the code for the algorithms used by the platform to perform the forecasting, optimization and KPI calculation as well as the **DBconnectors** repository linked as a submodule can be found.

In the likeness of the **Workers** repository, the **forecast_optimization_api** repository also imports the **DBconnectors** repository as a submodule. This **forecast_optimization_api** repository is where the *Web Dev* container build code is situated.

3.2.2.1 Low Cohesion and High Coupling

This old architecture has severe problems regarding its Cohesion and Coupling. The readers will notice that, as shown both above and on Figure 3.2, there are multiple services performing read and write operations to the InfluxDB database. Although concurrency is not a major problem, having different schemas and tag names for InfluxDB queries in different services has historically led to multiple timeseries data not being detected when querying the database. Such things happen when a different querying service places the data in the database. This is due to mismanagement of repositories and git submodules, and requires additional care, planning and communication from the developer team's side. Here, having a specific service to perform pre-prepared queries, with very detailed database schemas, to which all other services would connect to query/write to the database would solve this problem. Once again, cohesion is low, since code that is used to connect to the database is spread out throughout the codebase. It also means that there is *major implementation coupling.*

Then, there are issues regarding *temporal coupling.* When performing data intake, the timeseries database, the data intake service (Telegraf), the SFTP service and the Nginx service all need to be up and running and accessible to be able to perform data intake. If one of these services is unavailable, data that is sent from the Client is not processed when the unavailable service becomes available again. This has been something that has happened before, with some regularity, and the only way to recover the missing data is through manual data intake of the received text or spreadsheets that are sent to the SFTP service which is tedious and prone to errors.

Finally, there is the issue of *deployment coupling.* In order to deploy a small change to any of the services, all docker containers are shutdown and restarted, which results in considerable *downtime* for each Client.

3.2.2.2 Replaceability

The old architecture possesses low replaceability, since it's not easy to change a service for another, such as the timeseries database or the data intake service. For example, with the exception of the data visualization service (Grafana), major refactoring of code would have to be done, since each service connects to the timeseries database in different ways.

3.2.2.3 Resiliency

Despite the high coupling of the application, with the old architecture using docker containers, usually the application doesn't become totally unusable if one of the service fails. The problem lies with the docker orchestrator or the EC2 instance itself. If one of these fail, then the whole application becomes instantly unavailable. If the EC2 instance becomes unresponsive, as has happened multiple times before, then a manual, full system reboot of the instance followed by a redeployment of the whole application stack is inevitable.

3.2.2.4 Deployments

A deployment of the application that uses the old architecture is a tiresome and arduous affair, since the application is highly deployment coupled as stated before. After *committing* a change to one of the repositories, deployment involves *tagging* the repository of code in which the change was made in order to create the necessary docker images with those tags through CI/CD. In this step, a GitLab runner will pull the repository, checkout any git submodules, perform unit tests, and then start building a docker image with the code inside the whole repository. Currently, for the old architecture, this step takes around 10 minutes to complete. After the tagging of the repository of code is done, the next step is to perform the same task with the *dbs* repository, where another 10 to 20 minutes of docker image building takes place. Then, after all of the docker images are built, the GitLab runner opens an SSH shell to the EC2 machine of the Client, pulls the respectively tagged docker images and performs a docker-compose operation. This operation instructs the docker orchestrator to shut down all running containers, delete the volumes (with the exception of the databases) and then re-create the containers with the newer docker images. In all, this process takes around 30 minutes to complete. During this last step, the Users experience *downtime* with a duration of between 1 and 4 minutes, if the deployment is successful.

If a change is to be done for all Clients simultaneously, this process needs to be repeated individually, once for each client.

One of the faults with the older architecture is also the lack of different environments for deployment. That means that every deployment made to each Client has the very real possibility of breaking Production for that particular Client, where the faults would impact the Client's usage of the platform directly. This is a recurring event when deploying, as the algorithms are quite complex. Given the fact that some algorithms use real-time data gathered from the last one hundred (100) days, the somewhat unpredictable nature of the algorithms' execution results make the repeatability of results from day to day not trivial. Breaking changes are also not always apparent, since some algorithms perform calculations using data generated by other algorithms and/or real-time data and such mistakes only become apparent on the following work day, after their execution. There are cases when the algorithms run perfectly during week days, but fail during the weekends (since the water consumption patterns change accordingly), and are left in broken state until the next working day.

All of these mishaps lead to the creation of a Staging server where changes to the platform or algorithms could be tested with real data, causing no impact to the Clients and allowing the results to be monitored for longer periods of time in order to ascertain system reliability. As such, a Staging environment should replicate as much as possible the Production environment, be it the Operating System version, it's installed packages, *Python* versions, *Python* packages, the data in the server, the quick-fixes applied to Production, etc. This, however, meant that a similar, Staging environment EC2 instance needed to be running simultaneously with the Production environment's EC2 instance, effectively doubling the infrastructure costs. Since each Client had its own EC2 instance, this approach would also be impossible to maintain. An attempted approach was to use a single EC2 machine, sized similarly to the highest performing EC2 machine used by one of the Clients, to act as a Staging server for each Client at a time. Each time a major change was to be deployed to a Client, it would be first deployed during a set time to this Staging server and upon success, be deployed to the Client's Production server. Having multiple developers perform different deployments, for different Clients, at the same time, meant that Deployment Frequency lowered and Lead Time for Change increased as well.

3.2.2.5 Testing

Having the components of the application so tightly coupled, means that it requires the entire application to be executed in order to properly test the entire application. This is cumbersome and forces the developers to have a local copy of the entire application, including the timeseries data. This data can have big dimensions, and the only method to test with this data is to execute a script that connects remotely through SSH to the EC2 machine and makes a copy of the timeseries' docker instance's volume. Said volume copy is then transferred back to the developer through an SSH tunnel, so that the developer can then use that volume with the timeseries' docker instance that is running locally, for testing. Besides the massive data copy, which occupies disk space in the developers' computer and results in higher data transfer costs in the AWS account, the developer is required to have hardware capable of loading and executing all services simultaneously.

3.2.2.6 Scaling

The contrast between the different services' computational and storage requirements is one of the major issues of the old architecture. Adequate instance sizing is essential to lower infrastructure costs with compute resources. As can be seen in Figure 3.4, the CPU average utilization is usually very low, indicating that the resources allocated to this instance are way overestimated, elevating the infrastructure costs for no reason. However, the peaks in CPU usage that can be observed in this same Figure, which are caused by the periodically-running algorithms, push this CPU usage up to levels that suggest the allocated resources are somewhat adequate for this use-case. And wherein lies one of the major issues: over a 24-hour period, the amount of time spent with very low CPU usage is visibly and significantly superior to the time spent with adequate CPU usage for the instance size.

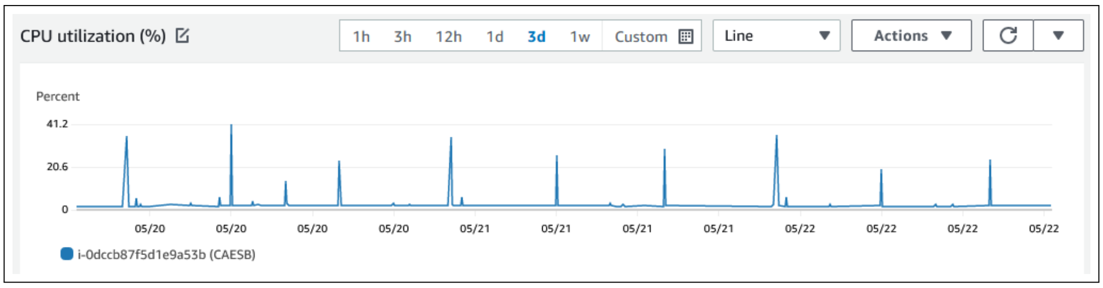

Figure 3.4: Client's EC2 Instance average CPU usage, during a three-day period, in 5 minutes intervals

The EC2 instance upon which these services reside can be provisioned and sized to different computational and storage needs. However, this would mean that it would either be adequately sized for the times the workers are dormant and undersized for when the worker's algorithms are running, or oversized for most of the time and only adequately sized while running said algorithms. Unfortunately, resizing an EC2 instance requires downtime for the whole platform, since it requires the EC2 instance to be rebooted. Since this would also stop Client access to the DSS and data intake service, this option cannot be contemplated. After testing a platform implementation with an instance adequately sized for the instants when workers are dormant, it was concluded that the algorithms would either refuse to run or crash when performing resource intensive calculations due to low RAM availability. The decision was then made, to keep the platform running in oversized, and costly, EC2 instances.

One possible solution was to split the resources based on their compute resource requirements. Having the workers on a separate EC2 instance that would be automatically and periodically provisioned and unprovisioned according to a schedule would allow the remaining services to be placed in a lower cost EC2 instance, lowering the overall infrastructure costs. However, without altering the existing architecture, this would mean that the alteration would only be the place where the Workers' docker container would be executed. Since the amount of EC2 instances is directly proportional to the amount of Clients, having two instances would duplicate the computational resources, networks connections and storage space needed to maintain the platform for all Clients. This would exacerbate the problem of limited compute resources available to our Amazon Web Services (AWS) account.

One of the issues with the old architecture is that the number of EC2 instances needed was directly tied to the amount of Clients, since each Client required its own instance to host the platform, generating what is called a Scalability problem. For the company's AWS account, a limit of thirty-two (32) Virtual CPU (vCPU) units (each vCPU corresponds to a processing thread in a CPU core) was imposed by Amazon as default, which meant that the sum of EC2 instance's vCPU units could not surpass this value. Each client requires an EC2 instance of the type `t3a.large` or `t3a.xlarge`, respectively two (2) or four (4) vCPU units, depending on the Client's Water Network's size and complexity and contracted services. This would mean that the amount of clients was limited from sixteen (16) clients if they all used the smaller instance or down to eight (8) clients if these Clients required more resources. As can be concluded, this is a hard limit on the amount of clients that can be served simultaneously by the company, which is an obvious problem.

3.2.2.7 Cost-Effectiveness

As of June 2022, the hourly price for on-demand (*EC2Cost*) `t3a.large` and `t3a.xlarge` EC2 instances in the nearest AWS region (`eu-west-3`) was, respectively, \$0.085 and \$0.1699. Since each instance requires storage, and the free storage is not enough for the data, the Elastic Block Storage (EBS) volume for each instance was of 256 GB in size (*EBS Size*). The pricing for the `gp2` EBS volumes is \$0.116 per GB-month of provisioned storage (*EBS Pricing*). Thus, for each Client, the total monthly cost of just the instances *Client Instance Monthly Cost* is given by the formula:

$$ClientInstanceMonthlyCost = (EC2Cost \times 24 \times days) + (EBSSize \times EBSPricing) \tag{3.1}$$

With this information, assuming a standard month of 30 days and a `t3a.large` instance, the average cost expenditure with a single Client's EC2 machine is:

$$\begin{align} ClientInstanceMonthlyCost &= (0.085 \times 24 \times 30) + (256 \times 0.116) \tag{3.2} \\ &= 90.896 \tag{3.3} \end{align}$$

Using the same variables, but for a `t3a.xlarge` instance:

$$\begin{aligned} ClientInstanceMonthlyCost &= (0.1699 \times 24 \times 30) + (256 \times 0.116) \qquad (3.4) \\ &= 152.024 \qquad (3.5) \end{aligned}$$

3.2.2.8 FKMs

Regarding the FKM, based on the previous issues that have been shown, by using the old architecture, the development team has suboptimal results in the metrics.

As mentioned in the Deployment issue, the difficulty with which deployments are performed forces the development team to gather numerous changes before deploying them so that the deployment can be performed less frequently, resulting in less *downtime* for the client. This, inevitably, lead to both a lower *Deployment Frequency* and also a bigger *Lead Time for Change*.

In the Deployment issue, it's also mentioned the amount of *downtime* that is to be expected when deploying changes to Production. Such an amount of *downtime* is unacceptable for a normal, eventless deployment. However, when failure occurs in the application using the old architecture, the *Time to Restore Service* is composed of multiple amounts of time: the time it takes for the failure to be detected, the time is takes to find a mitigation for the failure or troubleshooting time, the time it takes to re-deploy the application (which is around 20 to 30 minutes in case the mitigation requires changes to the code) and the time it takes for the recently deployed services to be back online. If failure occurs during the run of a task, that task (and subsequent tasks that failed to start) will need to be run manually as well.

Since the development team gathers numerous changes before deploying them to Production, the chance of failure increases. Besides exacerbating the problem with the *Time to Restore Service*, this method of deployment also increases *Change Failure Rate*.

3.2.3 Observability

One of the issues with the old architecture was the lower Observability that it provided to the Maintainers. Despite having extensive logging for each one of the services, the other two key components of Observability - metrics and tracing - were not present at any meaningful scale. Having to peruse hundreds of lines of code, filtering different services and log levels just to manually create metrics for algorithm execution time was time-consuming and tiresome. There was also no tracing put into place anywhere in the platform. To combat this, it was stipulated by the *stakeholders* that the new architecture should contemplate measures to increase observability of the entire system.

3.2.3.1 Alerts

As a consequence of the old architecture's lack of system observability, there were no useful metrics being created and stored besides the ones pertaining to the algorithm's result. In order to produce alarms, metrics are required to have a set of thresholds for each one of them. Alarms automatically inform the Maintainers and *stakeholders* of unexpected system behavior or catastrophic system failure in a timely manner, giving the chance for the development team to trace the cause(s) of the problem(s) before they become apparent and/or disruptive to the Clients. For some Clients, there were metrics and alarms setup based on the Client's water tank level, that would send messages to a Slack channel shared between the company and the respective Client, but fell into disuse.

3.3 Proposed New Architecture

'The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended' — Bashar Nuseibeh and Steve Easterbrook

Next, a proposed new architecture of the Application is presented. This new architecture minimizes or resolves the issues present in the old architecture, mentioned in the previous section.

Changing from the old architecture to the new one isn't a straightforward process. Having clients who are still using the infrastructure upon which the old architecture relies doesn't allow for mistakes while doing the migration. This brings several challenges, which are compounded by the lack of a functional new web interface for the new architecture. For this migration to occur, careful planning is to be done and checked by the *stakeholders* before any changes are put into Production. Measures such as changing network configurations, restarting services or run benchmarks on the old infrastructure cannot not affect any Clients using the old infrastructure.

To further complicate the planned migration, during the planning and implementation phase of this project the *stakeholders* required multiple changes to accommodate new Clients, which had to be applied to the new architecture. These changes and late-requests shape the decisions taken during the planning and implementation phase of the migration. For one of the new Clients, that the *stakeholders* arranged while the migration was concurring, there was a dilemma: Further increase the number of Clients using the old architecture (and subsequently, old infrastructure) or risk having this new Client as test subject for the new architecture? After discussion with the *stakeholders*, the development efforts were shifted from all current project to implementing the new architecture and adapting the algorithms to make use of this new architecture.

3.3.1 Solving the Deployment issues

In order to solve the deployment issues, one first step is to combine all of the different Applications (one for each Client), into a single Application that can serve all Clients simultaneously, as shown in figure 3.5. With this, applying further modifications later on to improve the old architecture is easier, since it's only one Application that requires change. Note, however, that in figure 3.5 the application is shown inside the VPC, with no underlying infrastructure. This is by design, since the underlying infrastructure is not relevant to this point.

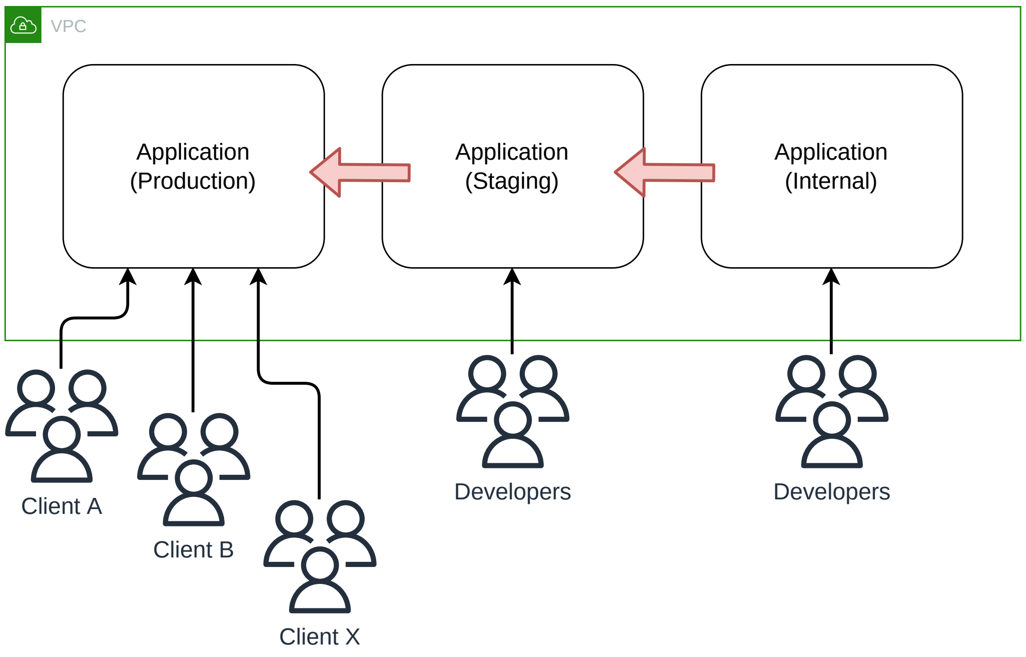

Figure 3.5: Initially proposed architecture. All Clients access the same Application and the Application is tested in a separate Staging and Internal environment.

With this approach, the Application can be deployed only once for all Clients, reducing the workload of the development team. By having the Clients use the same Application, bugs or failures that might have occurred in multiple clients in the old architecture can now be resolved simultaneously by a single deployment. A single Application also means that the use of different environments can now be contemplated again, seeing as there is only the need for one copy of the Application per environment. Using three environments — Production, Staging and Testing (referred as Internal throughout the document) — the development team can apply changes and test them before they reach Production. Developers use the Internal environment to test new code to ensure functionality. If the Application proves to be stable, it's then copied to the Staging environment, where it stays running for a set period of time so that developers can simulate User behavior and ensure that there are no problems whatsoever with the Application. Then, after Staging, the Application is sent to Production. This simple step, although not easy to implement as it requires extensive code refactoring, is one of the most important changes to solve the Deployment, Scaling and Cost-Effectiveness issues. This measure also allows for improvement in the FKM. *Deployment Frequency* increases since there's only one Application to be deployed, therefore less work is required when compared to having to deploy multiple Applications for small changes for all Clients. *Lead Time for Change*, *Time to Restore Service* don't alter as much, as the deployment time for a single Application hasn't changed much with this first step and neither has the time it takes to restore service. *Change Failure Rate* decreases drastically by implementing this first step. By having two other deployment environments to where the changes are first introduced, the code that is deployed to the Production server has been thoroughly tested and therefore less prone to failure.

Although moving to a single Application for all of the Clients, reducing the amount of resources needed overall even with the introduction of multiple deployment environments, the Application is still monolithic. As such, Scalability, Replaceability and Resiliency issues are still not resolved.

3.3.2 Solving the Scalability issues

By decoupling the Applications' components, it becomes possible to individually and automatically scale the necessary components when needed. Using this approach, services which are only run periodically can also be scaled down to 0 when not in use, such as the Workers service. If each worker performs a task per day, and if task has a duration of 15 minutes, then the total amount of time that service is online daily is reduced to around 98%, massively reducing computational resource use and effectively lowering cost as well. As explained in the subsection regarding Scaling in the previous section, different modules of the Application have different compute resource needs. Some modules which are run constantly, such as databases and APIs, can be run on low powered containers. The opposite can be done for Workers, where high-powered compute resources can be given to the container, albeit for very short periods of time, thus increasing the *Cost-Effectiveness* of the Application.

For that reason, the next iteration of the architecture encompasses these changes, as can be seen in Figure 3.6.

Now that each component is separate, they can be run on different infrastructure. For that reason, the Application will be running as containers as in the previous architecture with the change that these will not be orchestrated and hosted on an EC2 instance but instead on elastic infrastructure. By using AWS's Elastic Container Service (ECS)[15], each individual container can be run independently of each other and be automatically orchestrated and deployed by AWS. These containers can use either On-Demand ec2 instances, managed by AWS, or an on-demand serverless service — Fargate[16] — provided by AWS. These instances can be combined to form clusters, to where these containers can be deployed.

Using this approach, there are multiple advantages over the previous architecture:

[15] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html
[16] https://aws.amazon.com/fargate/

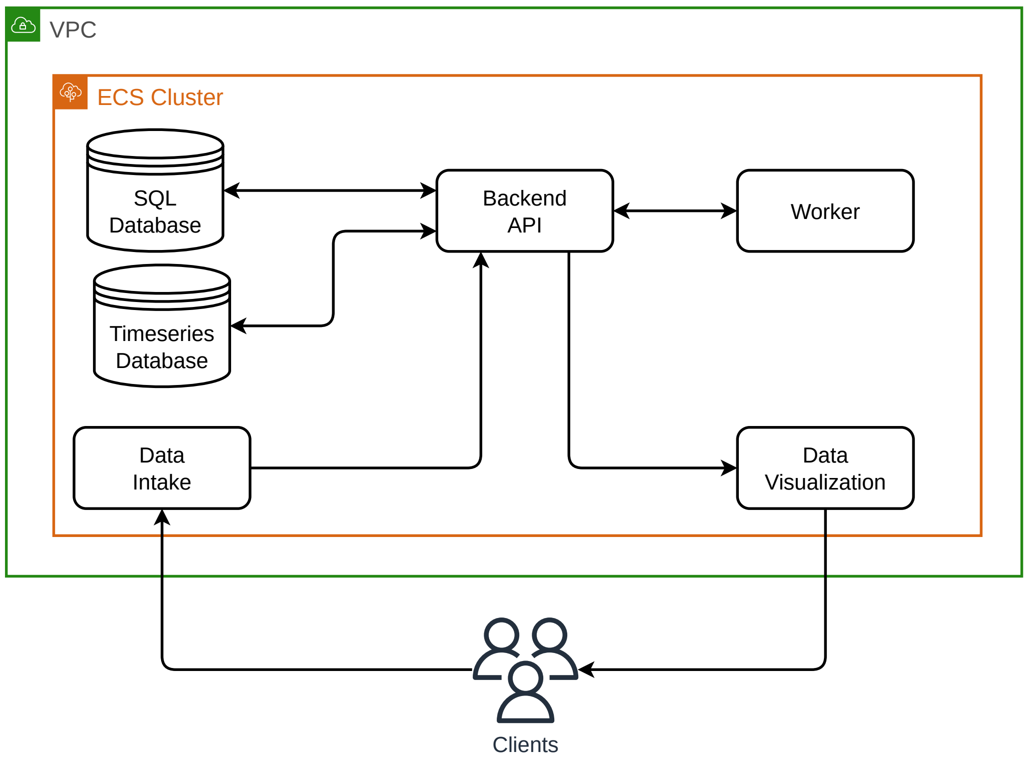

Figure 3.6: new-arch-scalable caption under figure

3.3.2.1 Automatic Container Orchestration

By leaving the orchestration of the container up to the AWS service, the services are automatically restarted in case of failure, repeated failures generate alarms and trigger actions such as reverting to the previous service version. By pushing a new service to the orchestrator, if a previous version of that service is already running and load balancing[17] is configured, then requests to that service will only be redirected to the newer version of the service when it reaches a stable state and is proven to be healthy. After proving that the new service is healthy and stable, the old service is then shutdown. This drastically reduces *downtime* for the Clients, since the transition from the old service to the new one is seamless and instantaneous and only occurs if the new service starts and maintains a steady and healthy state for a set period of time.

Additionally, there is an option to enable Blue Green[18], which would make the transition from the old service to the new one even more seamless.

3.3.2.2 Automatic Container Scaling

When using AWS ECS, resource usage metrics are gathered. When alarms and actions are configured, these metrics trigger automatic scaling up or down procedures for each

[17] Efficiently distributing incoming network traffic across a pool of available services

[18] Blue green deployment is an application release model that gradually transfers user traffic from a previous version of an app or microservice to a nearly identical new release—both of which are running in Production.

individual containerized service depending on set thresholds. This allows for quick scaling up when sudden usage spikes occur without needing human assistance and for scaling down after said spikes to avoid extra cost.

There are two types of scaling that can be performed with the containers. Vertical scaling — where the resources allocated to a container are modified which requires restart of the container, and Horizontal Scaling where the cardinality of containers for a service is modified. When defining a container, defining its CPU and RAM allocations is required. Those values are not dynamic and cannot be changed after a container is running, without restarting the container. Horizontal scaling can be performed automatically, whereas vertical scaling cannot. However, the same task can be performed by a similar container that is configured to use more resources, in which case, scaling vertically can be done automatically, by first scaling horizontally this more powerful container and shutting down the previous. There will be, however, two containers of the same task running simultaneously until the most recent task reaches a steady state and allows the older task to shut down.

3.3.3 Replaceability

As for Replaceability, the proposed architecture improves on that measurement. Using a Backend API and HTTP calls, a common interface to use between all services is created. This Backend API allows for authenticated and authorized connections to be performed between the Application's components consistently, thus ensuring that any new service or modification to an existing one doesn't require change propagation beyond that service's business and logical vertical.

3.3.4 Resiliency

Since the application is split throughout different, independent services that are managed by AWS's ECS, there are no virtual machines or EC2 instances, no Docker orchestrators that can fail. The availability of the ECS service provided by AWS is very high, and therefore, the risk of failure of the infrastructure is very, very low.

Seeing that the infrastructure is highly resilient, the remaining points of failure to improve are the Application's services themselves. As mentioned previously, the Automatic Container Orchestration service that the ECS service provides is configured to automatically restart containers in case of failure. If the failure persists, then the orchestrator alerts the DevOps team and attempts to restart the service using a previous configuration of that same service. Therefore, failures are more likely due to occur due to human error when writing code. However, the code that is put into Production is tested and thoroughly vetted before deployment to Production, which decreases even further the risk of failure in that environment.

3.3.5 Temporal Coupling

Despite the best attempts, failure can still occur. If a message is sent from a service to another while it's unavailable to receive it, then it's lost. If synchronization issues arise, then the temporal coupling that torments the old architecture can make a task fail to be executed. The cause for unavailability can be due to the service failing to start, fail during execution, not being reachable due to network misconfigurations, if it experiences manual scaling events or if it's unresponsive due to high amounts of requests to it. If a new architecture is to be drawn, then it needs to address this issue.

To that end, another version of the proposed architecture is presented in figure 3.7.

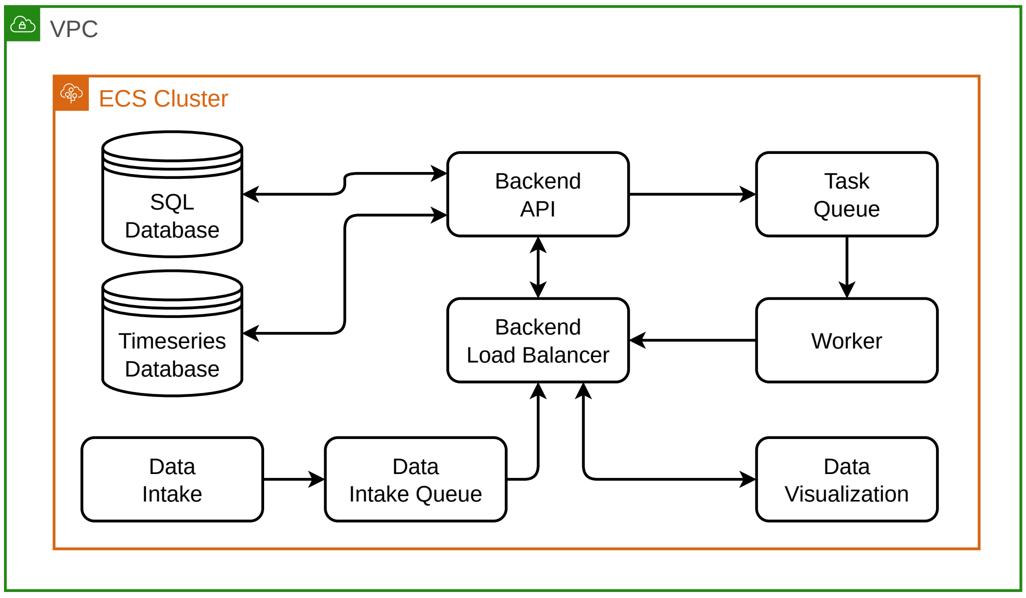

Figure 3.7: new-arch-queues caption under figure

With this new version, the Temporal Coupling has been massively reduced, the Resiliency has improved and it now allows for horizontal scalability for the services more prone to require it. By introducing queues and load balancers to the architecture, services are no longer temporally coupled and the Application can now handle interruption of some services. Messages in the queues will be delivered to the corresponding services as soon as they become available again. With this, resiliency has also improved, since the effects of the failure of a service can be minimized by having duplicate services running and a load balancer that can re-route that service's requests to an available service as can be seen in figure 3.8. It's now also more resistant to load peaks, due to the introduction of queues and load balancers. When scaling is the most adequate solution, the service is scaled and the load balancer is informed and start routing the requests to better distribute the load between the pool of service workers. If horizontal scaling is not possible, too costly or the peaks are not big enough, the queuing systems ensure that the requests are attended to as soon as possible.

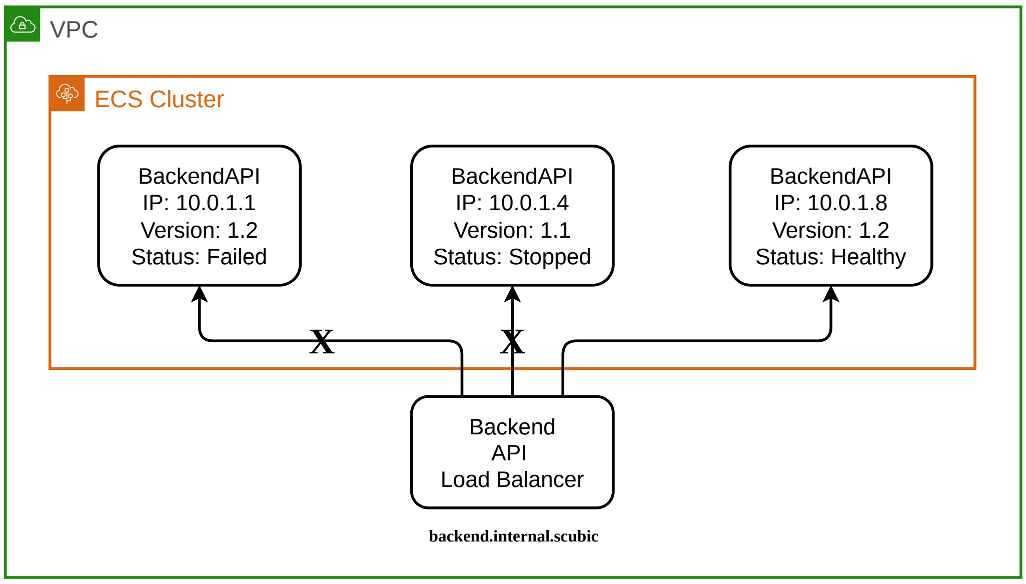

Figure 3.8: A Backend API service Load Balancer re-routing traffic towards the most recent and healthiest version of the service.

3.3.6 Implementation Coupling

As mentioned previously, in the old architecture, the implementation coupling was strong mainly due to the database access methodology. With the newest architecture, this coupling is drastically lowered, since the only way to access the databases is through HTTP calls to the Backend API. This means that implementing methods or services that require the use of data in the databases is as simple as an API call. If a new service needs a different data structure output or input to/from the database, then the changes are made in the Backend API, making that new data structure available for future use by other services as well, in case they need it.

3.3.7 Testing

The new architecture contemplates having different environments, so that a staggered deployment to Production can happen and testing can be performed in the Internal environment and then confirmed again the Staging environment. This facilitates testing the Application on real infrastructure. However, not all testing requires testing the whole Application, and for that, the new architecture allows for each one of the services to be run locally for testing and at the same time, use the services in the Application that is running in any of the Environments. In order to test a Worker service or a Backend API service, the developer needs only to execute the service to be tested locally and connect it to the VPC.

This is possible due to three new services: A Discovery Service (AWS Cloud Map[19]), a private Domain Name System (DNS) server and a private VPN server. Services created in

[19] https://aws.amazon.com/cloud-map/

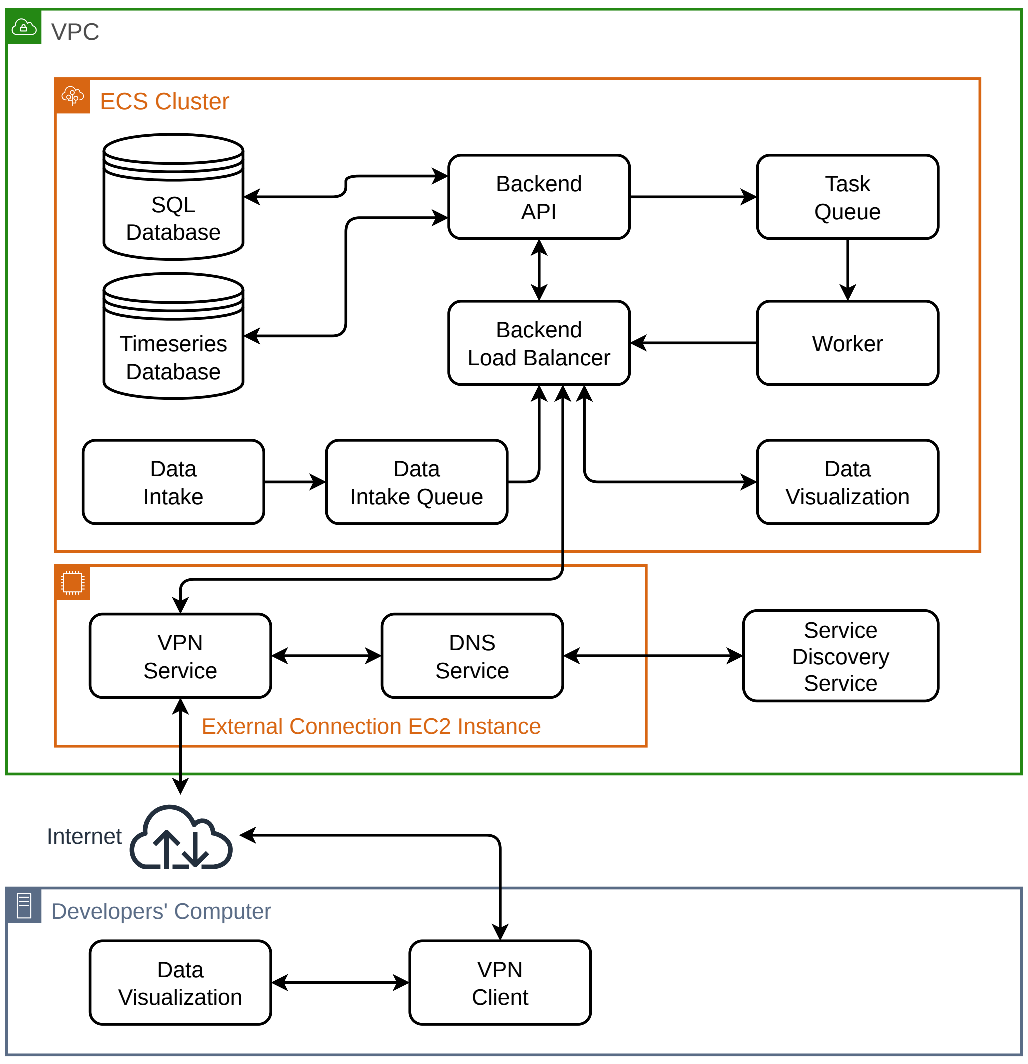

Figure 3.9: Example use of the VPN to test local Data Visualization Service with remote Backend API.

the ECS service are automatically added to the Discovery Service by AWS. These services are given human-readable names that explicit the service name and the environment as well as the domain they belong to (e.g. `backendapi.internal.scubic.pt`). This name is then picked up by a private DNS server running alongside a private VPN server. Together, these three services allow both developers and other services for seamless connection to services. If the service is scalable, then the address points not to the service itself but to the load balancer serving that address, as seen in Figure 3.9. This allows for reliable connection to the service.

This new approach uses the namespaces created by the AWS Cloud Map Discovery Service, and the naming scheme chosen was intuitive, as demonstrated in figure 3.10

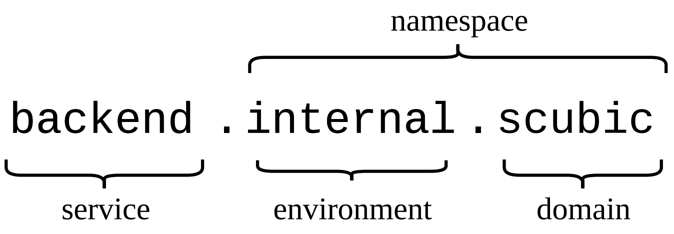

Figure 3.10: A service naming scheme example.

3.3.8 New components

3.3.8.1 Serverless

As can be seen in Figure 3.6 and Figure 3.7, the Clients send their data to the Data Intake. The data intake process is relatively simple: Take the Client's sensor data, correctly tag the data and send it to the Data Intake Queue, to be placed in the Timeseries Database. However, not all Clients are the same. Some Clients send raw text files, some send Comma-Separated Values (CSV) text files and even Microsoft Excel files. Some files are received through e-mail, others through SFTP. Those who do not send files, have given secure access to their databases through their own API. Either way, each Client has a different method to give access to their sensor data. Building a single service to deal with all possible methods of data intake is not possible and would end up a monolithic solution. Thus, the solution proposed here is to separate the data intake in general from the specific data intake from each Client, by having a service that accepts only pre-processed and tagged sensor data and inputs it into the Timeseries database, and a service for each type of data intake method. Clients using e-mail would have their data pre-processed and tagged by a different service than clients who give access to their APIs. However, having multiple services running is costly and so, these services would need to be run only when needed. A similar solution to the one used in the whole Application would be to use containers that scale up and down at set intervals. But the time that it takes for a container to be up and running is billable and superior to the average time it takes to intake data. And so, the decision was made to use AWS' Serverless Services, namely AWS

Lambda[20]. Using Lambda, the code for the data intake doesn't require to be in the form of a container, thus reducing space requirements and the time it takes for a function to be called, loaded and executed. This service is billed by the millisecond and has low compute capabilities, which make it ideal for these simple tasks.

By building and deploying several Lambda functions, one for each method of data intake, thousands of data points can be read simultaneously. However, the Timeseries database expects pre-processed data with the correct tags and measurements, so a single Data Intake Lambda Function is also created to perform the standardization and normalization of the data points and ensure the correct placement in the database. One problem that might occur is at set times, such as midnight, when most Clients send their data simultaneously. In order to avoid bottlenecks and ensure all messages are delivered, a message queue is set up between all Client data intake functions and the main Data Intake function, as shown in figure 3.11.

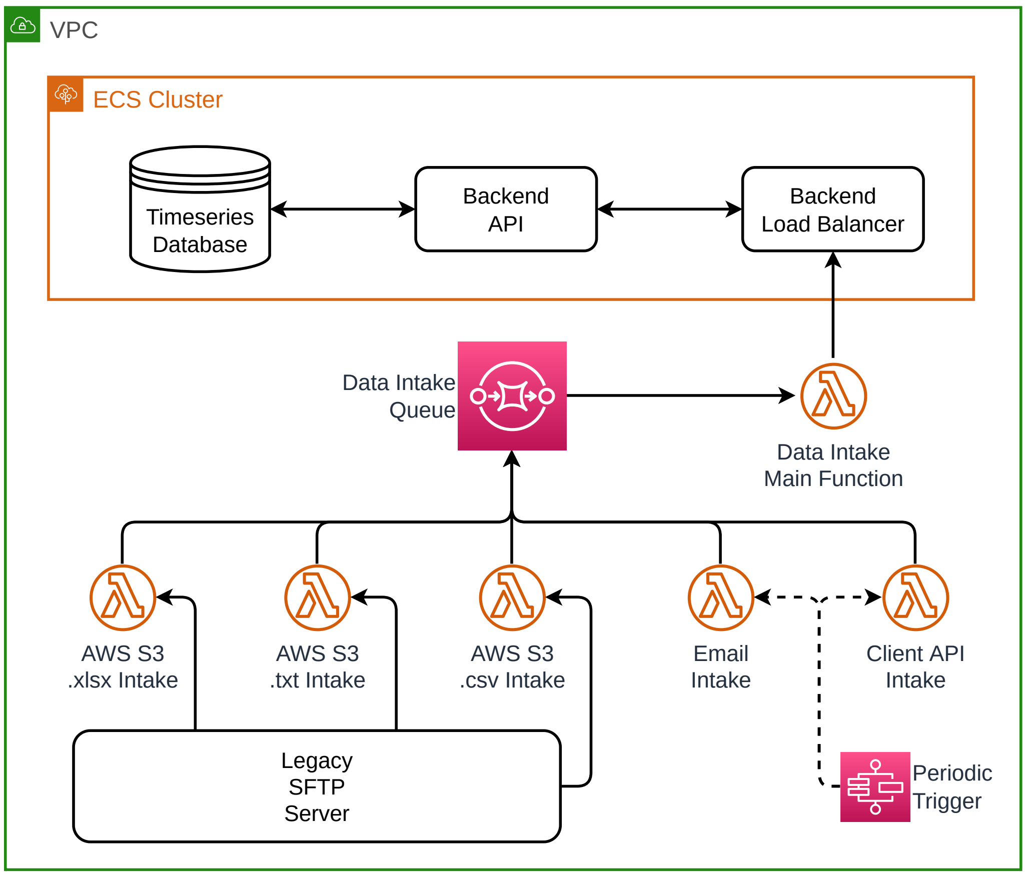

Figure 3.11: Serverless Data Intake. Client's data arrives, is pre-processed by the corresponding AWS Lambda function and sent to the Data Intake queue for further processing by a main Data Intake Lambda Function

[20] https://aws.amazon.com/lambda/

3.3.8.2 Backend API

The first element of the new architecture to be researched and produced was the Backend API. A new API solves the problem that existed with having different methods to read and write to the databases. Using this Backend API, each service that requires access to the database is therefore required to have authorization to access the Backend API, which in turn reinforces security regarding database access. Having a standardized method to access the databases also allows for easier debugging, since every service uses the sameAPIinterfaces, which can help rule out databases and the BackendAPIfrom possible fault causes.

The old architecture had a Flask API that served a webservice through which developers could manually tweak optimization and forecasting settings and issue tasks. This API, however, had no security features nor any authentication in place, with its access limited only through network settings, where each developer had to manually establish an SSH tunnel to the Client's EC2 machine in order to access said API. Due to time constraints and limited knowledge inside the company regarding securing a Flask API, research had to be performed in order to determine the best course of action regarding the choice of web framework for a Backend API.

3.3.8.3 SQL Database

After discussion with the *stakeholders* and the Chief Technology Officer (CTO), the decision to pursue a more feature-rich API such as Django was taken. Besides the many authentication, authorization and overall security features that Django includes in the base installation, its main feature is its Object-Relational Mapper (ORM). After working with Flask and the MongoDB database, Django and an SQL database such as PostgreSQL is apparently an easier option that has much more community support and the SQL database is more suited for Production environments where critical systems are used, in this case: Water Utilities's data. Since there wasn't big support for MongoDB in neither the Django official modules nor its community, the decision to change to an SQL Database was made. This was a simple decision, since the data that is present in old Clients' MongoDB databases is mostly auto-generated before the deployment is finalized, each time a deployment occurs. That meant that the old Clients' data wasn't hard to place in the new database. Using Django's ORM also means that changing data-types, changing fields associated with an object and other changes that result in database schema changes are performed automatically and each change is stored in the Django directory as a migration file. This file, when placed into the git repository of the Backend API allows for these changes to be kept and enable rollbacks in case of mishaps.

3.3.9 Workers

As a result of trying to improve the architecture, splitting each one of the tasks — Water Forecasting, Solar Power Forecasting, Weather Forecasting, Simulation, Optimization, Model Training and KPI Calculation into their own service and code repository was successful. With this, it gave more liberty for developers to work on different tasks without complicated *git* merges and also increased the resilience of the system. Where previously, if an error occurred with the Worker container it stopped all of the tasks, they are now independently orchestrated. The only exception is the innevitable temporal coupling between the Forecasting tasks and the Optimization task, since the latter requires data from the former.

3.3.10 VPN and DNS

For the VPN, the chosen implementation was to use the now popular WireGuard[21] VPN server. This was a decision taken based on (*24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017* 2017). As for the DNS server, the implementation chosen was a popular choice among the Networking community — Unbound[22], since it was only required due to regular failures with the AWS DNS server. After a full audit report by the Open Source Technology Improvement Fund, published online (2022), the decision was taken to use this implementation.

3.3.11 Observability

Increasing the old architecture's observability is not easy. After carefully studying the most popular Observability technology stacks, it was decided in conjunction with the *stakeholders* and the *CTO* that further studying of this subject was in order so as to properly implement it. Using empirical data from other companies and industry veterans, implementing the proposed Observability technology stack was deemed arduous and out of the scope of this work.

However, this new architecture relies heavily on AWS infrastructure and as such, produces a large amount of logs and metrics that are aggregated by AWS in real-time. Using AWS's own data visualization tools and the data that is currently being logged and quantified, the Observability of the new architecture has, objectively, been increased greatly, reaching the goal set in the introductory part of this document.

3.3.12 Finalized Proposed Version

After compiling all of the information in the previous sections and subsections, final diagrams that encompass the new architecture are presented hereafter. This is the new architecture that has been put into Production for the new Clients that have been gathered since this migration project started, and has been used daily since May 2022.

[21] https://www.wireguard.com/
[22] https://nlnetlabs.nl/projects/unbound/about/

3.3.12.1 Networking

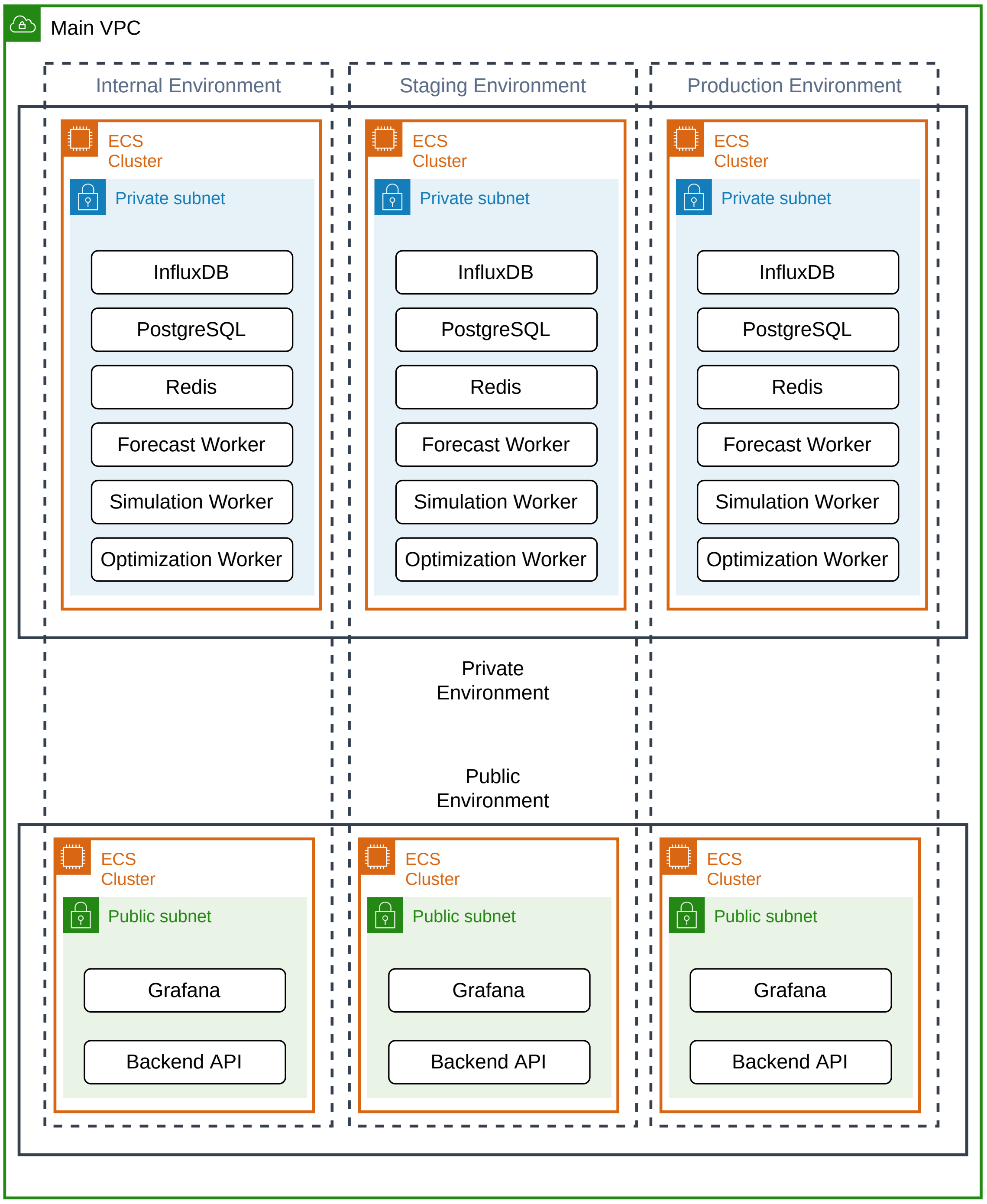

Figure 3.12: The VPC is divided in six different ECS Clusters, each with their subnet. The VPC is divided in two ways between Public and Private Environments and also divided in three parts, one for Internal use and the other two for Staging and Production Environments, respectively.

Starting with a networking and security overview, as can be seen in figure 3.12, the different services are shown in their respective matrix of environments. Note that there are no relationships between the services, as these relations have already been demonstrated in previous sections. These environments have the goal to separate the different development environments — Internal, Staging, Production — and the services which require both access to the Internet and to be accessed from the Internet. On the Public Environment, both the Grafana services and the Backend APIs are accessible from the Internet, as they are connected to the Main VPC's Internet Gateway which allows containers bidirectional communication to the Internet. In contrast, the services hosted on the Private Environment Clusters are inaccessible from the Internet and need a NAT gateway service[23] to be able to access the Internet (although only for request that originate from inside the private subnets).

Additionally, each subnet represented in figure 3.12 actually depicts a set of three subnets, each associated with a different Availability Zone[24]. Services launched into an ECS cluster are launched into any of the three available subnets for that Cluster.

3.3.12.2 Storage

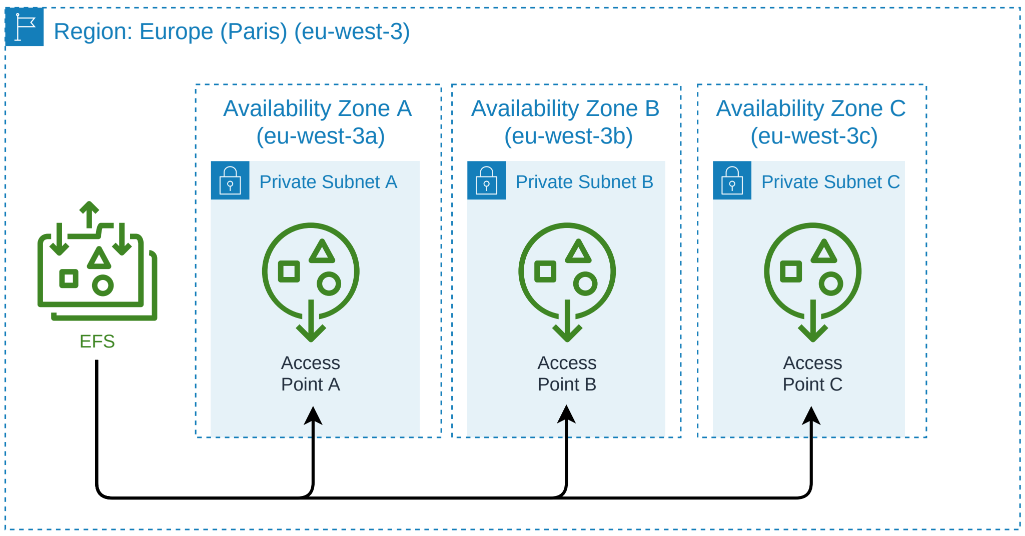
Region: Europe (Paris) (eu-west-3)
Availability Zone A (eu-west-3a)
Availability Zone B (eu-west-3b)
Availability Zone C (eu-west-3c)
Private Subnet A
Private Subnet B
Private Subnet C
EFS
Access Point A
Access Point B
Access Point C

Figure 3.13: Availability of an Elastic File System throughout multiple Availability Zones

While the containers orchestrated by AWS's ECS have 30 GB of disk space to use freely, it's ephemeral storage. If the service running in the container needs to be restarted or fails, that data is lost, hence the need for persistent storage. When the ECS cluster is composed of EC2 instances, it's possible to mount the container's volumes to the instance's block storage but such implementation is not adequate for everyday use or even Production usage. If the ECS cluster uses Fargate, such option is not even available. As such, data must be stored in an independently managed storage. The compatible AWS service that provides storage for containers in both Fargate and EC2 Clusters is AWS's Elastic File

[23] https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html

[24] Distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones.

System (EFS)[25]. As presented in figure 3.13, each EFS is accessed through one of three Access Points, one for each Availability Zone, which coincide with this architecture's subnet configuration. When a service that requires the use of an EFS is launched in ECS, the container orchestrator mounts the volume to the correspondent access point, depending on which subnet the container was launched into.

[25] https://aws.amazon.com/efs/

Chapter 4

Results and Discussion

4.1 Scalability Improvements

4.1.1 Cost Rundown

The Infrastructure used to provide the Application to the Client is not free. As such, before comparing the older and newer architectures, the costs associated with each individual item are presented in Table 4.1 and Table 4.2, respectively.

Table 4.1: Individual Cost of AWS Infrastructure used with the old architecture. Each Month corresponds to 30 days.

Name	Description	Monthly Use (h)	Hourly Cost ($/h)	Monthly Cost ($/month)
AWS EC2 t3a.large	EC2 Instance with 2 vCPU and 8 GB RAM	720	0.085	61.200
AWS EC2 t3a.xlarge	EC2 Instance with 4 vCPU with 16 GB RAM	720	0.1699	122.328
AWS EBS Volume	EBS Volume of type: gp2 with Size: 256 GB	-	-	29.696

Each Client using the old architecture would require one of the two EC2 instances presented in the Table 4.1 table, plus an EBS volume. The total expenditure with infrastructure, on a monthly basis, for a Client, is the sum of the EC2 instance and the EBS volume. Together, that value varies between $90.896 and $152.024. This value can be even higher, if the Client has an abnormal need for extra compute or memory resources, which would further reduce the scalability of this old architecture.

As mentioned in section 3.2.2.6, due to constraints in the AWS EC2 service, there is a maximum limit of 32 vCPU units for each AWS account. This means that, even if all of the Clients only required the smaller option that uses 2 vCPU, the maximum number of Clients would still be only 16 Clients. With the new architecture, there is no hard limit on the number of clients that can be served simultaneously. As of the time of writing, the new architecture is currently serving three different clients, being that one of these Clients has a water network that generates as much data as all previous Clients combined, for both the old architecture and the new one. It was also tested using 17 fictional Clients,

Table 4.2: Individual Cost of AWS ECS Fargate Containers used in the new architecture. Each Month corresponds to 30 days.

Type	Name and Description	Configuration	Daily Use (h)	Monthly Use (h)	Hourly Cost ($/h)	Monthly Cost ($/month)
A	ECS Fargate Container	0.25 vCPU	24	720	0.01215	8.748
	Very Low Power Requirements	0.5 GB RAM	24	720	0.00265	1.908
B	ECS Fargate Container	0.5 vCPU	24	720	0.02430	17.496
	Low Power Requirements	1 GB RAM	24	720	0.00530	3.816
C	AWS Fargate Container	2 vCPU	0.25	7.5	0.09720	0.729
	Medium Power Requirements	4 GB RAM	0.25	7.5	0.02120	0.159
D	AWS Fargate Container	2 vCPU	0.5	15	0.09720	1.458
	Medium Power Requirements	4 GB RAM	0.5	15	0.02120	0.318
E	AWS Fargate Container	4 vCPU	1.50	45	0.19440	8.748
	High Power Requirements	8 GB RAM	1.50	45	0.04240	1.908

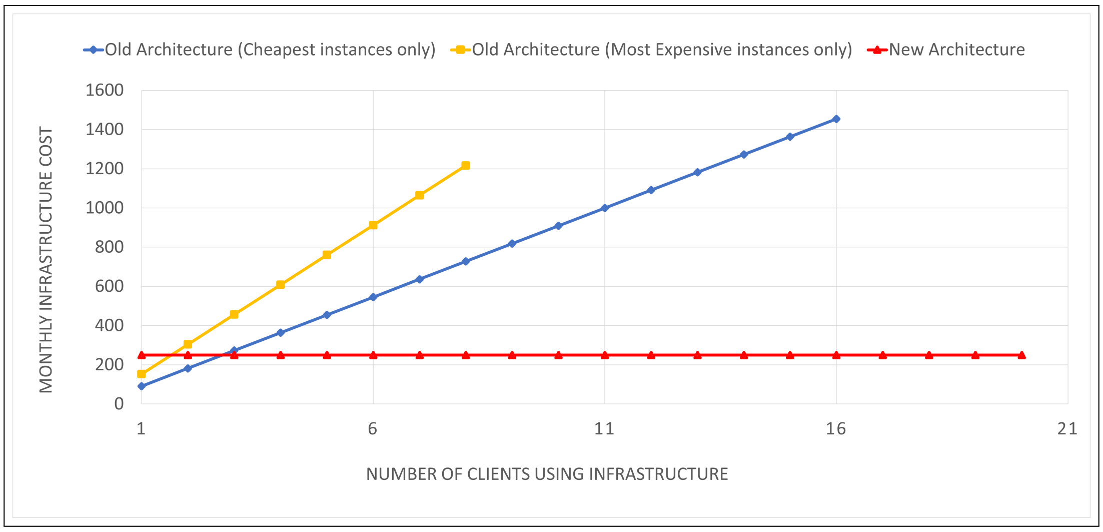

Figure 4.1: Infrastructure Cost Per Client

totaling 20 Clients being served by the Application simultaneously, with no discernible issues for the Users nor resource usages above the normal everyday operation with the three regular Clients.

The new architecture has one Application for each development environment — Production, Staging and Internal. Using Table 4.2 as reference, each Environment uses two containers of type A for the PostgreSQL database and Redis services, two containers of type B for Backend API and InfluxDB database services, one container of type C and D for the Forecasting and KPI Calculation services, respectively. Finally, type E containers are used for the Optimization service. In order to test reducing even further the resource usage for services that are running constantly, in the Internal environment, the Backend API service and InfluxDB database service have been scaled down vertically to use containers of type A. Additionally, other services are being run in this Internal environment for further testing, but are not accounted here due to being temporary services used while the Web Platform is not ready for Production.

As can be observed in figure 4.1, the monthly cost of maintaining the infrastructure needed for the Application scales linearly for the old architecture, while the new architecture's cost for low amounts of Clients is kept steady. Despite the fact that the new infrastructure has not yet been tested for more than 20 Clients, hence the data in the figure ending at 20 clients, according to the resource usage during both peak hours and idle times, these were largely unaltered, and usage difference was negligible. Therefore, the usage of the Application using the new architecture is sure to be capable of handling many more Clients simultaneously.

4.1.1.1 Remaining Costs

The costs presented here are only the main costs associated with the differences between the architectures. These are the bulk of the monthly costs with the infrastructure used. There are also costs related to Data Transfer in and out of the VPC, costs with Costumer Support, costs associated with automatic backups of data and other resources. Some resource usages are not reported in this document due to Client confidentiality agreements, being free of charge or being used so sparingly or in low quantities that the costs are irrelevant or included in the Free Tier[1].

4.1.2 Development and Deployment Issues

Besides the infrastructural scaling issue, there was also a problem scaling the human resources needed for development. Having the Application be the same for all of the Clients, instead of being one separately maintained Application for each Client, drastically lowers the developer manpower needed for maintaining and updating an Application for multiple Clients. Therefore, this new architecture is substantially and clearly scalable when compared with the previous architecture.

[1] Certain AWS resources are free until a certain level of usage is achieved.

4.2 DevOps Improvements

The Company's developer team's general sentiment towards the implementation of the new architecture is one of relief and excitement. As stated in Section 3.2.2.4 and section 3.2.2.5, there were several issues regarding the developer workflow when working in the Application. After implementing the new architecture, these concerns have diminished considerably. The measures taken in section 3.3.1 and Section 3.3.7 have brought a new and better workflow to the development team. This can be attested by the results in the FKM.

4.3 Observability Improvements

Here, show some dashboards of the observability system, exemplifying normal day-to-day operation, error detection, etc...

Chapter 5

Conclusion

5.1 Final Considerations

5.2 Future Work

References

(2022).
URL: https://ostif.org/our-audit-of-unbound-dns-by-x41-d-sec-full-results/ (cit. on p. 41).

24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017 (2017). The Internet Society.
URL: https://www.ndss-symposium.org/ndss2017/ (cit. on p. 41).

Adams, Patton, Omar Al-Shahery, Joseph Chmiel, Amy Cunliffe, Molly Day, Oliver Fay, Charlie Gardner, Gian Luca Giuliani, Samuel Goddard, and Larry et al. Karl (2022). *2020 CYBER THREATSCAPE REPORT*.
URL: https://www.accenture.com/_acnmedia/PDF-136/Accenture-2020-Cyber-Threatscape-Full-Report.pdf (cit. on p. 7).

Ali, Abdulrazzaq, Abu Bakar Md Sultan, Abdul azim abdul ghani, and Hazura Zulzalil (Sept. 2017). "in Critical Issues Across SAAS Development: Learning from Experience CRITICAL ISSUES ACROSS SAAS DEVELOPMENT: LEARNING FROM EXPERIENCE." In: pp. 2393–2835. (Cit. on p. 6).

Aljamal, Rawan, Ali El-Mousa, and Fahed Jubair (2018). "A comparative review of high-performance computing major cloud service providers." In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. DOI: 10.1109/iacs.2018.8355463. (Cit. on p. 6).

Alnumay, Waleed (2020). "A brief study on Software as a Service in Cloud Computing Paradigm." In: *Journal of Engineering and Applied Sciences*, pp. 1–15. DOI: 10.5455/jeas.2020050101. (Cit. on pp. 5, 6).

Bass, Len, Ingo Weber, and Liming Zhu (2015). *DevOps*. Addison-Wesley. (Cit. on p. 14).

Candela, Ivan, Gabriele Bavota, Barbara Russo, and Rocco Oliveto (2016). "Using Cohesion and Coupling for Software Remodularization." In: *ACM Transactions on Software Engineering and Methodology* 25.3, pp. 1–28. DOI: 10.1145/2928268. (Cit. on p. 9).

Cavusoglu, Hasan, Huseyin Cavusoglu, and Jun Zhang (2008). "Security Patch Management: Share the Burden or Share the Damage?" In: *Management Science* 54.4, pp. 657–670. DOI: 10.1287/mnsc.1070.0794. (Cit. on p. 7).

Chen, Rui, Shanshan Li, and Zheng Li (2017). "From Monolith to Microservices: A Dataflow-Driven Approach." In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. DOI: 10.1109/apsec.2017.53. (Cit. on p. 10).

Dillon, Tharam, Chen Wu, and Elizabeth Chang (2010). "Cloud Computing: Issues and Challenges." In: pp. 27–33. DOI: 10.1109/AINA.2010.187. (Cit. on p. 6).

Eugene Wiegers, Karl and Joy Beatty (2013). *Software Requirements.* 3rd ed. Microsoft Press, U.S., pp. 4–23. (Cit. on p. 8).

Forsgren, Nicole, Dustin Smith, Jez Humble, and Jessie Frazelle (2019). *2019 Accelerate State of DevOps Report.* Tech. rep.
URL: http://cloud.google.com/devops/state-of-devops/ (cit. on p. 14).

Glenn, Ashton (2018). "Equifax: Anatomy of a Security Breach." PhD thesis. Georgia Southern University. (Cit. on p. 7).

Gopal, Madan (1993). *Modern control system theory.* New Age International. (Cit. on p. 15).

Hasselbring, Wilhelm (2018). "Software architecture: Past, present, future." In: *The Essence of Software Engineering.* Springer, Cham, pp. 169–184. (Cit. on pp. 8, 9).

IDC (2019). *IDC FutureScape: Worldwide IT Industry 2019 Predictions.* (Cit. on p. 12).

IDC (2021). *IDC FutureScape: Worldwide IT Industry 2021 Predictions.* (Cit. on p. 5).

"ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary" (2017). In: *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712. (Cit. on p. 7).

Kim, Won (2009). "Cloud Computing: Today and Tomorrow." In: *The Journal of Object Technology* 8.1, p. 65. DOI: 10.5381/jot.2009.8.1.c4. (Cit. on p. 5).

Lewellen, Stephanie (2020). "Identifying Key Stakeholders as Part of Requirements Elicitation in Software Ecosystems." In: *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B* B, pp. 88–95. DOI: 10.1145/3382026.3431249.
URL: https://dl.acm.org/doi/pdf/10.1145/3382026.3431249 (cit. on p. 8).

Marks, Eric A and Michael Bell (2008). *Service-oriented architecture: a planning and implementation guide for business and technology.* John Wiley & Sons. (Cit. on p. 8).

Martin, Robert C. (2014). *Agile Software Development, Principles, Patterns, and Practices: Pearson New International Edition.* 1st ed. Pearson. (Cit. on p. 12).

Mell, P M and T Grance (2011). "The NIST definition of cloud computing." In: DOI: 10.6028/nist.sp.800-145. (Cit. on pp. 5, 6).

Mills, Everald E (1988). *Software metrics.* Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. (Cit. on p. 16).

Newman, Sam (2015). *Building Microservices.* 1st ed. O'Reilly Media. (Cit. on p. 12).

Newman, Sam (2019). *Monolith to microservices: evolutionary patterns to transform your monolith.* O'Reilly Media. (Cit. on pp. 1, 10, 11).

Niedermaier, Sina, Falko Koetter, Andreas Freymann, and Stefan Wagner (2019). "On Observability and Monitoring of Distributed Systems – An Industry Interview Study." In: *Service-Oriented Computing.* Ed. by Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari. Cham: Springer International Publishing, pp. 36–52. ISBN: 978-3-030-33702-5. (Cit. on p. 15).

Niknejad, Naghmeh, Waidah Ismail, Imran Ghani, Behzad Nazari, Mahadi Bahari, and Ab Razak Bin Che Hussin (2020). "Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation." In: *Information Systems* 91, p. 101491. DOI: 10.1016/j.is.2020.101491. (Cit. on p. 8).

OpenTelemetry (2022).
URL: https://opentelemetry.io/docs/concepts/observability-primer/ (cit. on p. 15).

Pacheco, Carla, Ivan García, and Miryam Reyes (2018). "Requirements elicitation techniques: a systematic literature review based on the maturity of the techniques." In: *IET Software* 12.4, pp. 365–378. DOI: 10.1049/iet-sen.2017.0144. (Cit. on p. 8).

Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules." In: *Communications of the ACM* 15.12, pp. 1053–1058. DOI: 10.1145/361598.361623. (Cit. on p. 9).

Rapid7 (2018). *Security Report for In-Production Web Applications.*
URL: https://www.rapid7.com/globalassets/_pdfs/whitepaperguide/rapid7-tcell-application-security-report.pdf (cit. on p. 7).

Rezaei, Reza, Thiam Kian Chiew, Sai Peck Lee, and Zeinab Shams Aliee (2014). "A semantic interoperability framework for software as a service systems in cloud computing environments." In: *Expert Systems with Applications* 41.13, pp. 5751–5770. DOI: 10.1016/j.eswa.2014.03.020. (Cit. on p. 5).

Sallin, Marc, Martin Kropp, Craig Anslow, James W. Quilty, and Andreas Meier (June 2021). "Measuring software delivery performance using the four key metrics of DevOps." In: *Lecture Notes in Business Information Processing*, pp. 103–119. DOI: 10.1007/978-3-030-78098-2_7. (Cit. on p. 14).

Sommerville, Ian and Pete Sawyer (1997). *Requirements engineering.* Wiley, pp. 4–5. (Cit. on p. 8).

Tal, Liran (2022). *Alert: peacenotwar module sabotages npm developers in the node-ipc package to protest the invasion of Ukraine | Snyk.*
URL: https://snyk.io/blog/peacenotwar-malicious-npm-node-ipc-package-vulnerability/ (cit. on p. 9).

Appendices

Appendix A

Appendix example

This is the first appendix.

A.1 A section example

Similarly to a chapter we can add sections, subsections, and so on...

Appendix B

A second example of an appendix

This is the second appendix.