DOCUMENTO PROVISÓRIO

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

DOCUMENTO PROVISÓRIO

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

List of figures

List of tables

List of abbreviations

API . . . . . Application Programming Interface
AWS . . . . . Amazon Web Services

CAPEX . . . . . Capital Expenditure
CI/CD . . . . . Continuous Integration/Continuous Deployment

DSS . . . . . Decision Support System

EC2 . . . . . Elastic Compute Cloud
EIP . . . . . Elastic IP
ENI . . . . . Elastic Network Interface

HTTPS . . . . . Secure Hypertext Transfer Protocol

KPI . . . . . Key Performance Index

NIST . . . . . National Institute of Standards and Technology

OPEX . . . . . Operational Expenditure
OTel . . . . . OpenTelemetry

SaaS . . . . . Software-as-a-Service
SCADA . . . . . Supervisory Control And Data Acquisition
SDK . . . . . Software Development Kit
SFTP . . . . . SSH File Transfer Protocol
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

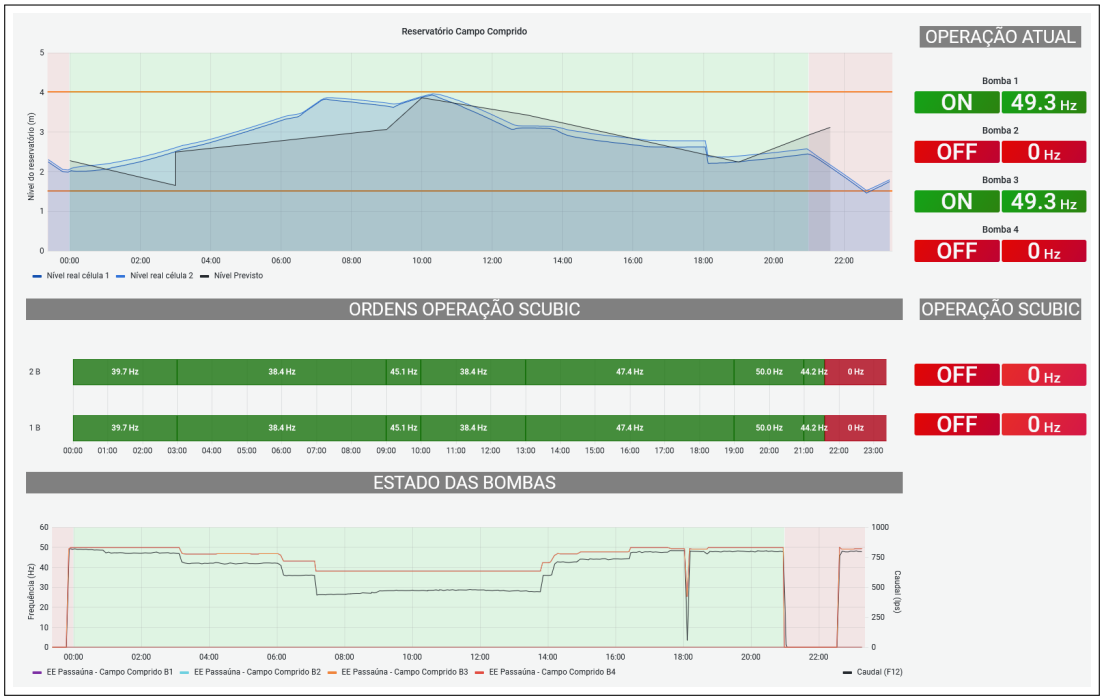

Figure 1.1: Example of a DSS interface from one of SCUBIC's Clients

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 Section 3.2.1, a complete analysis of this architecture is provided and explained in detail.

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 and scalability in comparison to the old DSS Architecture. The new architecture software 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 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.

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 regarding the state of software architecture and cloud-based software solutions is presented. It's divided in three sections, starting with some insight into how Cloud Computing and Software-as-a-Service (SaaS) impact the software landscape, a second part that will reflect on how Observability can improve software development and lastly, how Software Architectures are planned, executed and then analyzed. Some text regarding the general technologies used throughout the work is also analyzed here.

Chapter 3 is divided into multiple sections. Firstly, a more detailed explanation of the old architecture and its inherent flaws is presented, flaws which end up showcasing the need for a new and improved software architecture. The second section is related to the first step when engaging a new engineering project: requirements. In this section, the goals for the new architecture are laid out along the multiple constraints that are in place throughout the whole execution of the work. Here, the methodology to be adopted for this work is presented as well. In a third section, the plans for implementing the new architecture are laid out chronologically. Lastly, the final section is related to the actual implementation of the proposed software architecture. The procedures taken, the challenges and decisions made throughout the implementation are shown and contextualized in this section. In here, the finalized architecture is shown with the help of diagrams.

Chapter 4 analyzes if the new architecture complies with the restraints imposed by the stakeholders, achieves the required and desired results and how it compares with the old architecture. In a first part, the functional requirements are analyzed, followed by the non-functional requirements and overall feedback from the development team that accompanied this software architecture migration. Then, a methodology for measuring the performance indexes is presented. Finally, a cost analysis is made for both the recurring monetary infrastructure costs and overall impact on team productivity.

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 service over normal Internet access (Alnumay, 2020). It can also be understood, as the U.S. National Institute of Standards and Technology (NIST) indicates as: *Cloud computing is 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).

Incorrectly used as a synonym of on-demand computing, grid computing or even 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

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.

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.

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.

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.

Hybrid Cloud

This deployment model is a combination of one or more of the previous deployment models, where data transfer or task handling can ocour 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 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.

2.3.1 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.

2.3.2 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)

2.4 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.

2.4.1 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)

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). Requirements elicitation should be performed during the early stages of the software planning phase in order to prevent

2.5 Software Architecture

((Que tipos de arquitecturas de software existem))

2.6 Code Deployment

2.6.1 DevOps

The *portmanteau* of Development and Operations - DevOps - is as "(...) 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). There are, therefore, three things to retain from this definition. The two time periods where Development and Deployment occur, the quality of the changes to be committed to a system and the quality of the processes of putting those changes into production.

Development Time

This time
Sallin et.al. (Sallin *et al.*, 2021)

2.7 Cloud-Based

2.7.1 Evaluating Architectures

Chapter 3

Methodology

3.1 *Stakeholders*

The first step when handling a Software Engineering problem is to identify the key *stakeholders*. For this project, the key stakeholders are all the people involved in the company and the Clients' project managers. In the company, SCUBIC, the teams are divided into the development team, the executive team and the operations team, where their members are part of one or more of them.

3.2 The Old Architecture

3.2.1 Old Architecture Components

The current software architecture is still in use as of the date of publication of this body of work. This older architecture consists of an amalgamation of Docker containers, each running a different service. Each Client has its own VPS wherein these Docker containers are deployed.

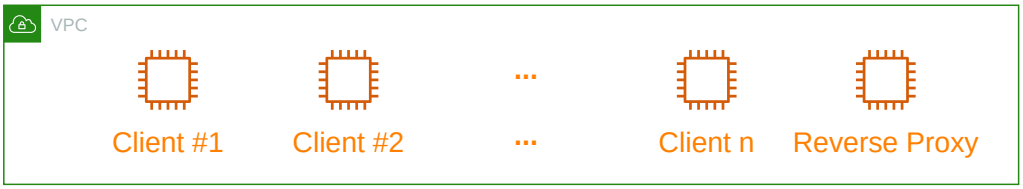

Figure 3.1: The AWS VPC used, hosting the old architecture's EC2 VPS

VPC

These VPS are general-purpose Amazon Web Services (AWS) Elastic Compute Cloud (EC2) *Instances*. 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. The Reverse Proxy serves as, as the name implies, as a reverse proxy to enable the use of a single

Elastic IP (EIP), a single Elastic Network Interface (ENI) by all Clients's servers, since the availability of public IPs is limited to five EIP.

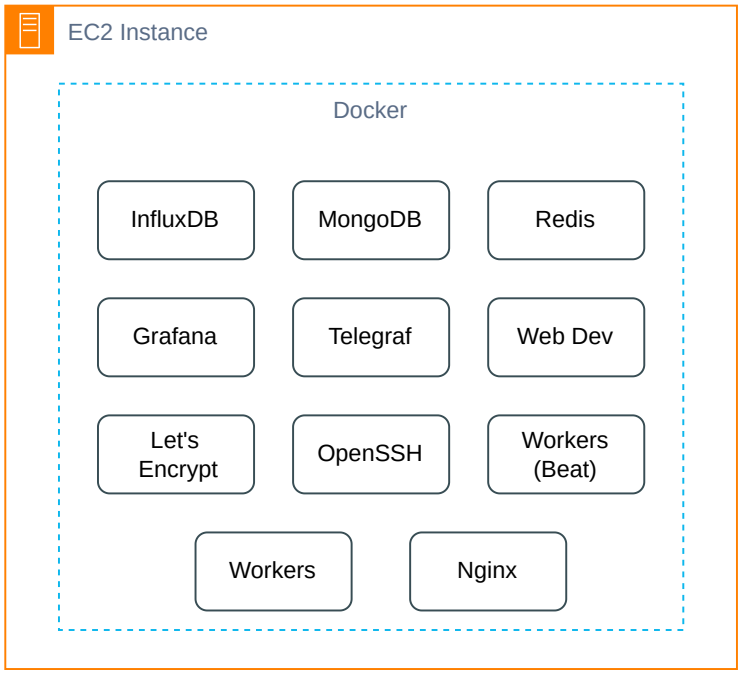

Figure 3.2: A singular AWS EC2 instance, hosting the old architecture's docker containers

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

- **InfluxDB** (Timeseries Database)
- **MongoDB** (General use, no-SQL, Document Database)
- **Grafana** (Web platform for data visualization, the front end of the DSS (Chakraborty and Kundan, 2021))
- **Telegraf** (Data collecting service)
- **Nginx** (Reverse proxy with Secure Hypertext Transfer Protocol (HTTPS) capabilities)
- **Let's Encrypt** (Automatic Transport Layer Security (TLS) Certificate installer, companion for the Nginx container)
- **Web Dev** (Web platform / API for managing Workers' settings)
- **Redis** (Message Queue System for queuing Worker's jobs)
- **OpenSSH** (*atmoz/sftp*) (Secure Shell (SSH) Server for receiving client data through SSH File Transfer Protocol (SFTP))

- **Workers** (Container running the Forecast, Simulation and Optimization Python Algorithms as well as the KPI Algorithms.)
- **Workers** (*Beat*) (Container that periodically *triggers* jobs in the Workers container)

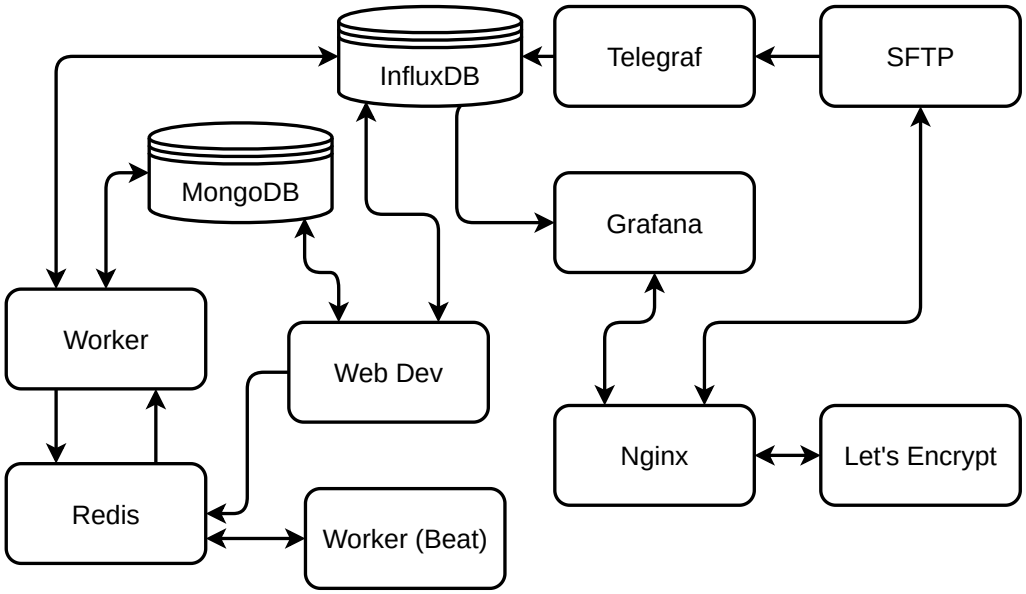

Figure 3.3: old-arch-connections caption under figure

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.

Grafana

This web platform allows the visualization of the Timeseries data from the **InfluxDB** database. This is a freely-available 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. This is used in both the new and old architecture, since the new visualization platform is still not operational (not within the scope of this body of work).

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.

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 does the file intake.

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 ENI associated to it exclusively. This ENI is then connected, exclusively, to a single EIP to which the Clients connect, like Figure 3.4 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.

Redis

Redis (*Introduction to Redis* 2022) is used as a message queue backend for Celery (*Distributed task queue* 2022), enabling other services to send Celery tasks to a queue for asynchronous execution by the Workers.

Web Dev

Based on Flask (*Flask* 2022), 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

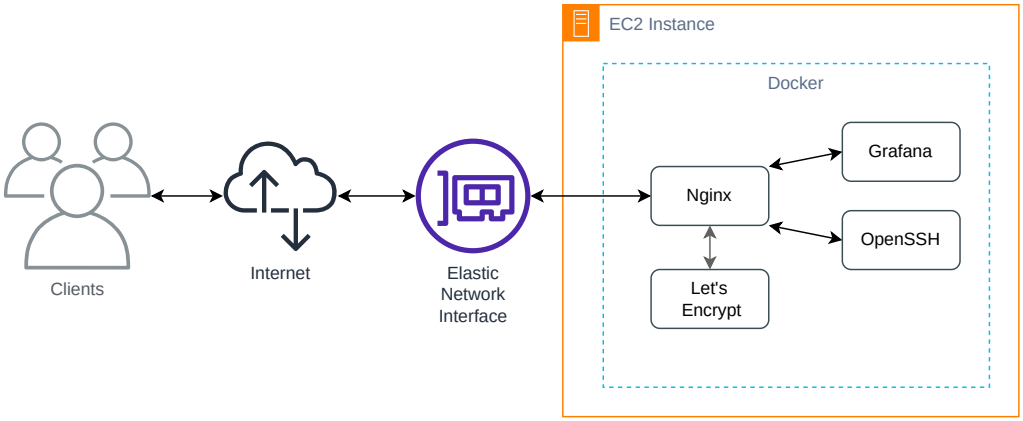

Figure 3.4: Internet access to the Client-facing services

Celery tasks to the queue. This application connects directly to both databases.

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 it 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.

In Figure 3.3 the relations between these containers can be (subsequently?) 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.

3.3 Issues

3.3.1 Resource Sizing

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 3.5, 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.

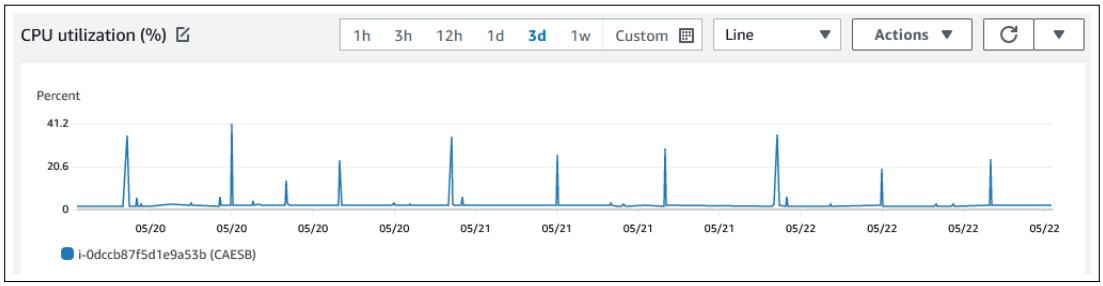

Figure 3.5: Client's EC2 Instance average CPU usage, during a three-day period, in 5 minutes intervals

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.

Therefore, one of the goals of this work is to attempt to solve this problem. One of the possible general 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 AWS account.

3.3.2 Limited Compute Resources

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.

3.3.3 Individual Codebases

Besides an individual EC2 instance, each Client also has an individual GitLab (*The One DevOps platform* 2022) project, which is composed of several, and different Git (Spinellis, 2012) 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)

- **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 Continuous Integration/Continuous Deployment (CI/CD) pipeline that deploys the old architecture to the instances also resides here.

As for the **DBconnectors** repository, database connectors can be found. 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.

The readers notice that, as shown both above and on Figure 3.3, 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 when a different querying service placed 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.(citation *still* needed)

3.3.4 Observability

One of the issues with the old architecture was the lower Observability(citation needed) 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.

Alerts

A consequence of the old architecture's lack of system observability, there were no useful metrics being created and store besides the ones pertaining to the algorithm result. Metrics are required in order to, having a set of thresholds for each one of them, produce alarms. 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 Tank's water level that would send messages to a Slack channel shared between the company and the respective Client, but fell into disuse.

3.3.5 Deployment

The process of deploying new functionality or code fixes to Client's servers that use the old architecture can quickly become a multi-hour endeavor. Despite being a somewhat modular architecture, given that each service has its own docker container, they are dependent on each other when initializing the containers. On some deployment procedures, namely when changing code in the Workers container, it requires updating the Workers' docker image, running GitLab's CI/CD pipelines for this deployment of the Workers' docker image and then tag a completely different repository — **dbs** — inside the same Client's GitLab project so that it triggers another CI/CD pipeline which replaces all the containers within the Client's EC2 instance with the *latest* version of each service's container image. Although the last step, which replaces the containers is performed rather quickly and the apparent downtime for the Client is minimal, the amount of time for the image building in the CI/CD process is cumbersome, reaching a combined time of 20 to 30 minutes on average. This chaotic and time-consuming process leads to lack of motivation for the development team to introduce new features regularly. This leads to lower Deployment Frequency, increased Lead Time for Change and Time to Restore Service (when a deployment or unexpected bug occurs). Additionally, the complexity and tight-coupling of services leads to increase Change Failure Rate.

Singular Environment

One of the faults with the older architecture was the lack of different environments for deployment. Such fact meant that every deployment made to each Client had 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 was 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 made the repeatability of results from day to day not trivial. Breaking changes were also not always apparent, since some algorithms performed calculations using data generated by other algorithms and/or real-time data

and such mistakes only became apparent on the following work day, after their execution. There were cases when the algorithms ran perfectly during week days, but failed during the weekends (since the water consumption patterns change accordingly).

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 for results to be monitored for longer periods of time 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.

3.4 Requirements

"The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended"

3.5 Planning the Architecture Migration

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 would bring several challenges, which were compounded by the lack of a functional new web interface for the new architecture. For this migration to occur, careful planning had to be done and checked by the *stakeholders* before any changes were put into production. Measures such as changing network configurations, restarting services or run benchmarks on the old infrastructure could not affect any Clients using the old infrastructure.

3.5.1 Changing a car's wheel while driving

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

shaped 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 and

3.6 New Architecture

Based on the recently elicited requirements and the issues with the old architecture, a generic plan for the new architecture is defined as follows on Figure 3.6

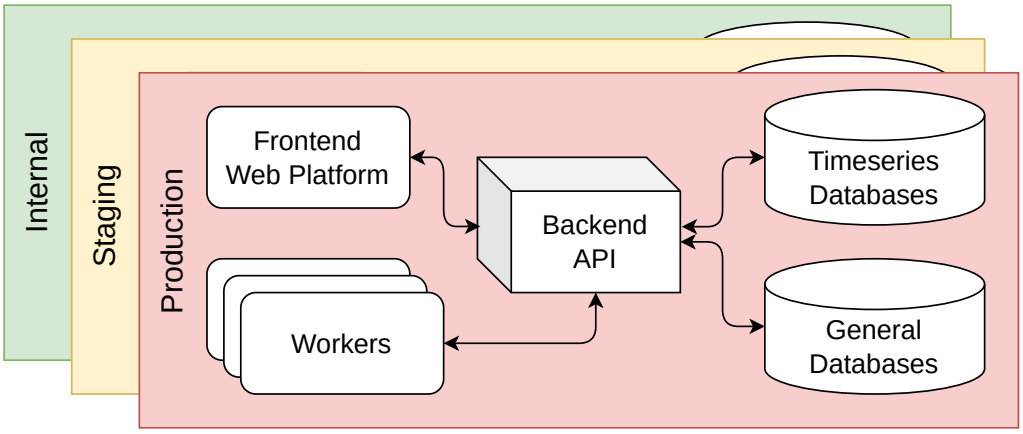

Figure 3.6: new-arch-basic caption under figure

3.6.1 Implementing the Architecture

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.

Backend API

Serverless

Chapter 4

Results and Discussion

Chapter 5

Conclusion

References

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. 10).

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).

Chakraborty, Mainak and Ajit Pratap Kundan (2021). "Grafana." In: *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, pp. 187–240. ISBN: 978-1-4842-6888-9. DOI: 10.1007/978-1-4842-6888-9_6.
URL: https://doi.org/10.1007/978-1-4842-6888-9_6 (cit. on p. 12).

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).

Distributed task queue (2022).
URL: https://docs.celeryq.dev/en/stable/ (cit. on p. 14).

Eugene Wiegers, Karl and Joy Beatty (2013). *Software Requirements*. 3rd ed. Microsoft Press, U.S., pp. 4–23. (Cit. on p. 9).

Flask (May 2022).
URL: https://palletsprojects.com/p/flask/ (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. 7).

Introduction to Redis (Mar. 2022).
URL: https://redis.io/docs/about/ (cit. on p. 14).

"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. 9).

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. 9).

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. 8).

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

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. 7).

OpenTelemetry (2022).
URL: https://opentelemetry.io/docs/concepts/observability-primer/ (cit. on p. 8).

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. 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 com-

puting 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. 10).

Sommerville, Ian and Pete Sawyer (1997). *Requirements engineering*. Wiley, pp. 4–5. (Cit. on p. 9).

Spinellis, Diomidis (2012). "Git." In: *IEEE Software* 29.3, pp. 100–101. DOI: 10.1109/MS.2012.61. (Cit. on p. 17).

The One DevOps platform (May 2022).
URL: https://about.gitlab.com/ (cit. on p. 17).

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.