DOCUMENTO PROVISÓRIO

**Carlos Manuel
Basílio Oliveira**

**Arquitectura de Software Escalável para
Sistemas de Apoio à Decisão para Entidades
Gestoras de Àgua**

**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 Àgua

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

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

Resumo O fornecimento de àgua às populações é um símbolo de qualquer grande sociedade, desde o início da Civilização. Hoje em dia, enormes quantidades de àgua são fornecidas constantemente a residências e indústrias variadas utilizando motores eléctricos acopolados a bombas de àgua que consomem vastas quantidades de energia eléctrica. Com o recurso a tarifas de electricidade variáveis e dinâmicas, dados em tempo real de variados sensores nas empresas de fornecimento de àgua e a modelos da rede de distribuição de àgua, o software da SCUBIC consegue monitorizar e prever consumos de àgua e assim optimizar a operação destas bombas por forma a baixar os custos operacionais das empresas gestoras de àgua.

O software fornecido pela SCUBIC é uma amálgama de serviços construídos numa fase embrionária da empresa, que por se manterem inalterados ao longo dos anos, não se adequam mais ao plano de negócios e escalada 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, foi possível alterar a arquitectura do software usado pela SCUBIC. 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

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

SCADA . . . . Supervisory Control And Data Acquisition
SFTP . . . . SSH File Transfer Protocol
SSH . . . . Secure Shell

TLS . . . . Transport Layer Security

vCPU . . . . Virtual CPU
VPC . . . . Virtual Private Cloud
VPS . . . . Virtual Private Server
VSD . . . . Variable-Frequency Drive

WSS . . . . Water Supply Systems
WU   . . . . Water Utilities

Chapter 1

Introduction

This 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.

1.1   Water Supply Systems

The water supply systems that are prevalent in our society play a very important role in our daily lives, distributing water throughout the country from water reservoirs or water treatment plants up until it reaches our houses and industries. These Water Supply Systems (WSS) can be quite complex and difficult to manage without proper processes that ensure the operation of such networks is made without problems, in an environmental and economically sustainable way. For this reason, the use of specialized software to aid operators or even automatically control the operation of these WSS is of uttermost importance nowadays. Water, be it in quantity and quality, 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 enormous WSS requires the use of vast quantities of mechanical work, which in turn requires lots of energy, namely, electric energy. With the ever-growing political, economic and environmental pressure to improve and optimize how we use energy, and with the current geopolitical issues, 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 lower energy use as well, is growing within Water Utilities (WU).

1.2   Existing Decision Support System

In order for the WU's to 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. 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 at like shown in Figure 1.1
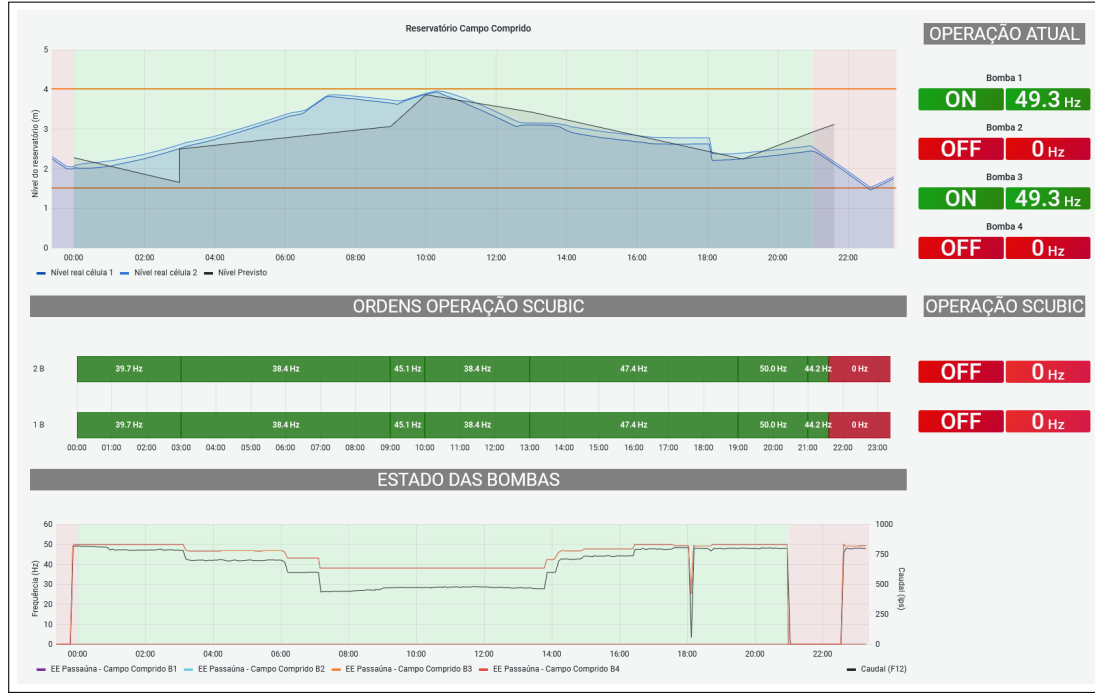

Figure 1.1: Example of a DSS interface that uses *Grafana*.

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 Private Server (VPS), one for each Client, where a set of Docker containers contain (containers contain?) 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 and as such there is an unsurmountable amount of code drift between the same services of the different clients. This is, quite apparently, not scalable nor even remotely manageable for any software development team. Further ahead, on Section 3.1.1, a complete analysis of this architecture will be 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 better, improved software, considering the requirements from the stakeholders. This new architecture hopes to improve the performance, reliability, resilience, security and scalability of the old DSS Architecture. A new, better 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's also hoped to save monetary resources 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 might mean a more optimal use of compute resources, resulting in lowering such costs.

1.4 Structure of the Document

This document is composed by a total of X chapters.

In Chapter 1, the reader is presented with a summary of the work to be done as well as the context behind such decision.

In Chapter 2, a bibliographical analysis regarding the state of software architecture and cloud-based software solutions is presented. It's divided in two sections, starting with some insight into how a Software Architecture is 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. There wasn't a single plan because the requirements and constraints kept changing during the planning and implementation part of the work. 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. (Where can/should I introduce the methodology for measuring the performance indexes and other benchmarks that analyze the new architecture?)

Chapter 4 is where we can see whether 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, we analyze the functional requirements, followed by the non-functional requirements and overall feedback from the development team that accompanied this software architecture migration. 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 archi-

tecture.

Chapter 2

State-of-the-Art

2.1   Software Engineering

2.1.1   Defining Requirements

Here, we will show what's the state-of-the-art regarding requirements definition. This is an important step in Software Engineering, or any Engineering.

2.2   Software Architecture

2.3   Code Deployment

2.3.1   What is 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)

2.4   Cloud-Based

Chapter 3

Methodology

3.1 The Old Architecture

3.1.1 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.

Virtual Private Cloud (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.
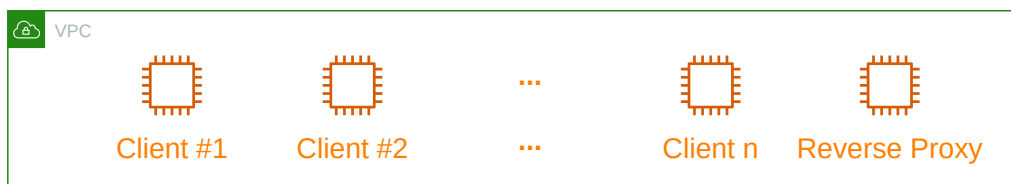

Figure 3.1: The AWS VPC used, hosting the old architecture's EC2 VPS

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)

7

- **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 Key Performance Index (KPI) Algorithms.)

- **Workers** (*Beat*) (Container that periodically *triggers* jobs in the Workers container)
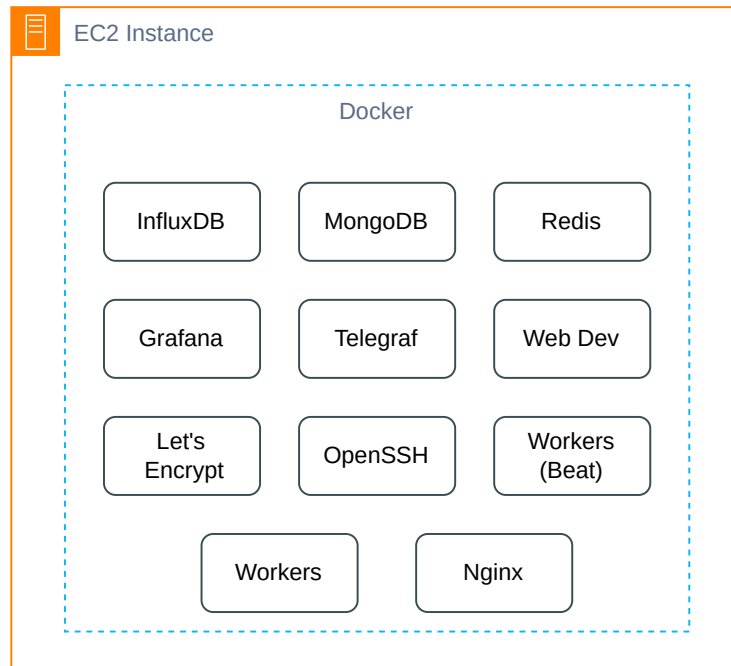

Figure 3.2: A singular AWS EC2 instance, hosting the old architecture's docker containers

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 3.3 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.
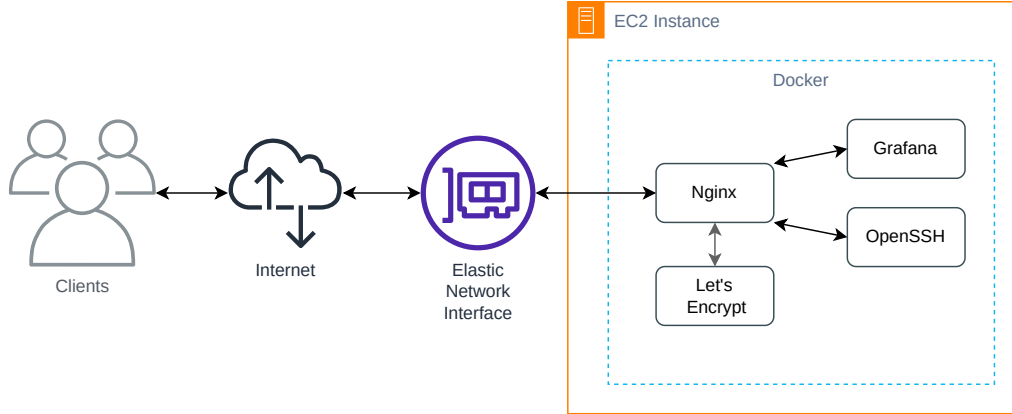

Figure 3.3: Internet access to the Client-facing services

Redis

We use Redis (*Introduction to Redis* 2022) 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 Application Programming Interface (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.

Workers

This is where the *magic* happens. 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.
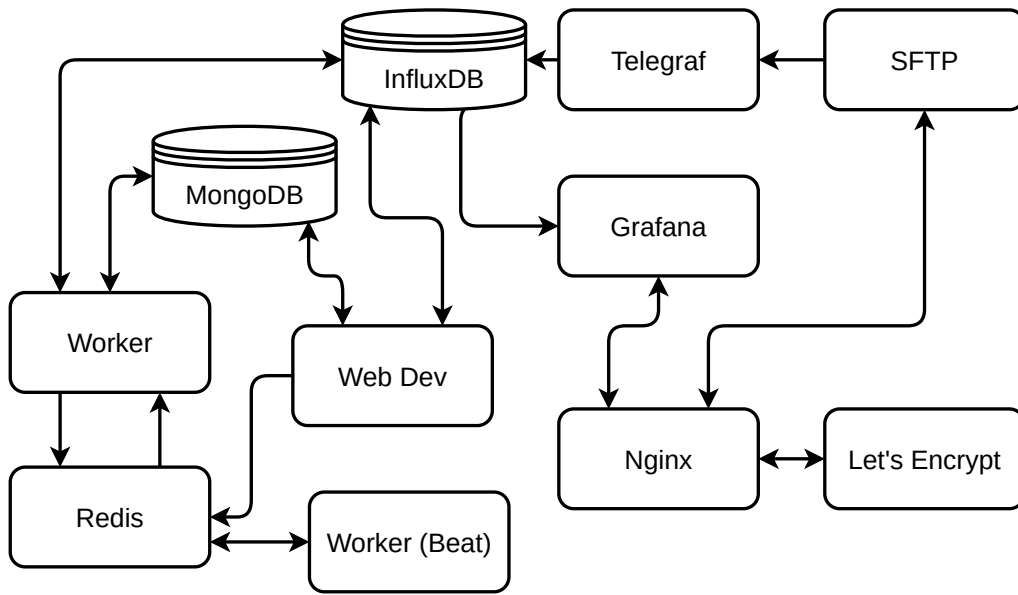

Figure 3.4: old-arch-connections caption under figure

In Figure 3.4 we can see the relations between these containers. 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 graphs 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 Issues
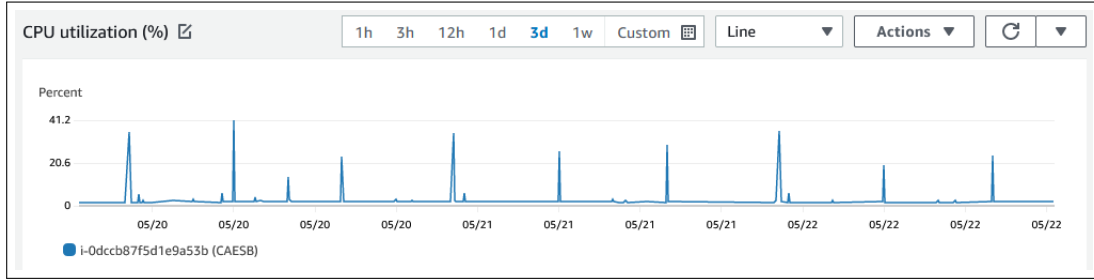
3.2.1 Resource Sizing

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

The contrast between the different services' computational and storage requirements is one of the major issues with 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.

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 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 when the workers are dormant, the development team came to the conclusion 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 solutions 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.

3.2.2 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.

3.2.3 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, 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, we can find build scripts for custom docker images for InfluxDB, Nginx and Telegraf. 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, we can find 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.

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.

Astute readers will notice that, as shown both above and on Figure 3.4, that 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 the service querying it was not the same that 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 needed)

3.2.4 Deployment

When deploying new functionality or code fixes to Client's servers that use the old architecture, this process 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 from 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.

3.3 Requirements

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

3.4 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.

3.4.1 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

3.5 Implementing the Architecture

3.5.1 Backend API

3.5.2 Serverless

Chapter 4

Results and Discussion

Chapter 5

Conclusion

References

Bass, Len, Ingo Weber, and Liming Zhu (2015). *DevOps.* Addison-Wesley. (Cit. on p. 5).

Distributed task queue (2022).

URL: https://docs.celeryq.dev/en/stable/ (cit. on p. 10).

Flask (May 2022).

URL: https://palletsprojects.com/p/flask/ (cit. on p. 10).

Introduction to Redis (Mar. 2022).

URL: https://redis.io/docs/about/ (cit. on p. 10).

Newman, Sam (2019). *Monolith to microservices: evolutionary patterns to transform your monolith.* O'Reilly Media. (Cit. on p. 2).

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. 5).

Spinellis, Diomidis (2012). "Git." In: *IEEE Software* 29.3, pp. 100–101. DOI: 10.1109/MS.2012.61. (Cit. on p. 13).

The One DevOps platform (May 2022).

URL: https://about.gitlab.com/ (cit. on p. 13).

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.