FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
AUTOMATION AND SYSTEMS DEPARTMENT
UNDERGRADUATE COURSE IN CONTROL AND AUTOMATION ENGINEERING

David Steiner Sand

A Framework for Scalable Web Data Collection

Florianópolis
2022

David Steiner Sand

A Framework for Scalable Web Data Collection

Final report of the subject DAS5511 (Course Final Project) as a Concluding Dissertation of the Undergraduate Course in Control and Automation Engineering of the Federal University of Santa Catarina.
Supervisor: Prof. Carlos Montez, Dr.
Co-supervisor: Rafael Jung, Eng.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Sand, David Steiner
A Framework for scalable web data collection / David Steiner Sand ; orientador, Carlos Montez, coorientador, Rafael Jung, 2022.
70 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2022.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Scalability. 3. Big data. 4. Software Architecture. I. Montez, Carlos. II. Jung, Rafael. III. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. IV. Título.

David Steiner Sand

A Framework for Scalable Web Data Collection

This dissertation was evaluated in the context of the subject DAS5511 (Course Final Project) and approved in its final form by the Undergraduate Course in Control and Automation Engineering

Florianópolis, December 1, 2022.

Prof. Hector Silveira, Dr.
Course Coordinator

Examining Board:

Prof. Carlos Montez, Dr. Advisor
UFSC/CTC/DAS

Rafael Jung, Eng.
Supervisor
Company Jungsoft GmbH

Prof. Jomi Fred Hubner, Dr.
Evaluator
UFSC/CTC/DAS

Prof. Eduardo Camponogara, Dr.
Board President
UFSC/CTC/DAS

To my family, for dedicating their time, love and knowledge to my personal growth throughout my entire life.

ACKNOWLEDGEMENTS

To the entire Jungsoft and especially Rafael Jung, Gabriel Prá and Eduardo Schmidt for this opportunity, cooperativeness and advisory.

ABSTRACT

In order to stand out in today's highly competitive European market, companies tend to rely more and more on information regarding the market itself and competitors. Jungsoft GmbH is a long-term focused technology development company for highly demanding projects, which works with the author of this document and with a European client focused on turning data into a competitive advantage by combining high-quality big data, modern analytics and deep automotive expertise. The client sells many products to solve its objective, all of which rely on having abundant, high-quality data. The project hereby presented aims to provide the basis to these products, by allowing the fast collection of data on the web. To do so, a distributed, scalable, easy-to-use and with well-defined interfaces tool was created from scratch. As with every new creation, many obstacles arose in the way, such as: how to design, implement and deploy a scalable software architecture; how fast could new websites be included in the data collection pipeline; and how could inexperienced developers contribute to the project's growth. The project was successfully implemented and currently runs in production, being used by many developers and collecting millions of data points per day.

Keywords: Scalability. Big data. Software Architecture.

RESUMO

A fim de se destacar no mercado europeu altamente competitivo de hoje, empresas tendem a depender cada vez mais em informações relativas ao próprio mercado e aos seus concorrentes. A Jungsoft GmbH é uma empresa de desenvolvimento de software de longo prazo para projetos altamente exigentes, que trabalha com o autor deste documento e com um cliente europeu focado em transformar dados em vantagem competitiva, combinando *big data* de alta qualidade, análises modernas e profunda experiência automotiva. O cliente vende muitos produtos para resolver seu objetivo, todos os quais dependem de ter dados abundantes e de alta qualidade. O projeto aqui apresentado visa fornecer a base para estes produtos, permitindo a rápida coleta de dados na web. Para isso, foi criada uma ferramenta distribuída, escalável, fácil de usar e com interfaces bem definidas. Como em toda nova criação, muitos obstáculos surgiram no caminho, tais como: como projetar uma arquitetura de software escalável; quão rápido poderiam ser incluídos novos websites na coleta de dados; e como desenvolvedores inexperientes poderiam contribuir para o crescimento do projeto. O projeto foi implementado com sucesso e atualmente está em produção, sendo utilizado por muitos desenvolvedores e coletando milhões de pontos de dados por dia.

Palavras-chave: Escalabilidade. *Big Data*. Arquitetura de Software.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

This document describes the development of a scalable distributed web data collection tool as part of a project in partnership with Jungsoft GmbH (JUNGSOFT. . . , n.d.) for a German client.

The client aims to give its automotive industry customers a competitive advantage through data collection and analysis. They have been working in this market for years, but the technical part of the company (responsible for collecting, storing, handling, and presenting data), has developed in a disorganized way.

Jungsoft is a technology development company focused on long-term highly demanding projects. It is based in Berlin, has a multi-national team, and currently works with many clients in a business-to-business model. Jungsoft defines its objective on its website (JUNGSOFT. . . , n.d.) with the following words: "We help both startups with non-technical co-founders and established companies by assuming complete technology ownership and responsibility. We help with Vision and Roadmapping, Architecture Design, Innovation Products, Hiring, Processes, Infrastructure, Optimization and all aspects of innovative digital transformation. Our end goal is to help you validate your product and then hire our replacement team when the time comes."

In order to accomplish its objective in this project, Jungsoft started a long-term business contract with the client related to this work to rebuild its technical sector, which can be unfolded into 3 main areas: frontend, backend and data. This work impacts both the backend and the data areas, focusing on populating the backend database with web data. Jungsoft hired the author of this report to help accomplish this objective.

1.1 OBJECTIVES

This work aims to rebuild the client's data collection area. The objective can be broken down into the following goals:

- Be able to collect data from hundreds of websites periodically;
- Enable rapid development of bots by programmers with little experience;
- Make the data available in a standardized way to the database;
- Allow easy observability of the collection status, in order to quickly visualize performance and identify problems.

1.2 STRUCTURE OF THE DOCUMENT

This report is divided into 6 chapters, where the current chapter focuses on giving an introduction to the work done. A short summary of the objective of the next chapters follows:

- **Chapter 2** - Web Data Collection Fundamentals: Presents fundamental concepts of web data collection.
- **Chapter 3** - Problem description: Identifies problems in the previous solution used by the client.
- **Chapter 4** - Planning: Lists requirements for the new solution, presents the methodology used and managerial process, as well as exploring out-of-the-box solutions.
- **Chapter 5** - Solution: Presents a detailed explanation of the solution implemented.
- **Chapter 6** - Conclusion: Concludes the work done, recapping what was done, how it improved the previous solution and how the objectives were accomplished.

2 WEB DATA COLLECTION FUNDAMENTALS

This chapter focuses on clarifying important concepts for understanding the problem faced by the client and the solution for this project.

2.1 WHAT IS A WEB BOT

A web bot is a program that interacts with the World Wide Web. Web bots are usually built to automate repetitive tasks, doing them in a much more efficient and cheaper way than humans would.

Both web scrapers and crawlers are web bots that visit web pages in order to collect information. Crawlers focus on finding pages that could be visited (by looking for links). This process is called "URL discovery". Scrapers focus on collecting data from already known links and sending this information to whoever wants it.

Googlebot (GOOGLEBOT. . . , n.d.) is a good example of a web crawler. It is the mechanism used by Google Search to crawl the internet looking for new websites or revisiting known ones, in order to keep its content up to date.

Usually, just web scraping/crawling is not enough to create a final product. All the collected information must be smartly stored and presented in order to create value from it. Using Google Search's example, 2 more processes are applied to the data after it is crawled by Googlebot: indexing and serving results. Indexing focuses on scraping the page, looking for valuable information in it, and storing it in a concise and smart way in Google's website index storage. Serving results aims to match the find in the index storage as the best match for a given search phrase, this way, Google can provide structured and objective data about your search.

2.2 WEB COMMUNICATION

A user of the internet might download data in many ways. A common way to do it, especially when talking about simple websites, is using the Transmission Control Protocol (TCP) (TCP. . . , n.d.). TCP is a communication protocol that allows programs to communicate with each other through a network. It is widely used, especially because of it's end-to-end data delivery assurance. Other protocols, like User Datagram Protocol (UDP) (UDP. . . , n.d.), do not guarantee delivery, but speed it up.

Relying on the TCP standard, HTTP (HTTP. . . , n.d.) is another protocol which focuses on specifying instructions on how data should be read and processes. In the HTTP protocol, clients can make requests to servers asking for information. These requests usually contain an URL, which indicates where the data is located, essentially the destination of the requests. They might also contain a body and headers - additional information which might affect what data is returned by the server.

HTTP requests also require a verb - an argument which indicates what action should take place once the request reaches the server. For example, **GET** HTTP requests indicate to the server that the request aims to only collect data from the server and **POST** requests tells the server that the client wants to change the state of the server using the request's body.

Web browsers (WEB-BROWSER. . . , n.d.) are softwares that perform networking requests, like HTTP requests, to present data to users in a user interface (UI). Browsers like Google Chrome and Firefox are popular especially because they facilitate the interaction between humans and the World Wide Web.

The first step to scrape/crawl websites is to communicate with them and download their data. A bot can download the data in many ways, performing HTTP requests directly to the website, creating browsers and using its UI to gather data, or through other protocols not mentioned here, like the File Transfer Protocol (FTP), which allows direct transmission of files between computers.

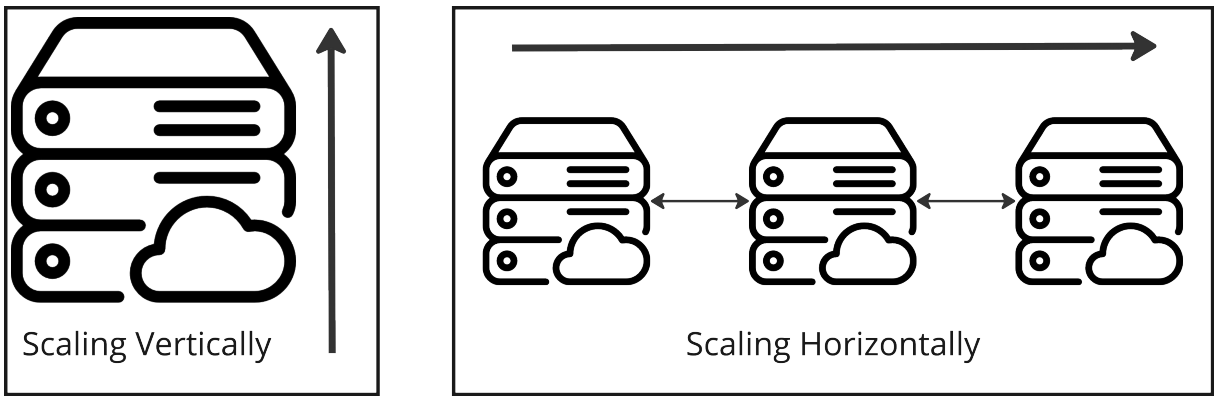
Although this project focuses more in scalability of web applications, rather that on the specificities of network communication, these concepts were essential for building a generic and powerful Framework, that allows bots to communicate to websites using a variety of tools

2.3 DATA COLLECTION IN SCALE

Usually creating a synchronous bot that downloads a link, scrapes/crawls its content and stores it is not such a big challenge. This might get much harder as the number of links and bots increases: if a big website containing hundreds of thousands of links must be daily scraped, the challenge and the solutions become much more complex. Multiple the number of bots by hundreds and add websites with blocking mechanisms in the middle and the difficulty increases even more. To surpass these difficulties, the solution built must be very well planned and implemented.

A central idea of this project is scaling distributedly and horizontally (SCALING. . . , n.d.). A scalable system is a system that can handle the demand, growing and shrinking as the demand varies. This elasticity allows for savings in resources (e.g. CPU and memory), but also for spending resources when the time comes. Scaling vertically means increasing or lowering the power of the machines that already run the system in question, by adding more CPU to it or replacing it for a more suitable machine. In the other hand, scaling horizontally means adding or removing machines from the infrastructure as shown in image 1.

Figure 1 – Scaling



Source: Personal Archive

Scaling vertically might bring complications for big systems, as it is a single point of failure, meaning the system will go down if the server crashes. Also, a machine cannot scale vertically forever, as they will eventually reach resource limits that can only be surpassed by adding more machines. Scaling horizontally is usually more suitable for these situations. When done correctly, it allows for machines to be turned off and still keep the health of the system intact. It also allows scaling to much greater magnitudes, as one more machine can always be added to the system, increasing its power.

The disadvantage of systems that scale horizontally is mostly in complexity. More machines mean more software/hardware maintenance, a communication channel between them and load balancing so that one machine is not overloaded while the others are idle.

The decision of how a system should scale must be done in the early stages of any software project. The mechanism used for horizontal scaling is usually deeply ingrained in the applications developed, so changing from one approach to the other may be too costly to be done once the software is already in production.

The project of this document scales horizontally. A deep explanation of how this happens is given in the next chapters.

2.4 THE VALUE OF WEB BOTS TO THE CLIENT

Web scrapers/crawling bots are a central tool used by the client to achieve its goal of providing a competitive advantage through data analysis and collection. Gathering data from many sources allows the centralization of information and the consequent ease of data handling.

Although on a much smaller scale and specific use case, the final product built for the client can be analogous to Google's search engine (see section 2.1). It will collect

data, index it and store it in a structured way. Finally, the data can be presented to the end user in a concise and organized way.

2.5 FINAL COMMENTS

Now that the key concepts of web bots are clear, the problem faced by the company can be clearly explained in the next chapter.

3 PROBLEM DESCRIPTION

The data collection solution implemented by the client in the past had many issues. The technologies used were outdated/poorly used and the whole data collection pipeline was built with little planning, from the bot tools used to the insertion of data in a database. The specific problems identified are listed below.

1. The bots were built in the programming language PHP. Although used a lot in the past for this purpose, there are more suitable tools nowadays that accomplish the same objective, but have more community support, are simpler to use and have better libraries and maintenance. Examples to replace it are Python, Go and Ruby.

2. The building of bots did not follow any framework. Frameworks for web scrapers/crawlers enforce the developer to build bots following given interfaces and provide many built-in functionalities which speed up development, avoid mistakes and improve quality overall.

3. The system scaled vertically. As mentioned in the last chapter, this can become a big problem once more complex demands reach the system. In this case, some bots developed in the old system were not able to run as fast as required.

4. The bots had too many responsibilities.

    - Each bot implemented its own download logic. No abstraction was created with built-in retry logic, proxy usage, rate limiting, and cache.
    - Each bot parsed the data in its own way, there was no abstraction that was used by all bots to standardize the data output.
    - Each bot inserted data directly into the database. This is highly problematic for many reasons:
        - Each bot needed to know the database schema and follow it strictly. Any database schema changes would require all bots to be updated.
        - The number of interactions with the database increased as more data was collected from the web. This means that the fastest data is collected, the more connections are opened with the database, which will overload it at some point.
        - The bots were responsible for handling database errors. This is one more responsibility that each bot will need to have and will need to know how to handle. As will be seen further in this document, ideally each bot has the minimum responsibility possible, so that new ones can be built fast and reliably.

5. There was no safety in case of failure. if any problem occurred, the bot would fail and its data collection progress would be lost.

6. The database was poorly designed. For relational databases that contain billions of records, like in this case, good planning is necessary. Usually optimizations like Elasticsearch (ELASTIC. . . , n.d.) and Timescale (TIMESCALE. . . , n.d.) are necessary in order to drastically reduce query durations. No optimization tool was used, instead the database was broken down into many table to speed up queries. This decision increased the complexity of the database a lot, making the process of inserting and extracting data from it very complicated.

7. It was not built for scalability.

    - As mentioned, database interactions were a problem as the system grew.
    - The development of new bots was unreliable and relatively slow since each bot did not follow a well-defined interface and had too many responsibilities. Also, there were no auxiliary tools to help developers debug bots or visualize the collected data in a quick way.
    - bots would always run synchronously. This means that every bot would run its commands step-by-step, instead of doing more than one thing at a time when possible. This considerably slowed down the performance of the system.
    - The deployment of bots was mostly manual and not inside Docker containers (see section 5.3.1 for Docker information).

8. There was no easy way to automatically start the bots periodically. Instead, there were employees focused on accessing many virtual machines and starting bots manually every day. In case a bot failed, the employee would again access the virtual machine and restart it.

9. No reliable proxy provider was used, where reliable proxies from a big pool of proxies are used. Instead, the company had a fixed set of proxies and always picked a random one from the set when downloading. This is prone to problems since proxies might go offline or just be blocked on certain websites.

Due to the high number of problems ingrained into the whole technical structure of the company, the decision to rebuild everything from scratch was made. This was surely the most time-consuming and challenging part of the project.

The managerial part was also restructured, and the main problems identified are listed below.

- There was no standard process to request the creation of a bot;

  - As for every team project, a tool to centralize all tasks, delegate them, and set deadlines is essential. Although managing tools were used, no clear process was defined, formalizing how a task should be executed. Instead, superficially defined tasks were created, saying for example "collect data from google.com". This made it really hard for the developer to know what had to be done, and a lot of communication was required between the developer and the business-level employees. This brought a lot of inefficiency to the process, slowing down development.

- Onboarding new developers was really hard;

  - Besides the lack of a framework to help new developers start programming quicker and also a standard process to manage developers, each developer needed to have a lot of knowledge about the data to be collected. The information to be scraped from the page was only formalized as relational database columns with no further description, so the developers needed to have access to the database to know the names of columns, which were not descriptive enough by themselves. Also, usually the data collected from a website needs to be transformed before being inserted into a database, but there was no manual on how to do it.

3.1 FINAL COMMENTS

In the next chapter, requirements for the new solution will be created from the problems identified in this chapter. Furthermore, the planning of the project will be presented.

4 PLANNING

This chapter lists the requirements for the project, then presents a timeline and the methodology used and ends by presenting rejected out-of-the-box solutions for the problem.

4.1 REQUIREMENTS

Based on the problems identified, the requirements for the project were defined, which are depicted in the table 1.

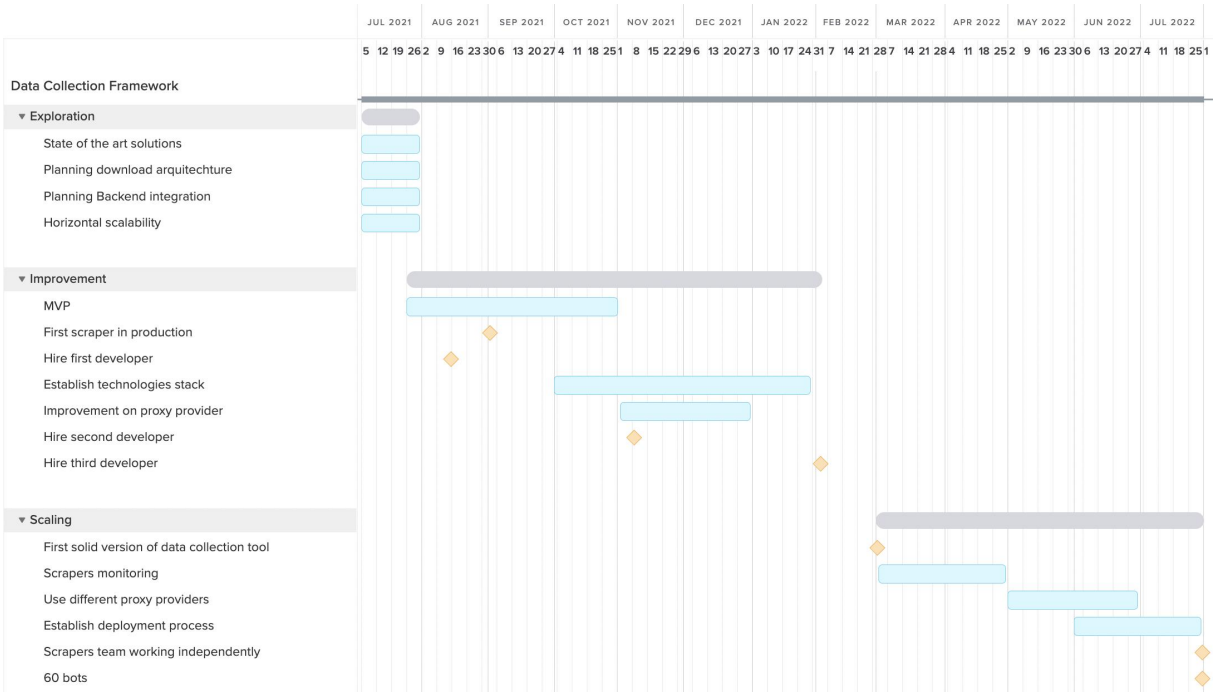
1	Create a system capable of robustly collecting web data on a large scale			
Description	Once the requirements are met, any website of interest which allows data collection should have its data in the company's database.			
Code	Name	Restriction	Obligatory	Permanent
1.1	Download	The framework must be able to download data from any website that allows it, providing an easy and robust downloading method, that handles proxies, cache, and retries.	x	x
1.2	Standardization	The framework must allow easy standardization of downloaded data.	x	x
1.3	Scalability	The framework must scale horizontally and in a distributed way, supporting hundreds of bots and millions of daily requests.	x	x
1.4	Save the progress	The framework must be able to save all bots' progress, so that an error does not force the bot to restart from the beginning.	x	x
1.5	Data persistence	The framework must save scraped data in persistent storage. The storage must not be overloaded at any point.	x	x
1.6	Rate limit	There must be a way to limit the download rate of any given domain in order to avoid overflooding a platform with requests.	x	x
1.7	Cache	The raw downloaded data must be stored somewhere so that in the future it can easily be reused/reparsed.	x	x
1.8	Cron job	The bots should be automatically started periodically by a cron job.	x	x
1.9	Monitoring	There must be an easy way to monitor web bots and the data collected by them.	x	x

Table 1 – Technical Requirements

4.2 TIMELINE

A timeline was planned for the project and it is presented in image 2.

Figure 2 – Planned time line



Source: Personal Archive

As can be seen in the image, the first month of the project would focus on exploration and planning. During this period, the main technologies that would fulfill the requirements would be defined. As it is hard to specifically define all technologies in advance for a project of this magnitude, the decisions taken during this period would only define generic concepts that could be implemented in many different ways. Following the requirements, the plan was to determine important architectural decisions and explore possible implementations of them, mainly in the 3 big areas of data collection: downloading, scalability and data persistence using the backend.

The improvement phase would provide an MVP of the final solution, using what was discovered in the exploration phase. It also forecasted hiring the first developers to start building bots using the tool and establishing all technologies and tools that would be used in the final solution.

The scaling phase aimed to have a solid version of the tool running in production by improving it and including more capabilities, like monitoring and other proxy providers. Also, by the end of this phase, it was expected that a team focused on collecting data would be already established, as well as having 60 bots running.

4.3 MANAGEMENT

This section aims to describe how overall management takes place, commenting about: how people would communicate with each other; how tasks would be created and documented; how developers would receive tasks; how the work of developers would be tracked; how the code base would be managed and stored; what was the work methodology for creating value efficiently.

4.3.1 Communication

Slack is a widely used communication tool that allows private messaging, group messaging, message storage, calls and integrations with other apps. It is mostly used for communication with the bots team, the backend team, and the company board of directors. It is also integrated with other tools for allowing bots to notify the team, like n8n (N8N. . . , n.d.), which will be explained in the future.

4.3.2 Tasks and People Management

Clickup (CLICKUP. . . , n.d.) is a management tool used to organize projects. It is a cloud-based solution that allows storing documents, collaborative work, creating tasks, and assigning them to people. Clickup is used by Jungsoft in many projects, so it was natural to also use it for this one.

Clickup is used in many ways in the data collection project. For instance, it allows structured communication between the business part of the company and the technical part. Weekly meetings with the client's business sector would take place, where the objective is to identify what was done, what is in progress, and what needs to be done in the future by the team. Every decision would then be converted into Clickup tasks, with a deadline and an assignee. Later on, the technical team would discuss how the task at hand could be translated into technical terms to be later implemented.

There are mainly 2 boards relevant to this project: the framework board and the bots board. The framework board holds tasks relevant to building the framework, while the bots board contains requests for building bots for specific websites. The author of this document was responsible for maintaining and implementing the framework board. The bots board was also the responsibility of the author in the first months of the project but was later delegated to somebody else as the number of tasks grew too quickly. Note that the requirements listed in the table 1 are all related to the framework board, not to specific bots. This report will not describe any bots in detail, but only framework-related tasks.

Beyond organizing tasks, Clickup is also used as a knowledge database, especially for the bot-building team. Some examples of information stored there are: techniques for downloading data from complicated websites; a catalog of attributes to

be collected from websites for a given market (e.g. prices of products, tax rates, brands, etc); explanations of how attributes should be parsed and transformed.

Clickup solved the communication problem previously observed in the company. Also, having a manager who focused on listening for busines-level tasks requests and formalizing them into Clickup tasks separated business employees and developers almost completely, increasing productivity.

4.3.3 Code Management

Every big coding project requires a good version control tool to allow organization of work coordination between many developers. The version control tool used is Git (GIT. . . , n.d.).

4.3.3.1 Git

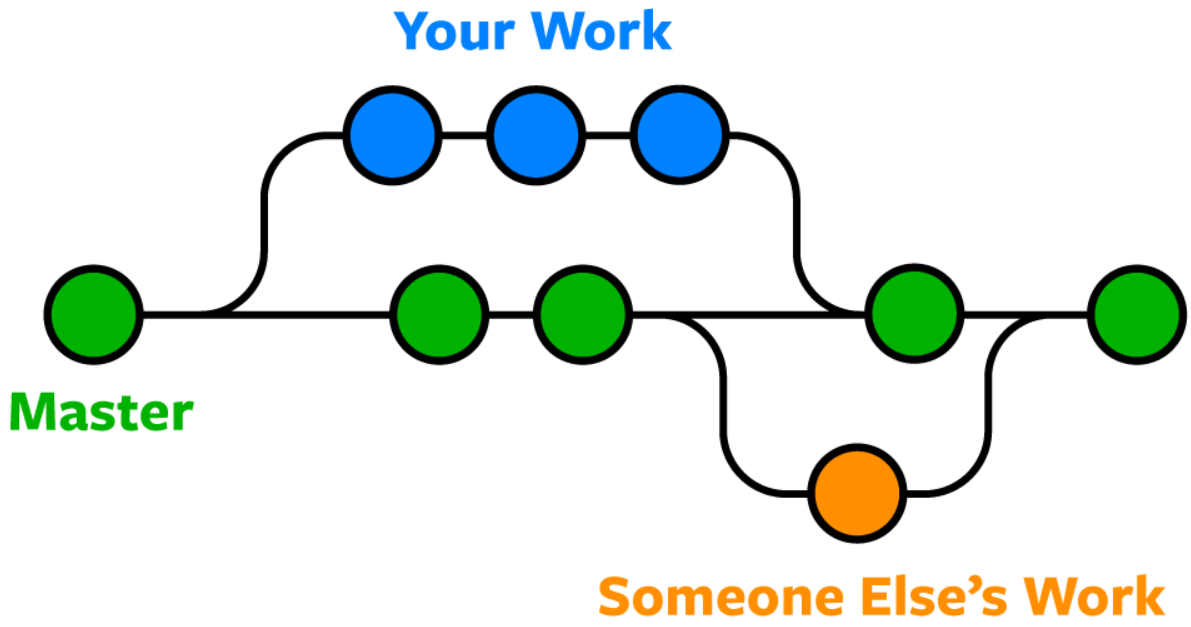
Git is a free, open-source, and distributed version control tool. It allows any written project to be versioned, be it code or just raw text. Projects that use Git are viewed by it as repositories, that are capable of tracking the project's changes (versioning) and easily being uploaded to cloud-based Git hosts. Github (GITHUB. . . , n.d.) and Gitlab (GITLAB. . . , n.d.) are 2 examples of cloud-based hosts for Git repositories, which allow the management of repositories remotely. This project was built using Gitlab, since Jungsoft uses it for every project and pays for an enhanced version of it.

New versions of a given repository can be registered by **commiting**, which is done by running the **git commit** command. This adds a **commit** to the repository's timeline, an atomic alteration of the software that organizes it in an understandable and manageable way.

Many parallel versions of the same software can also be created at the same time, though the concept of **branches**. Branches usually are created to contain new features of the software or to resolve a specific problem and are usually implemented by a single developer. They are an important part of the methodology presented in subsection 4.3.4.

Image 3 illustrates how many branches might work together in the same repository. Considering time moves from left to right, the software starts with an initial commit (the round symbols in the image) in the **master** branch, the branch which contains validated code. As time passes, more branches are created, which contain their own commits.

Figure 3 – Git branches



Source: (GIT-BRANCHES:. . . , n.d.)

Once the work done in a branch is done, someone can request to merge it with the work already available in the **master** branch, a process also described in subsection 4.3.4.

4.3.4 Work Methodology

The methodology used for developing the framework followed a similar idea as the Agile Methodology, see image 4.

Figure 4 – Agile Methodology

Source: (AGILE. . . , n.d.)

Weekly meetings with the client's board of directors and Jungsoft advisors would take place, where the first and second steps of the Agile Methodology would be completed: new requirements would be presented based on the performance of the previous week. The requirements would be transformed into Clickup tasks, as shown in the workflow at 4.3.4.1.

4.3.4.1 Workflow

The flow of work followed the following pipeline:

1. Create a Clickup task;
2. Rank task by importance level and set a deadline;
3. Augment task with technical descriptions, indicating how it should be implemented;
4. Assign a responsible for the task;
5. Once the task is ready to be implemented, the assignee should create a branch in the project's repository, upload it to Gitlab and link it to the respective Clickup task;
6. Once the developer thinks the implementation is good enough, a code reviewer should be assigned to the task. A code reviewer should be a developer different

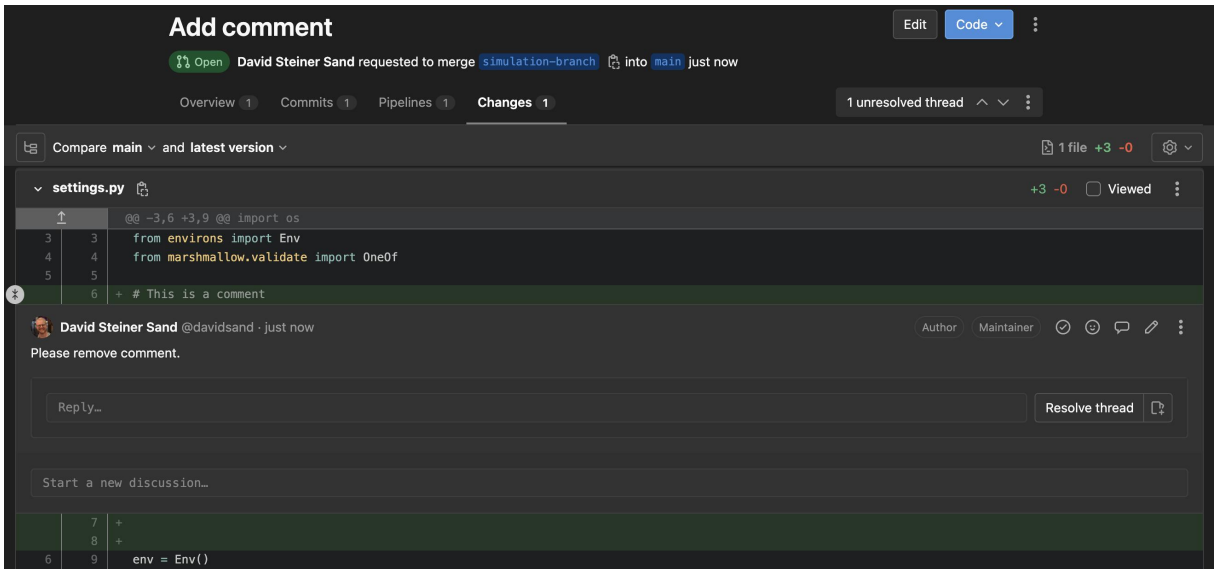
from the one that implemented the task, preferably with more experience. Gitlab has a good interface for code reviews, as shown in image 5. It allows users to create discussions, which need to be resolved before the branch in question is merged into the main branch. The interface also allows anyone with access to the Gitlab project to monitor what is being done in each branch.

7. The developer and reviewer iteratively discuss and implement improvements for the branch's work, until an acceptable implementation is reached (refers to the fourth step of image 4). Once this happens, the reviewer can merge the branch, adding its code to the main code base.

8. Since the master branch is now updated, it can be deployed. For deployment information, see section 5.3.

9. The deployment would be monitored frequently, where bugs and improvements could take place for the next sprint.

Figure 5 – Code review



Source: Personal Archive

The workflow for building bots followed the same idea as the framework one, presented at 4.3.4.1. It only contained one additional step, after the step 6: data review. The environment Homologation (explained in subsection 5.3.2.3) was used for this step; it allowed developers to easily access the Data Validation Report, shown in section 5.2.2 and create a diagnosis of the data collected by the bot. Just like in code reviews, a developer was assigned to the data review. In case the data was not good enough, the

developer implementing the task would then fix the bot and re-submit it to data review, in an iterative process until its data was acceptable.

4.4 OUT-OF-THE-BOX SOLUTIONS

At first, bot services that complete the whole data collection pipeline were explored (out-of-the-box solutions). Services like Zyte (ZYTE. . . , n.d.) would download the data and make it available in a standard way in a database. But since the project in question is quite big, a solution like this would be really expensive. Another possibility would be to only use the infrastructure of these services, where they accept bots written in a certain framework as input and run them in scale in their infrastructure. As this creates too much dependency between the company and the bot service, this approach was also not taken. So a completely ready-to-use solution was rejected.

Since the bots would have to be built, a programming language had to be chosen. Python was the winner here for 2 main reasons: it is widely used for web bots, so there are many tools available to support a project like this one, as well as a big community; developers would need to be hired in the future to build the bots, and Python developers are the easiest ones to find, especially for not so complex tasks.

The following article gave the team a good starting point for a solution: (MEDIUM. . . , 2020). In the article, an idea for a scraping framework is superficially explained, commenting on how it could scale and keep the state of bots even in case of failure. The article also mentions Scrapy (SCRAPY. . . , n.d.), a popular Python scraping framework, explaining how it can become hard to implement more complex logic in it, which was an obstacle also observed by the author of this document. Scrapy was rejected and it was decided that an in-house (i.e. by the author of this document and Jungsoft) Python-based solution would be developed.

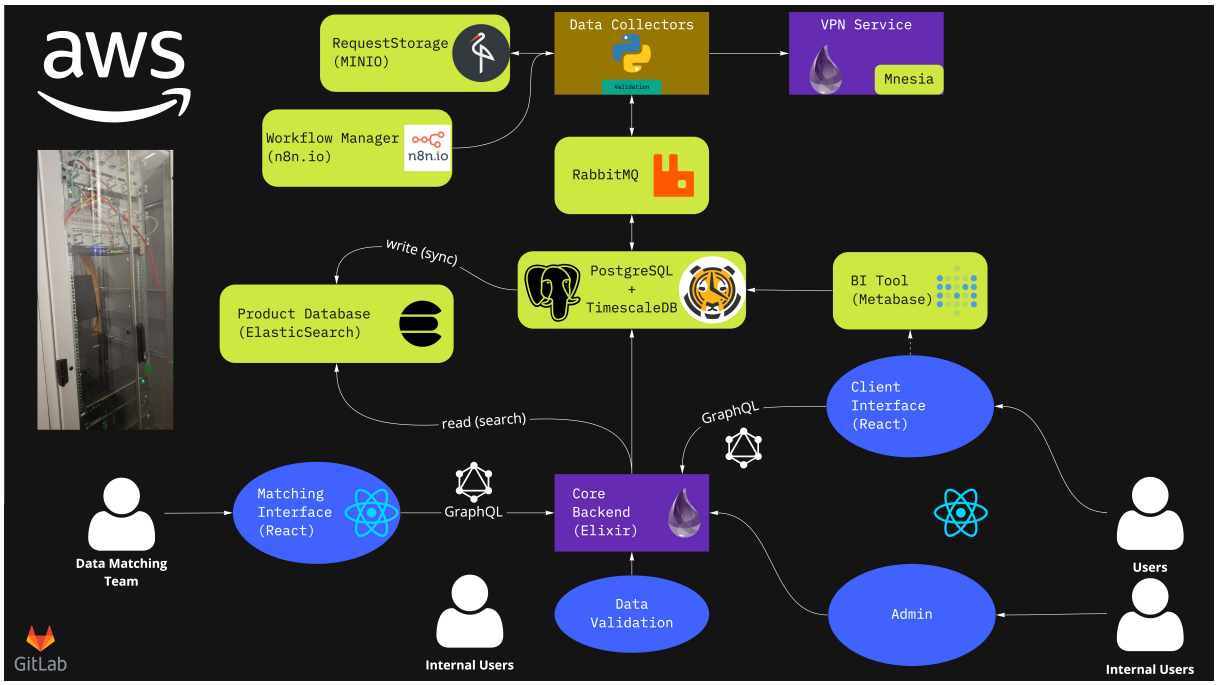
4.5 FINAL COMMENTS

In this chapter, the planning of the project was presented, as well as reasons for the necessity of building the solution to be presented in the next chapter.

5 SOLUTION

This chapter focuses on presenting the solution implemented. First, an overview provided by Jungsoft representing almost the entire new technical structure of the company is presented in image 6. This is important to explain that this project is a part of Jungsoft's solution, focusing on the data collection, which will be presented right after Jungsoft's overview.

Figure 6 – Infrastructure Overview



Source: Jungsoft Archive

The next paragraphs will use bold text to reference elements in image 6.

Starting in the right corner of the image 6, **Users** represent clients of the client of this project, i.e. buyers of the products of the client. **Users** are usually companies that sell products and are interested in comparing their prices with other companies. To provide a good user interface (UI) and user experience (UX) to **Users**, Jungsoft built a Frontend (**Client Interface (React)**) using React (REACT. . . , n.d.), a popular Javascript library for building UIs.

The Frontend communicates with the Backend (**Core Backend (Elixir)**), written in the language elixir (ELIXIR. . . , n.d.), through **Graphql** (GRAFANA. . . , n.d.) queries to access the data in the central database (**PostgreSQL + TimescaleDB**). The central Postgres (POSTGRES. . . , n.d.) relational database uses TimescaleDB (TIMESCALE. . . , n.d.) and ElasticSearch (ELASTIC. . . , n.d.) (**Product Database (ElasticSearch)**) to provide a quick way to interact with the stored data. By using these

technologies, Jungsoft noticed that some queries that in the old client's solution took minutes to complete now take less than a second.

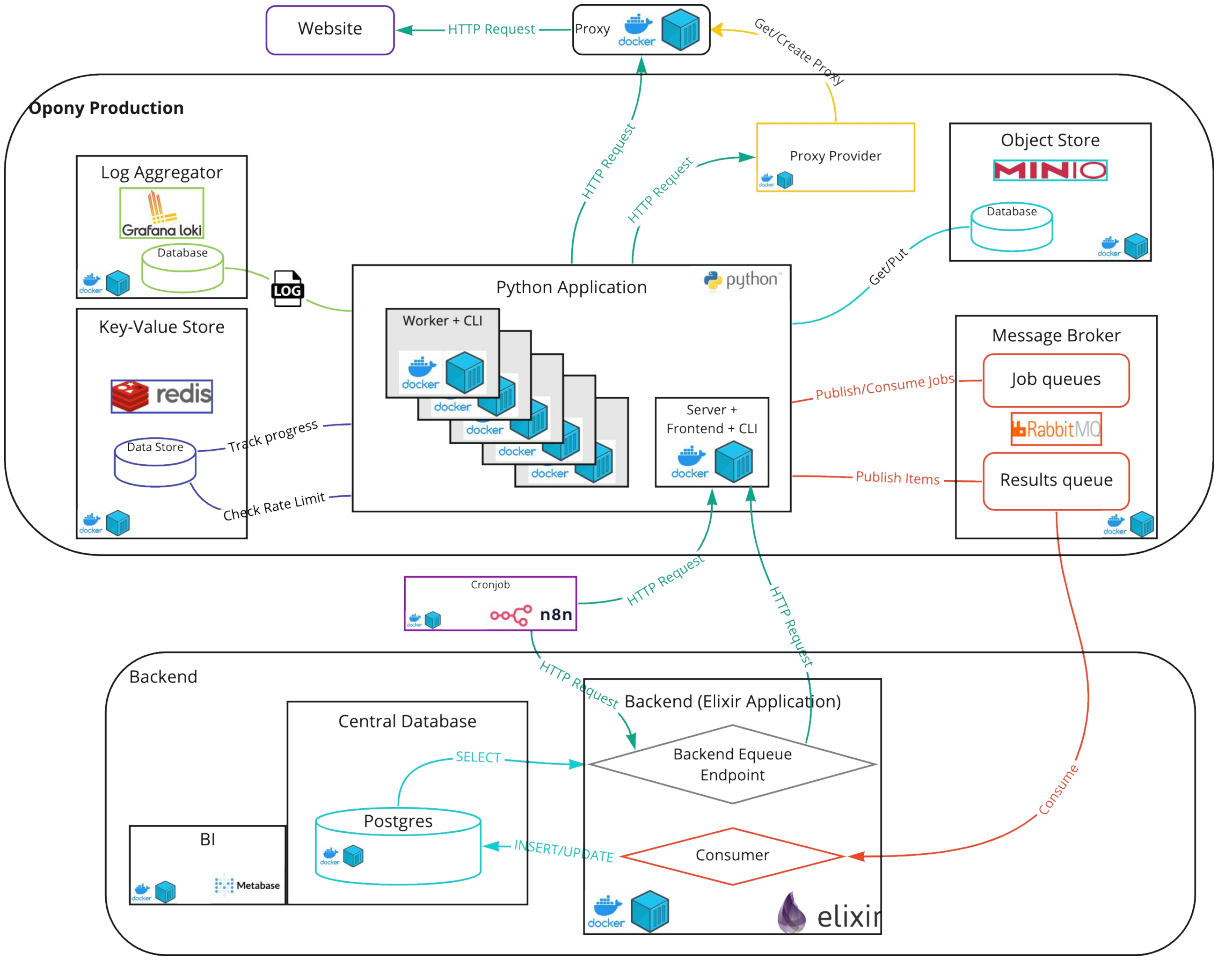
Moreover, the system is accessed by other types of users. **Internal Users** have an **Admin** access to the backend and database, allowing them to perform operations that **Users** are not allowed to. Because of this, **Internal Users** could easily perform **Data Validation**, making sure the data collected was of good quality.

The web data collection mechanism, the focus of this work, collects data from many different sources. The data ends up being inserted into the central database. However, a lot of the data collected from different sources reference the same information, e.g. website A may sell the exact same product as website B. The problem with this is that it may be very hard to identify that a product from website A is the same as the one from website B. I.e. it is very hard to match products between websites. Since this is very important to provide price comparisons for a given product, a reliable mechanism is essential here. No automated solution exists at this point in time, so the client has a **Data Matching Team**, which matches millions of products from hundreds of websites using a **Matching Interface (React)** built by Jungsoft.

The data collection solution consists of the top part of image 6. The author of this work, although having participated also in backend features and data validation, focused a majority of his time on this part. The implementation of the data collection system was done by the author with the advisory of Jungsoft.

Image 7 shows an overview of the data collection solution. Note that image 7 and image 6 intercept each other, as elements like **RabbitMQ**, **RequestStorage(MINIO)**, **Workflow Manager (n8n.io)** appear in both images. Every element in image 7 will be deeply explained in this chapter, since they are part of the data collection solution, differently from the other elements shown in image 6.

Figure 7 – Micro services overview in Production



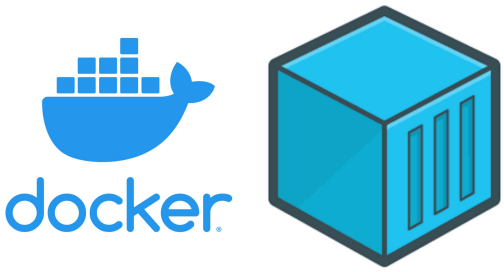
Source: Personal Archive

In the next sections, elements present in image 7 will be written using bold text.

The main part of the data collection solution is the web data collection framework, called **Opony**, which consists of a mix of microservices (MICROSERVICES. . . , n.d.). At the center of it, the **Python Application** microservice works as the brain of the solution, allowing communication between the microservices. It is also shown in image 6 as Data Collectors.

Each rectangular frame in the image 7 represents a microservice. Note that all microservice contain the symbol shown in image 8, which indicates that the microservice is run inside a Docker Container. Docker containers will be further explained in section 5.3.

Figure 8 – Docker Container Symbol



Source: Personal Archive

Each microservice is included in Opony to accomplish a specific objective, which helps to fulfill one or more requirements.

Each requirement described in table 1 has a different level of complexity and urgency. In the list 5, they are ordered from highest priority to lowest, to know which one should be tackled first.

- Scalability
- Download
- Data persistence
- Save the progress
- Standardization
- Cron job
- Rate limit
- Monitoring
- Cache

Next, a deep explanation of each requirement will be given, followed by how the implemented solution relates to each requirement and how each microservice helps accomplish the final goal.

5.1 FULFILLING THE REQUIREMENTS

Each requirement demanded the planning and implementation of various types of solutions, which are described in this section. Each solution proposed had to be carefully thought out since it may affect other solutions and make a requirement unfeasible to have.

5.1.1 Scalability

Building a scalable system is a delicate subject since the features that make it scalable are usually deeply ingrained in its root mechanism. All other requirements are affected by the decisions taken here, not only performance-wise, but also in the way they will be implemented. In other words, it is essential to get this part right, otherwise one may find itself with a useless tool once the demand for data grows. Therefore, this was the focus from the beginning, where the goal was to build a distributed system that could scale horizontally, i.e. a system that is able to run on many servers simultaneously and also accepts the inclusion of more servers always when needed.

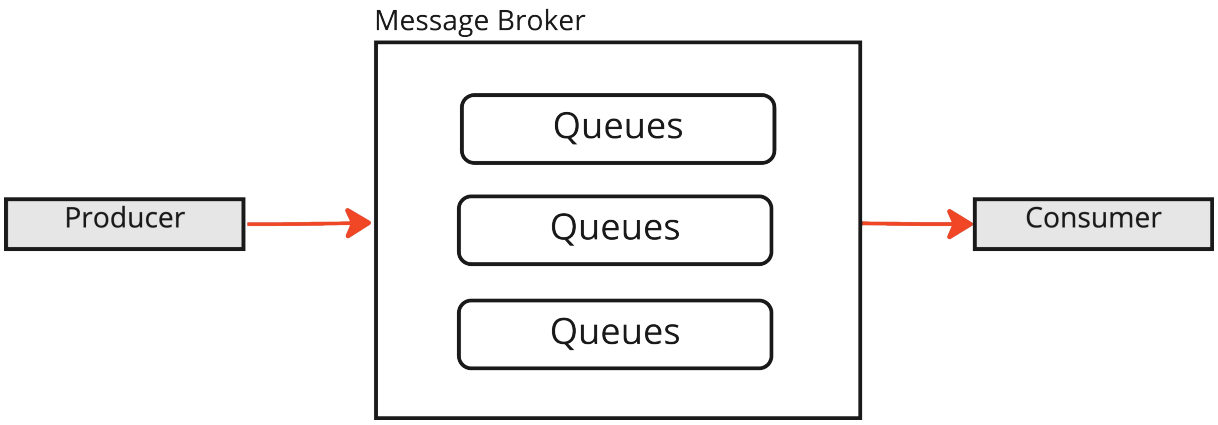
The microservice-oriented approach shown in image 7 allows such scalability since each microservice shown in the image can run on different machines. Beyond the services of the distributed system, a central and also distributed **Python Application** shown in image 7 served as the orchestrator of the services. It communicated with the other services through well-defined communication protocols, e.g. HTTP and AMQP.

The **Python Application**, as mentioned, was also distributed. Always when more power was needed, more Docker containers could be run to horizontally scale the application, allowing faster data collection.

In order to delegate tasks to container of the Python Application and store the data they output, a **Message broker** (MESSAGE-BROKERS. . . , 2022) was used: **RabbitMQ**.

- What is a **Message broker**?
  A **Message broker** is a service that allows communication between services by sending messages of defined formats using a message protocol. This allows communication between machines and applications, even if these are written in different languages. Beyond this, **Message brokers** can validate and also store data. For this project, the storing capability of the broker was widely used, as will be shown in the next sections. A basic diagram representing how a broker works is shown in figure 9.

Figure 9 – **Message Broker**



Source: Personal Archive

The **Message broker** service itself is represented in the middle. It accepts data, holds it, and releases it when required. The broker used, **RabbitMQ**, has 2 main building blocks:

- Queue - A FIFO queue that holds information. An instance of the broker might have many queues.
- Exchange - An abstraction that decides how to handle incoming information. When a message is published to the broker, it never goes directly to a queue - the exchange always decides before how and to which queues the message will go. The message may be duplicated and go to many queues, it may go to a single queue or even be discarded, depending on how the exchange is configured and how the message look like.

A given application must follow the messaging protocols imposed by the broker to speak to it. In the case of **RabbitMQ**, the AMQP protocol was used, which runs over a TCP connection that is much less prone to errors or data loss, when compared to UDP for example. AMQP also is more complex than other messaging protocols, like MQTT, but offers more features, so it was the picked choice since it was not clear at the beginning of the project how much complexity would be involved.

To interact with the **Message broker**, producers and consumers must be defined in the application. Producers send data to the broker's exchange and consumers get data from the broker's queues. Once a consumer gets a message, it can *acknowledge* it if desired, meaning the message is safe for the consumer and the broker can erase it from its memory without further concerns.

Many client libraries are available for **RabbitMQ** for Python, where the one used was Kombu (KOMBU. . . , 2022), which is used by *Celery* (CELERY. . . , 2022).

- What is *Celery*?
  Celery is a Python library and an abstraction above Kombu, which provides Python utilities that allow easy interaction with brokers. *Celery* defines itself as a distributed task queue software. It allows python programs to easily publish and consumes messages to the broker. By defining tasks, functions that process messages, a given python application can easily become distributed and asynchronous.

5.1.2 Download

Downloading data is prioritized second as it handles all the data input of the system. Once the data is in the system, it can be handled and transformed, but to get to this point, a robust and scalable downloading architecture is required. Here the focus is to define download mechanisms that are fast and reliable, to collect as much data as possible in a smaller interval. The main challenge is to define a proxy architecture that is able to handle the data load.

As mentioned before, the old data collection system used to have all the downloaded data passed through a static pool of proxies. When a download was made, one of the proxies in the pool would be chosen, but problems occurred since proxies might go offline or be blocked.

A solution for this problem is to implement a **Proxy Provider**: a service that provides proxies on demand, that would then be used by the Python Application to make requests to websites, see image 7. Initially, ready-to-used paid services were explored, like Brightdata (BRIGHTDATA. . . , 2022) and Smartproxy (SMARTPROXY. . . , 2022). The conclusion was that it would be too expensive to pay for all the data required by the company. Instead, they would only be used as backup **Proxy Providers** by Opony, i.e. they are only used if the main solution fails. The main solution was built and deployed in-house.

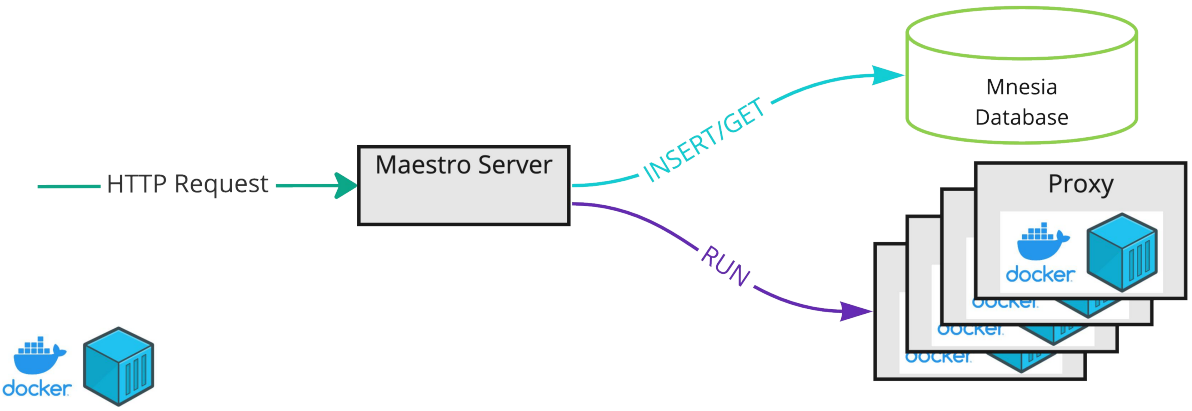
The requirements for the built **Proxy Provider** were:

- To be able to dynamically create and destroy proxies;
- To have a blacklist mechanism so that problematic proxies or proxies blocked in a given platform would not be used sequentially for the same platform.
- To be scalable since potentially millions of requests would be made to the provider and to its proxies daily.
- To be able to accept many different proxy accounts. Each proxy account could have a limit of simultaneous connections.

The solution is called Maestro, and it was built using the programming language Elixir. Elixir was chosen for many reasons: it is a very efficient compiled language, unlike Python; it is built on top of the Erlang virtual machine, excellent for concurrency, scalability and distribution; it is part of Jungsoft's and the author's programming languages stack. The core of Maestro's application was built by the author with the advisory of Jungsoft, and it was later maintained and improved by Jungsoft.

A simplification of how Maestro works is depicted in the image 10.

Figure 10 – Maestro



Source: Personal Archive

Maestro runs in a Docker container (to be explained in section 5.3.1). It consists of a server, which accepts HTTP requests, and runs Docker containers that contain the image of a Proxy. Maestro looks at the content of the request and at the Mnesia Database to decide how many proxy containers should run, and runs them, returning to the request's client the address of the proxy. Once the client has the address of a proxy, a request to a website can be easily made using the given proxy.

Mnesia is a database management system that synchronizes disc and memory to provide fast access to the database, which is extremely necessary since Opony performs almost as many requests to Maestro as to all websites. The database is used to store information about proxies that were not used successfully on a given website. This allows Maestro to know what are the best proxies for any given website already accessed.

5.1.3 Data Persistence

Data persistence aims to save the data once it is downloaded. As mentioned, the old system only used a relational database to persist information, which can be a big problem in big data systems, since relational databases are usually not built for hundreds of concurrent connections. Having hundreds of bots inserting data in

the database concurrently caused peaks of demand in the database, causing serious performance issues. Also, since the bots were responsible for inserting data, they had to also handle insertion errors, increasing their complexity.

Rabbitmq, beyond enabling communication within the distributed system, persists the data before it reaches the main database. Messages published to the broker do not need to follow any schema but are structured as JSONs, differently from relational databases. This intermediate step in the data lifecycle, where it passes through the broker, removes the responsibility of inserting the data in the **Central Database** from the bots and moves it to the **Backend**, as shown in image 7. The **Backend** runs a process that consumes messages in batches from the broker and inserts them into the database. This approach is also more flexible, allowing the **Backend** to parse the data before insertion.

Note that in case of failure, be it in the bots or in the **Backend**, no data in the broker is lost, as messages are only *acknowledged* after success.

5.1.4 Saving the Progress

Saving the progress of bots was a challenge, but not such a critical one as the previous ones. The goal here is to force all bots to constantly persist their progress in the message-broker so that an error could occur at any time and nothing would be lost.

In order to constantly save the progress of any bot, the bots had to be implemented in a certain way. Opony forces bots to implement 3 methods: *download*, *scrape* and *crawl*, all explained in detail at subsection 5.2 and used by the **Opony Pipeline**, which also explained at subsection 5.2. The bots should implement these methods in a minimalistic way so that *download* performs preferably only 1 download, *scrape* scrapes only 1 page and *crawl* crawls only 1 page. This allows the progress to be saved always after the **Opony Pipeline** is run, where new messages are published to the broker and therefore saved.

5.1.5 Standardization

Standardization is mainly defined inside the application, so no architecture was altered by it. To achieve it, the bots are forced to publish standard messages to the **Message Broker**, so that messages can be retrieved from the broker and inserted in the **Central Database**.

To enforce standardization, the Python library Pydantic (PYDANTIC. . . , n.d.) is used. Pydantic basically allows developers to easily validate data with its built-in features. For example, a smartphone could be represented in a Smartphone class, which contains fields like size, battery capacity and price. Then, Pydantic validations could be defined for each field, saying for example, that a price must be an integer, never a string. In case a validation fails, the **Opony Pipeline** (explained at subsection

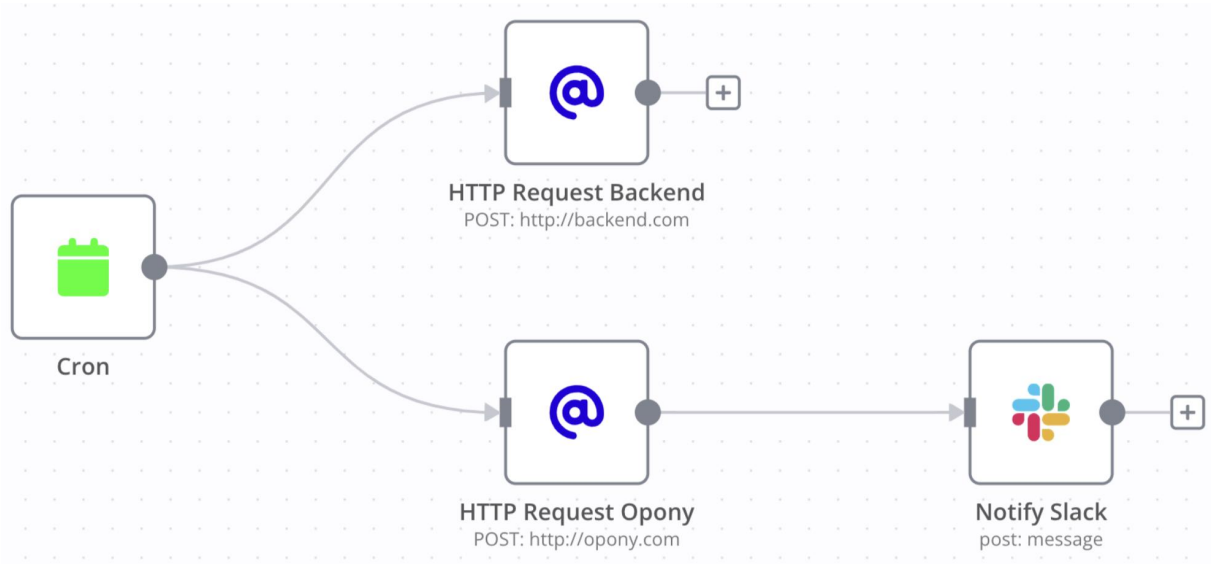
5.2) will alert about the error and move the message to a separate *dead letter queue* in the broker. A *dead letter queue* is a queue in the broker which holds messages that cannot be delivered to their destination queue, so they are a great fit for holding failed messages in this case.

5.1.6 Cron job

Since most of the client's data of interest must be collected every day, an automatic way to start hundreds of bots is necessary. Also, the exact period when each individual bot should be triggered (cron time) should be easily configured.

Therefore, another solution was explored. **N8N** (N8N. . . , n.d.) is a workflow tool that integrates easily with many popular apps, providing an easy to use UI also. Image 11 shows an example of a workflow used by the data collection tool.

Figure 11 – N8N workflow



Source: Personal Archive

The cron block triggers 2 different HTTP requests and sends a message to Slack, notifying the team that the request was made.

The request labeled *HTTP Request Opony* sends an HTTP request to Opony's server, which triggers a new bot to be started.

The request labeled *HTTP Request Backend* sends an HTTP request to the **Backend**. To understand why this request happens, the type of data collected by Opony needs to be shortly explained. Data collected from Opony can be divided into 2 categories: *scans* and *details*. *Scans* is data that needs to be collected periodically, like price data, since the client is interested in tracking this kind of information over time. *Details* on the other hand is information that does not vary over time, such as a certain

product specification, e.g. the size of a specific smartphone or the nominal power of a certain engine. The target websites usually store this information on different pages, e.g. there are pages that show a list of prices and each price contains a link to the details of that specific product. To spare requests, Opony only goes through the prices pages periodically, not the details page. Each details page only needs to be visited once, since its information does not change. To know what are the detail pages of every website and if these pages were visited already or not, the **Central Database** is used. Therefore the request *HTTP Request Backend* is made to the **Backend**, telling it to let Opony know which detail pages still need to be visited. This request is done periodically since new prices might appear on the websites, which contain details not present on the websites before, but the **Backend** only sends non-visited details to Opony.

Both HTTP requests are also depicted in image 7, departing from the N8N microservice.

5.1.7 Rate limit

The idea here is to avoid overflooding a certain website with requests while trying to collect data too quickly from it. There needs to be a mechanism to throttle the download speed in the framework. This is not such an easy task, since the framework works in a distributed way, i.e. many different machines maybe be collecting data at the same time from the same website. In order to solve this, the information of which website is being downloaded at any given point is centralized in a **Key-Value Store** - in this case, **Redis**.

Redis (REDIS. . . , n.d.) is a fast-access data store that is widely used for distributed systems. The Python Application makes a request to Redis (represented by the connection "Check Rate Limit" in image 7) any time a download is about to happen. The domain of the URL to be requested and rate limit information is sent to **Redis**, which checks if there was a recent request to the given domain registered in its memory. The response of **Redis** defines rather Opony continues and makes the request to the URL or if it postpones the request.

5.1.8 Monitoring

Monitoring is a big part of Opony. It is not an essential mechanism for the Framework, but it gets more and more important as the system grows in size. Monitoring can be divided into 3 areas: error monitoring, bots monitoring and data monitoring.

- **Error Monitoring**
  At any given time, a bot might fail for many kinds of reasons: the website from where the bot is collecting data might be offline, the bot might have been implemented wrongly, the website might have changed its structure and the bot built is

not capable of parsing its content anymore, etc. In order to capture errors, Sentry (SENTRY. . . , n.d.) was used, a tool that provides dashboards for real-time monitoring of crashes in applications. With Sentry, any error in the application can be almost instantly detected, located and quickly fixed.

Sentry is a paid service and therefore it applies a limit for the number of errors it accepts in a given period, depending on which plan is chosen. Since Opony does a lot of computation, an error in the application might result in thousands of error reports in Sentry. To avoid flooding Sentry and consequently maxing out its limit, Opony filters its errors before reporting them. E.g. Some errors are just not relevant enough to be sent to Sentry and the ones that are can be throttled so that the same error is not reported many times in a short period of time.

- **Bots Monitoring**
  Bots monitoring aims to understand what is happening in every bot in Opony at any given point in time. To achieve this goal, a log aggregating service, **Loki** (LOKI. . . , n.d.), is used, as is shown in image 7. A log aggregating service gathers the logs of an application in a smart way and provides a language to query data from it, in this case, the language is called LogQL (LOGQL. . . , n.d.). LogQL, like SQL, is a database query language, but while SQL queries data from relational databases, LogQL queries data from an unstructured database. The code shown below is an example of a LogQL query.

```
sum (
  rate(
    {application=~"opony"}
    |~ "Fetching document"
    [1m]
  )
)
```

  The query is explained line by line in the items below.

  - **application="opony"** - Collects log lines that are labeled with the application "opony".
  - |~**"Fetching document"** - filters any lines that match the regex expression **"Fetching document"**. Opony logs "Fetching document" every time a download is happening, so finding these log lines allows **Loki** to know how many downloads happen in a period of time.

- **[1m]** - For each data point found, collect every data point 1 minute in the past that also matches the application and the filter, creating many vectors, one for each data point.
- **rate** - Calculates the number of log lines per second of every vector created, resulting in a rate value for each data point.
- **sum** - Sums all rate values that refer to the same point in time. Opony sends log to **Loki** from many locations, since it is a distributed system. This line tells **Loki** to sum the values from these locations and return a single vector.

Therefore, for this example, **Loki** will provide a time series indicating the rate (per second) of log lines that match the given filter, i.e. the rate of download per second. Many queries like this one were created to help monitor Opony, each one with its own purpose.

Grafana (GRAFANA. . . , n.d.) was used to make these queries to **Loki** and show their results in dashboards. **Grafana** is an interactive platform for data visualization and alerting. By combining **Loki** and **Grafana**, many monitoring dashboards were created, some examples of metrics observed are: download rate per domain; memory usage; amount of data points collected per domain; most common errors per domain; time required to collect all data of a given website; proxy providers performance; interactions with the cache.

An example of a query and its result is shown in image 12.

Figure 12 – **Grafana** Chart

Source: Personal Archive

The image shows **Grafana's** user interface. The chart in the image displays the rate over time of log lines that match the query. **Grafana** also allows the user to define what time window will be used, in the image 12 it is set for "Last 1 hour" in the top right corner. This allows developers to easily visualize past data and locate unexpected behavior.

- **Data Monitoring**

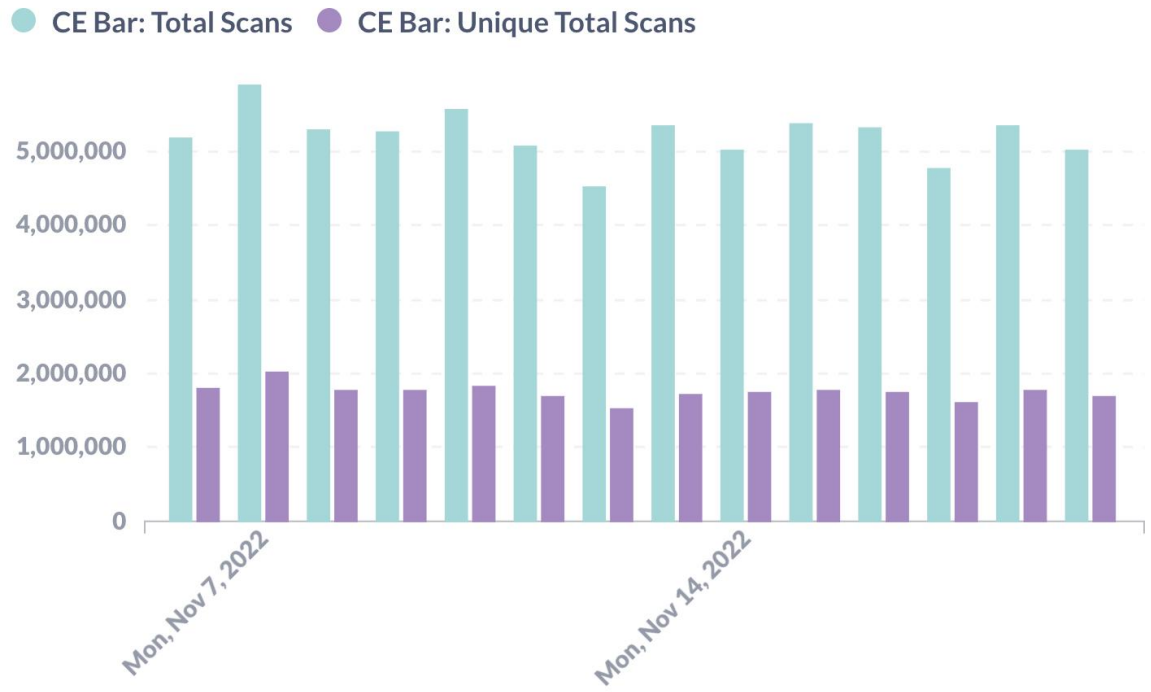
Data is monitored through 2 main tools: Data validation report, shown in section 5.2.2 and **Metabase** (METABASE. . . , n.d.). **Metabase** is a business intelligence tool, which integrates with many databases and provides a user interface that allows easy creation of charts.

While the Data validation report is used mostly for the Development (section 5.3.2.2) and Homologation (section 5.3.2.3) environments, **Metabase** is used mainly for Production (subsection 5.3.2.4) data, since in Production the volume of data is much bigger and more detailed analysis is necessary. **Metabase** connects to the **Backend Central Database**, as shown in images 6 and 7.

Image 13 shows an example of a dashboard created using **Metabase**. For each day, it shows the number of scans collected (tall bars) - which basically represent price points - and the number of unique scans (low bars) - which represents the

number of products reached. The number of scans is greater than the number of unique scans, since there may be many price points for a given product.

Figure 13 – **Metabase** Chart



Source: Personal Archive

Data is the resource extracted by the bots, so the data quality is a very good indicator of the bot's performance. Therefore, by looking at this and other charts every day, the team is capable of having a good understanding of the performance of the bots, knowing which ones are working well and which ones need maintenance.

5.1.9 Cache

After downloading a page of a website, Opony bots extract information that seems relevant (scraping) and discard the rest. The relevant information is published to the **Message Broker** and inserted in the database in the future, as shown in image 7. The problem on relying solely on this this approach is that there might have been some relevant information lost in the discarded data. To avoid this problem, all raw downloaded data is cached in the **Object Store** before scraping. Images 7 and 6 show the **Object Store** service: **Minio**.

By storing raw data, Opony is able to revisit websites as they were in the past, analyze if any changes occurred in them and collect more information, which was discarded in the past.

The textbfObject Store can also be very useful in development, where someone building a bot can fetch website data from it instead of always re-downloading data, making the process faster.

5.1.10 Requirements Conclusion

Solutions for all requirements have now been presented. The next sections aim to explain how Opony can be accessed, deployed and used to quickly build web bots.

5.2 BASIC FUNCTIONALITY

This subsection aims to gather the services presented in the previous section and explain how they could work together in a harmonious way.

As shown in image 7, the **Python Application** is at the center of Opony. It is responsible for deciding how to handle incoming requests and for how to use all available microservices.

The **Python Application** revolved around the concept of the **Opony Pipeline**. The **Opony Pipeline** can be thought of as a Data Pipeline (DATA-PIPELINE. . . , n.d.) since it consists of a series of data processing steps. A Data Pipeline has 3 key components: a source, a processing step, and a destination. The **Opony Pipeline** can be thought of as a more specific Data Pipeline, where its source is a website, its processing step is scraping/crawling and its destination is storing the data somewhere.

A simplified implementation of the **Opony Pipeline** in Python is shown in the image 14.

Figure 14 – **Opony Pipeline**

```
@celery_app.task()
def opony_pipeline(job, **configs):
    check_expired(job)
    try:
        bot = find_bot(job)
        bot.download(job)
        items = bot.scrape(job)
        publish(items)
        new_jobs = bot.crawl(job)
        generate(new_jobs)
        # Job successful, update Key Value Store.
        ...
    except RateLimitedError:
        # Postpone job for later
        ...
    except Exception:
        # Send to Dead Letter Queue
        # Job failed, update Key Value Store.
        ...
    finally:
        # Job is finished, update Key Value Store.
        # Webhook
        ...
```

Source: Personal Archive

The first thing to notice is that the pipeline is defined as a **Celery task**, as can be seen in the line **@celery_app.task()**. This means that a Celery consumer (i.e. worker) of the **Message Broker** can eventually call this function.

If a message is published to a queue of the **Message Broker** that is being consumed by a worker, some things will happen: the worker will identify if the message contains the expected format and if so, will call the function **opony_pipeline**. When the function (or task) is called, the content of the consumed message is passed as arguments to it, in this case, **job** and **configs**.

While a **job** is an abstraction that defines where and how data must be collected from, **configs** contains generic configurations which might override global configurations of Opony. For example, while a **job** might say "download data from the url *https://moodle.ufsc.br/* using user "x" and password y", **configs** might say "check if this information is already stored in the cache before downloading it from the website". A

job might finish successfully or fail, depending on whether any error is raised during the pipeline or not.

The next items explain what the **Opony Pipeline** does.

- **check_expired(job)** - each job has an expiry date. Jobs that fail for any reason (e.g. the source of data is offline) can be retried as many times as desired, always after the job fails. This is possible since the failed jobs are stored in *dead letter queues*. This may cause jobs to never be successfully finished. Therefore, expiring jobs after a while allows cleanup of the **broker** from time to time.
- **configure(configs)** - Configures the pipeline, answering the following questions for the given job:
  - Will the cache be used?
  - Which cache will be used?
  - Which **Key-Value Store** will be used?
  - Which **Proxy Provider** will be used?
  - Should scraping occur?
  - Should crawling occur?
  - Where will the scraped data be sent to?
- **find_bot(job)** - Finds the bot which is able to do the job. The index used to find bots is the **domain** of the url of a job. E.g. a job containing the url *https://moodle.ufsc.br/* would cause Opony to try to find the bot related to the domain *ufsc.br*.
- **snapshot = bot.download(job)** - Downloads data from a website following the job specification and retrieves a *snapshot*: a representation of the downloaded data. The download can happen in many ways, and it is up to the bot to decide how it will occur. Opony provides many Downloaders - classes that implement methods to help the download to happen in a reliable way, which can be used by the bots. Downloaders can work by doing direct HTTP requests or by simulating a browser and working with cookies and sessions.
- **items = bot.scrape(snapshot)** - Scrapes the *snapshot*, collecting relevant information from it (items). Most bots used BeautifulSoup (BEAUTIFULSOUP. . . , n.d.) to help the scraping process, a python library that parses HTML strings into python objects.
- **export(items)** - Exports the scraped items to the desired location. Opony always exported the items a queue in it's **Message Broker**, as shown in image 7.

- **new_jobs = bot.crawl(snapshot)** - Crawls the snapshot looking for information that could generate more jobs. If it does, store the new jobs in the *new_jobs* variable.

- **generate(new_jobs)** - Creates new tasks using the *new_jobs* variable. As explained, tasks are simply messages that are published to the broker to be consumed by a worker. This is also what brings data persistence and parallelism to Opony. Having a minimalistic **Opony Pipeline**, that does the minimum work possible, makes Opony constantly generate jobs, publishing messages to the **Message Broker**. Once the jobs are in the **Message Broker**, the bot's progress is saved in the disk, so the system can crash and nothing will be lost. Parallelism is achieved here, since a bot might generate many jobs, publishing many messages to the **Message Broker**, which will be consumed by workers. If there is only 1 worker active, the bot will be processed synchronously. However, as shown in image 7, many worker Docker containers can be run at the same time, and even in different machines, which allows true parallelism, where many workers can consume messages from the same bot.

- **register_successful_job()** - Makes a request to the **Key-Value Store** registering that a job was successfully finished. This allows the **Key-Value Store** (e.g. **Redis**) to track the progress of bots, knowing how many jobs are successful and how many still need to be processed, information important for monitoring bots (see Monitoring 5.1.8).

- **postpone_task()** - To avoid downloading data too fast from a website, Opony uses the **Key-Value Store** to register every time a download happens (see Rate Limit **??**). The error *RateLimitedError* is raised by Opony when it is downloading data too fast. The **Opony Pipeline** catches the error here and postpones the task to a later time in order to avoid this problem.

- **register_fail_job()** - Generic errors (Exception) are caught by Opony (see Monitoring 5.1.8) so that they can be correctly handled. In this case, the **Key-Value Store** (**Redis**) is updated, registering this specific job failed.

- **send_to_dead_letter_queue(job)** - Here the job is sent to a *dead letter queue*. The jobs in the *dead letter queue* can be later on republished to their original queue.

- **register_finished_job()** - The *Finally* block runs its content independently if an exception is caught or not. This commands updates the **Key-Value Store** (**Redis**) marking this job as finished.

- **webhook()** - Finally, Opony supports **Webhooks**. **Webhooks** are callback functions that allow communication between 2 or more applications. Adding a webhook here allows Opony to warn other applications that a certain job or a group of jobs (a batch) is finished. Opony is then capable of sending for example a message to a Slack group telling the team that the bot x has finished its job.

5.2.1 Command Line Interface (CLI)

Opony offers an easy-to-use CLI to interact with its resources. The CLI of Opony can be accessed by running the following command inside a session with Opony installed:

```
$ python -m opony
```

The output of this command guides the user on how to use the tool, explaining what other commands are available. The CLI has 3 main command groups:

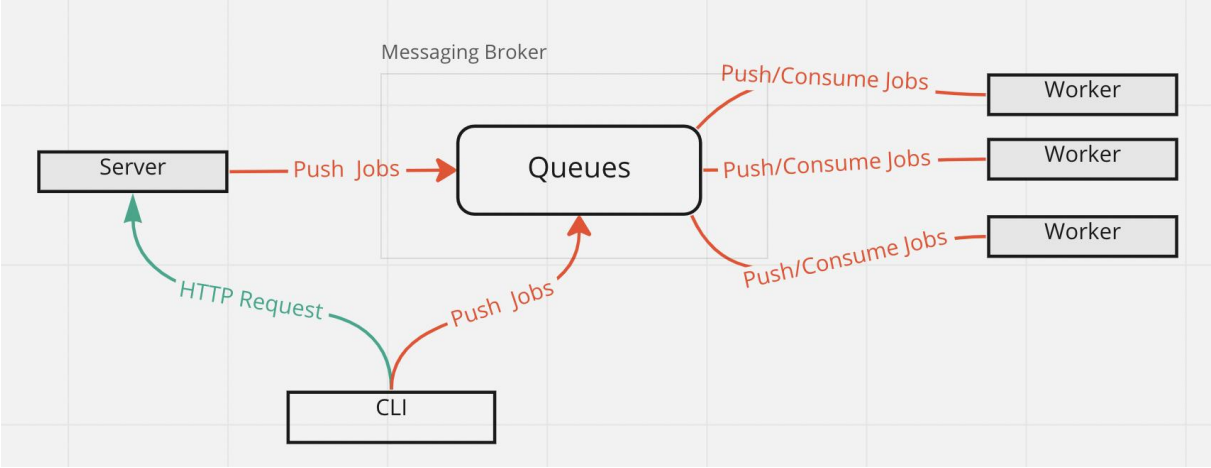
- **Bots**: Manages bots to allow easy testing. 3 commands are available here:
  - **python -m opony bots list**: provides the user with all unique identifiers of all available bots.
  - **python -m opony bots run**: runs the **Opony Pipeline** for a given bot, given an URL and some configurations as parameters. E.g. **python -m opony bots run –url="https://google.com"** would trigger Opony to look for a bot related to the domain "google.com" and send the url to it, which would run the **Opony Pipeline** and return the result.
  - **python -m opony bots report**: looks for all data scraped by bots and generated a Pandas Profiling (PANDAS. . . , n.d.) report with it. See subsection 5.2.2 for more information.

  All **Bots** commands can also be run remotely, by setting the environment variables **SERVER_HOST** and **SERVER_PORT**. This would cause a request to be performed to the configured **server** agent (to be defined in the next item), which would then run the commands using its local configurations.

  This can prove really useful since it provides a way to act on a deployed version of Opony without having access to the machine where it runs and without having to worry about how the specific configurations of the command should be set given that the deployed server configurations would be used in this case.

  The information flow for these commands can be seen in figure 15.

Figure 15 – Communication between CLI, Server, Broker and Workers



Source: Personal Archive

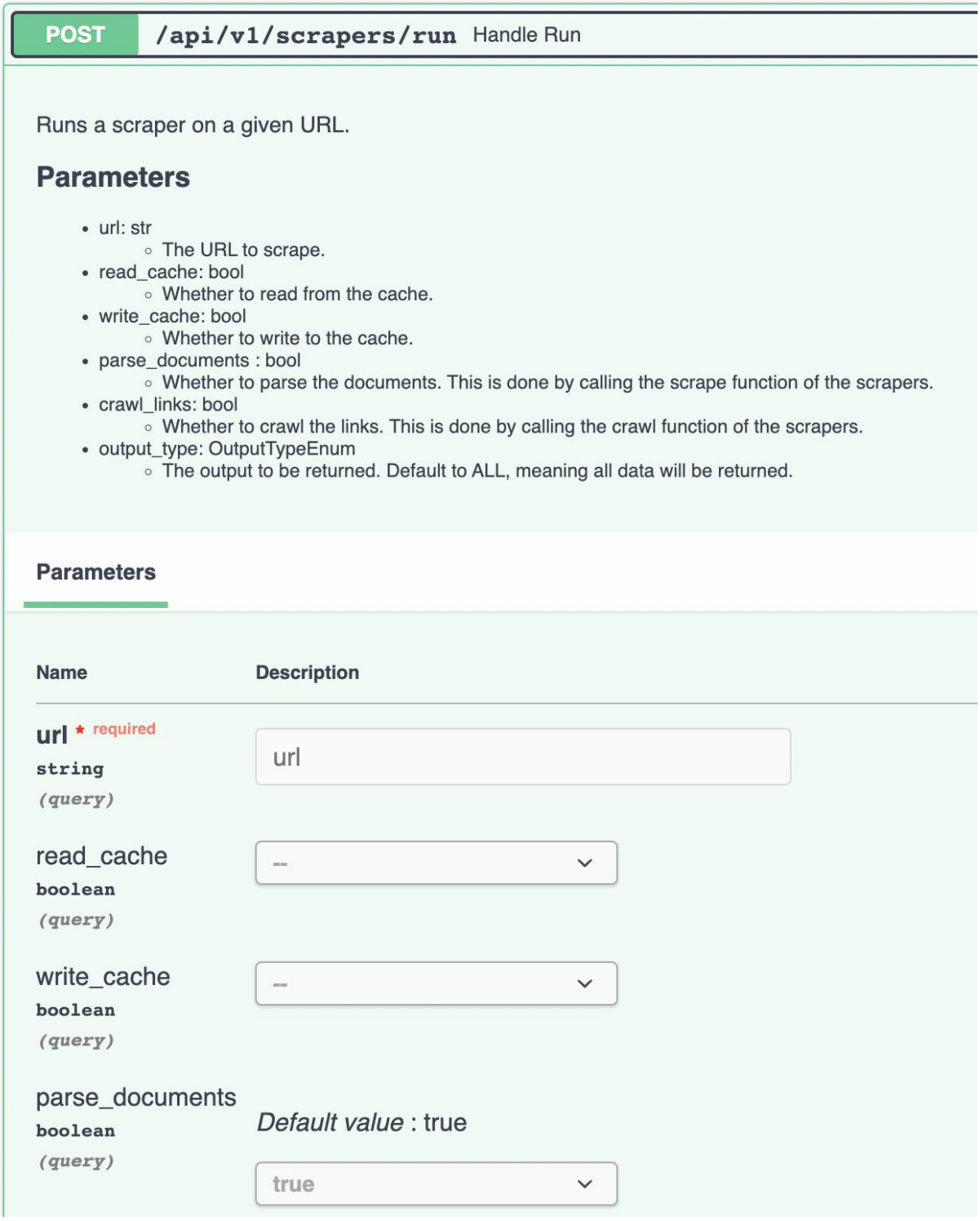
Note that CLI commands could also interact directly with the **Message Broker**, instead of always having the Server as an intermediary.

- **Agent** Creates processes that run until an external signal stops them.
  - **python -m opony agent –worker**: Creates a worker - process that waits idly until there are messages in the **Messaging Broker** to be consumed. Once a message is consumed, the worker goes through the **Opony Pipeline**.

    This command can be run many times in different sessions or even machines. Many workers can point to the same **Message Broker**, allowing scalability and distribution. If there are too many jobs in the **Message Broker**, more worker agents could be created and jobs would be completed more frequently, accelerating the data collection process.
  - **python -m opony agent –server**: A process that runs a web server and is available through an endpoint.

    The server is used mainly outside the Development environment (see **??**, as an easy way to communicate with Opony, although it is also available in Development, simply by accessing an endpoint in the *localhost* of the browser. The browser interface can be used to test, visualize and run bots. An example of an endpoint is presented in image 16.

Figure 16 – Example of Opony server UI for an Endpoint



Source: Personal Archive

All CLI commands, with the exception of the Agent group commands, are available as an endpoint as well.

The server and interface are created with the help of the popular Python

library FastAPI (FASTAPI. . . , n.d.).

  - **python -m opony agent –dev**: A process that runs both a worker agent and a server agent, one in its own thread, as shown in the diagram of image 21. This command is designed to be used in the Development environment (see subsection 5.3.2.2) and allows the developer to easily access all Opony features.

- **Jobs**

  - **python -m opony jobs list**: Allows the user to list all jobs in the broker quickly. Allow easy visualization of which jobs would be done by workers in the future.
  - **python -m opony jobs create**: Creates jobs for a given bot, aiming to collect all data of the bot's related website. The main required argument for this command is the **domain** argument, which triggers Opony to look for a bot that matched the given domain, using the same mechanism described at the command **python -m opony bots run**. Differently from **python -m opony bots run**, this command does not trigger the **Opony Pipeline**, as it only creates **jobs** to be published to the broker and consumed by workers in the future, which would then run the **Opony Pipeline**, possibly generating more jobs. The jobs created by this command are defined based on bot attributes, which define the starting point of the **batch**. A **batch** is defined as a group of jobs, and jobs usually contains a **batch_id**, used for monitoring the batch of jobs of the bot. Once the first jobs are created and published, workers can consume them and generate more jobs in the **Opony Pipeline**, which would also contain the same *batch_id* as the predecessor job.
  - **python -m opony jobs reparse**: Reparses downloaded content stored in the **Object Store**. It's important to remember here that there are 3 main steps in the **Opony Pipeline**: download, scrape and crawl. This command aims to skip the download step, fetching the data from the **Object Store** (e.g. **Minio**) instead of downloading it from the website of interest. This allowed flexibility since sometimes data is downloaded but it was not yet clear how it should be scraped or crawled. This way, data can be collected before clients are sure of how the data should be structured and sent to them.
  - **python -m opony jobs retry**: Allows failed jobs to be reprocessed by the **Opony Pipeline**. Sometimes jobs fail: the website can be offline, there might be a bug in the bot, or something totally unexpected might happen. When any of these scenarios occur, jobs are sent to a separate queue in the *broker*, called *dead letter queue*. This way, failed jobs are not lost, but just stored in

a separate place. This allows developers to investigate the problem and fix it. After that, running this command triggers Opony to take jobs from *dead letter queue* and publish them back to their origin queue, triggering the **Opony Pipeline** once again and successfully finishing the jobs.

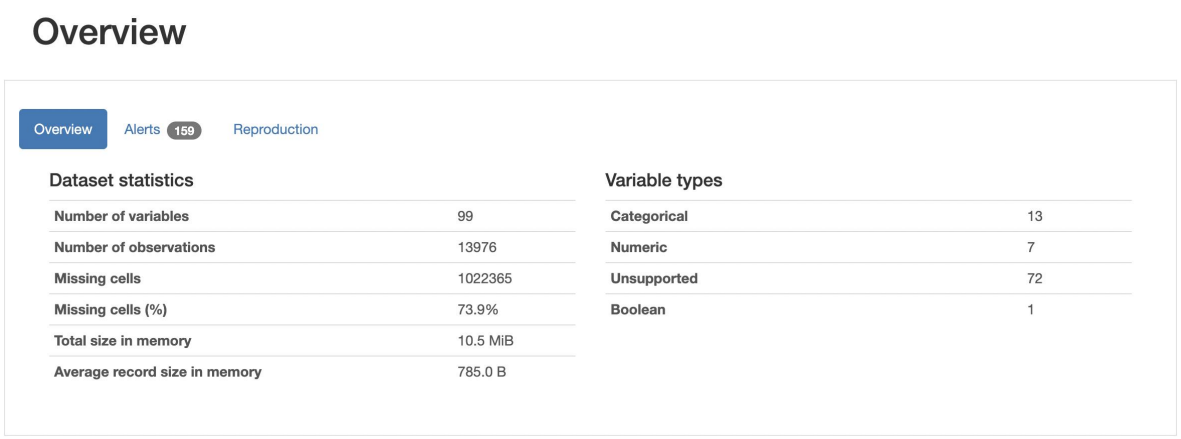
Just like the **Bots** commands, **Jobs** commands also have the capability of performing a request to a given server by setting the environment variables **SERVER_HOST** and **SERVER_PORT**.

5.2.2 Data Validation

Opony is capable of generating a data report using Pandas Profiling (PANDAS. . . , n.d.). The aim here is to allow the developer to easily debug a bot by looking at the data it outputs and statistics about the data. Many statistical indicators are available in the report, from generic statistics which describe all data collected to deeply detailed insights regarding a single attribute.

Image 17 shows generic statistics about the data, which is always formatted as a table. By analyzing the shape and missing data quantities, the developer can sometimes already infer if there is a problem with the bot or not.

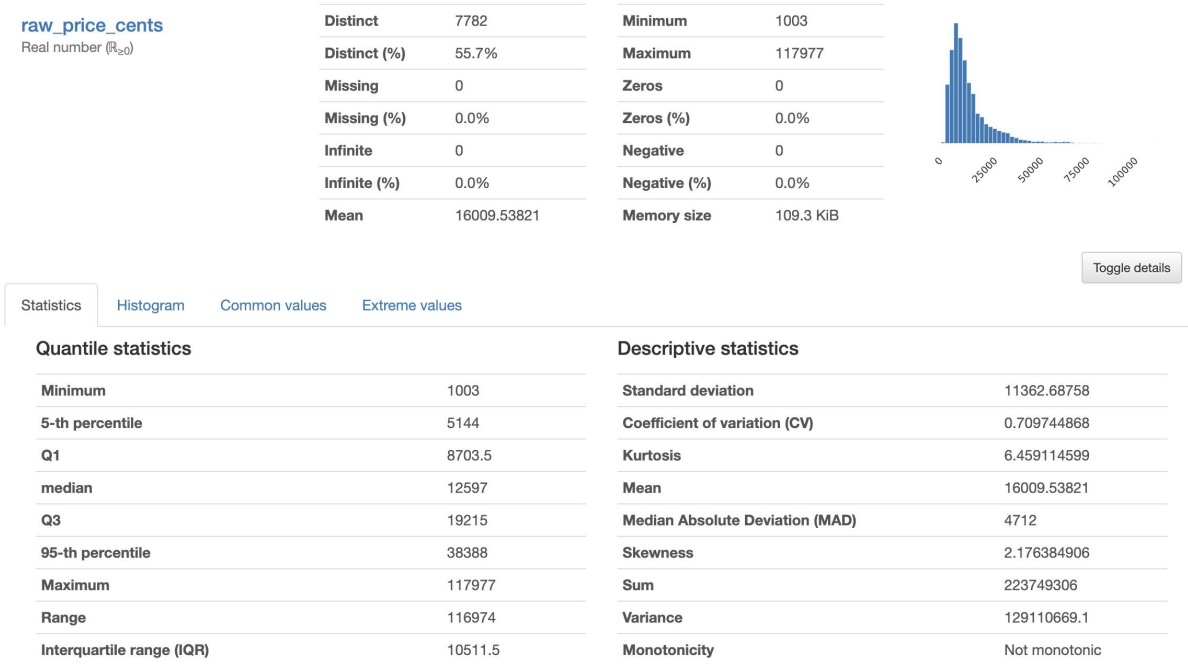
Figure 17 – Validation Report Overview



Source: Personal Archive

Image 18 shows how the report allows for a deep visualization of a single variable (in this case the variable is called *raw_price_cents*). Visualizing statistics like a histogram, minimum and maximum values, amount of missing values, mean, median, percentiles, sum, variance and other metrics allows the developer to identify flaws in the bot quickly.

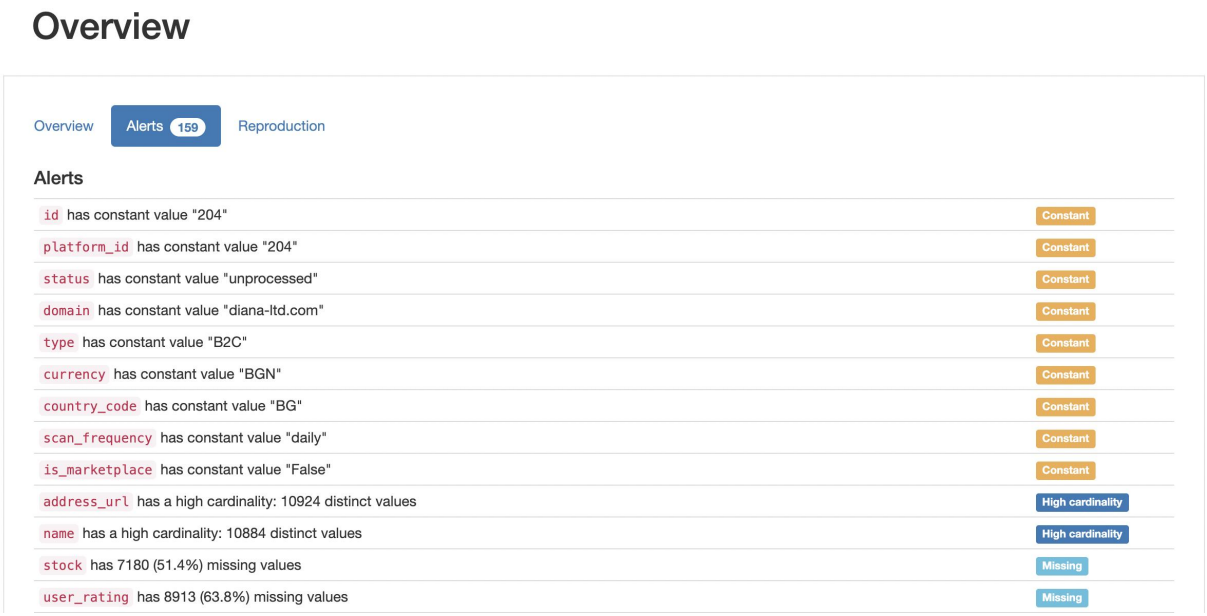
Figure 18 – Validation Report Attribute View



Source: Personal Archive

Finally, image 18 provides alerts that might also identify common errors in the data. Each column in the table passes through standard checks, that generate alerts if they fail. Some examples are: checking if a column is constant, checking if a categorical column has too many different values (high cardinality) and checking if too many values of a given column are empty.

Figure 19 – Validation Report Alerts



Source: Personal Archive

5.3 DEPLOYMENT

Software deployment is the process of making an application available for users. Deployment is usually broken down into some steps: install the software and its dependencies in the host machine(s); test the software to make sure it is working as expected; run it using the desired configurations; monitor its performance.

As the deployed system run on a different machine than the one used for development, it is essential to make sure both use compatible operating systems and software dependencies, otherwise, the deployment may be successful in one machine, but not in the other. To avoid this problem, every deployment outside the Development environment (to be explained at section 5.3.2.2) happens inside Docker containers, explained in section 5.3.1.

To automate the deployment process, Gitlab's CI/CD (CI/CD..., n.d.) mechanism is used. CI/CD stands for continuous integration/continuous deployment. It is the process of deploying the code of a repository continuously so that the latest version of the code is available as soon as possible.

In the case of Opony, each time a Git push operation happens to Gitlab's Opony repository, the CI/CD pipeline is triggered. The CI/CD is defined in a configuration file inside Opony, that follows certain standards that Gitlab establishes. Opony's configuration file defines 3 steps:

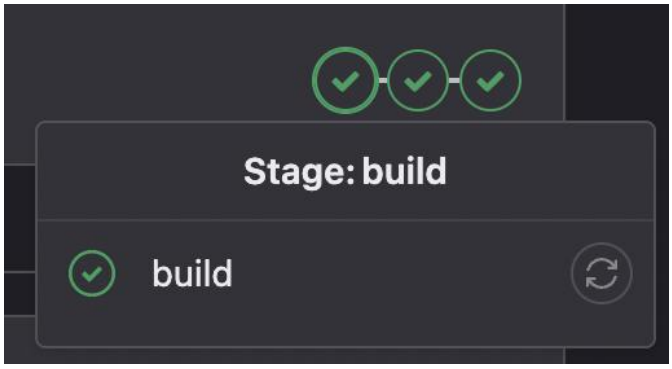
- Build: installs the application dependencies and encapsulates them in a file, called

a Docker image (to be explained at 5.3.1). The installation is done through Pipenv (PIPENV. . . , n.d.), a virtual environment and package manager tool. The Docker image is stored in Gitlab's registry, a database for storing Docker images, that can be used to deploy Opony from anywhere.

- Test: runs the unit tests defined in the application.
- Publish: Starts the application in the Homologation environment (see section 5.3.2.3) using the built image.

Gitlab provides a UI for the CI/CD pipeline of every branch in the repository, as shown in image 20.

Figure 20 – CI CD Gitlab's UI



Source: Personal Archive

By clicking in each pipeline step, a developer is able to visualize the logs of the CI/CD, identifying for example, why a certain unit test failed.

5.3.1 Docker

Docker is a software that runs on top of the machine's operating system (OS) kernel to provide application isolation in the form of containers. As described by Docker itself, "Docker containers are a standardized unit of software that allows developers to isolate their app from its environment, solving the 'it works on my machine' headache".

Unlike virtual machines, docker emulates the operating system in which the application is run, not the hardware itself. This allows many containers, each containing totally isolated software, to be quickly deployed to one or many host machines, enabling fast deployment and scalability.

Docker containers are run from Docker images. A Docker image is a file that provides resources to a container so that it can run. A Docker image can be built from a Dockerfile. The code below is an example of a Dockerfile.

```
FROM alpine:3.4
RUN apk add curl
```

If a Docker image is built using this example Dockerfile, the image will contain the alpine (ALPINE. . . , n.d.) distribution of the Linux operating system (OS) and the curl (CURL. . . , n.d.) command, which allows developers to transfer data to and from servers. This is all the container will have, the Alpine OS and the curl command. The container can be run on any machine that has Docker installed. The machine might run on any operating system, and the Docker container will simply work. That is the power of Docker, as it provides an easy way to release code anywhere.

Opony has it's own Dockerfile, and an Opony Docker image is built each time the CI/CD Pipeline is successfully finished.

5.3.2 Environments

The data collection system can be deployed in many different ways, configured in different environments, each one with its own purpose. In each environment, the system adapts itself to supply the user with its needs. A single environment variable called *ENVIRONMENT* controls which environment is used.

This subsection first presents a superficial comparison between the environments, then moves on to explain each one deeply.

5.3.2.1 Comparing the Environments

The differences between Opony instances running in different environments are presented here.

Table 2 summarizes the differences in characteristics between the environments and table 3 shown differences in services.

In table 2 an **X** is used to mark whether each environment has a certain characteristics.

	Environments		
Characteristics	Development	Homologation	Production
Good for developing Bots	X		
Used by developers	X	X	
Fast setup time	X	X	
Persists state	X		X
Allows data validation	X	X	X
Accessible from anywhere		X	X
Good for running long-lasting processes		X	X
Uses Docker		X	X
Deployed in a server		X	X
Runs many Opony workers			X
Distributed			X
Scalable			X

Table 2 – Comparison Between Environments

	Environments		
Service	Development	Homologation	Production
Key-Value Store	Memory	Redis	Redis
Object Store	Filesystem	Minio	Minio
Message Broker	Filesystem	RabbitMQ	RabbitMQ
Backend		Elixir Application	Elixir Application
Data Validation	Data Validation Report	Data Validation Report	Metabase
Logs	Terminal	Containers logs	Loki + Grafana
Proxy Provider		Maestro + Backups	Maestro + Backups
Cron Service			N8N

Table 3 – Comparison Between Services in Environments

The comparisons show that each environment is different, making some suitable for situations where others are not. It's also noticeable that there is an increase in complexity when moving from the Development environment to the Production one, which is natural since Production is supposed to do the heavy lifting, demanding more services and power.

5.3.2.2 Development

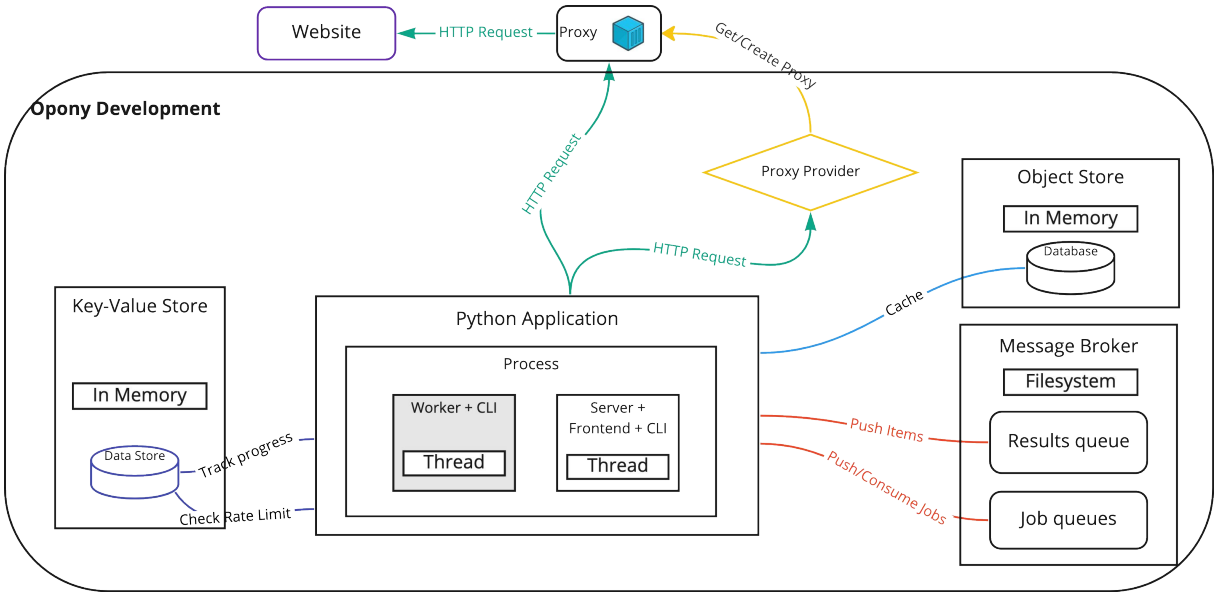
In the Development environment, it is essential that the user can quickly create and test bots. In order to do so, Opony has the capability of adapting itself to reduce its complexity and help the developer.

Firstly, Docker is not required in Development. The **Python Application** runs directly in the host's machine by the virtual environment Python interpreter. This reduces the complexity to run the system but makes it more vulnerable to incompatibilities between machines, since different machines run different operating systems and hardware. This is solved by forcing users to install the system in a virtual environment, using Pipenv.

As can be seen in the image 21, many of the services used in production (represented by image 7) are not available here. Instead, the host machine's memory and

filesystem are used. This replacement can be done since the system in Development did not need to be scalable, distributed or accessible through many machines. It simply needed to run smoothly in the user's machine.

Figure 21 – Micro services overview in Development



Source: Personal Archive

The following replacements are made:

- **Message Broker**: instead of **RabbitMQ**, messages are simply stored in the host machine's filesystem, and can be easily accessed by the developer.
- **Object Store**: instead of **Minio**, the machine's filesystem is used, where documents are stored in the disk.
- **Key-Value Store**: instead of **Redis**, the memory is used, both for tracking each bot and for storing rate limit information.
- **Grafana Loki**: not used in development since all logs are available on the user's terminal.
- **N8N**: not used, since there is no need to run bots periodically in development
- **Backend**: not used. The **Backend** is responsible for consuming all the data from the **Message Broker** and storing it in the **Central Database** in Production. In Development, there is no need to do so, since the idea here is not to collect a lot of data.

This allows Opony to fully run just by installing it, with no need to run microservices.

The easy substitution of services is possible due to well-defined object-oriented programming interfaces. Taking the **Object Store** as an example, an abstract class is defined called *ObjectStore* (see image 22), which defines methods (*put*, *get*) that have to be implemented by the concrete classes.

Figure 22 – Object Store Interface

```python
from abc import ABC, abstractmethod
from typing import Iterator, Optional

from opony.core.snapshots import WebsiteSnapshot


davidsand, 2 days ago | 2 authors (You and others)
class ObjectStore(ABC):
    @abstractmethod
    def put(self, *, key: str, website_snapshot: WebsiteSnapshot):
        pass

    @abstractmethod
    def get(self, *, key: str) -> Optional[WebsiteSnapshot]:
        pass
```

Source: Personal Archive

This approach is also used for the **Key-Value Store** and the **Proxy Provider**. The concrete classes to be used by Opony would be then controlled by the already mentioned *ENVIRONMENT* environment variable.

Generally, inheritances are avoided in the **Python Application**, since it may cause readability issues and make the code cumbersome. Instead, a more functional approach is used, with the use of dependency injection to allow easy testing of the software.

After installation using Pipenv, the user can directly run the command:

```
$ python -m opony agent --dev
```

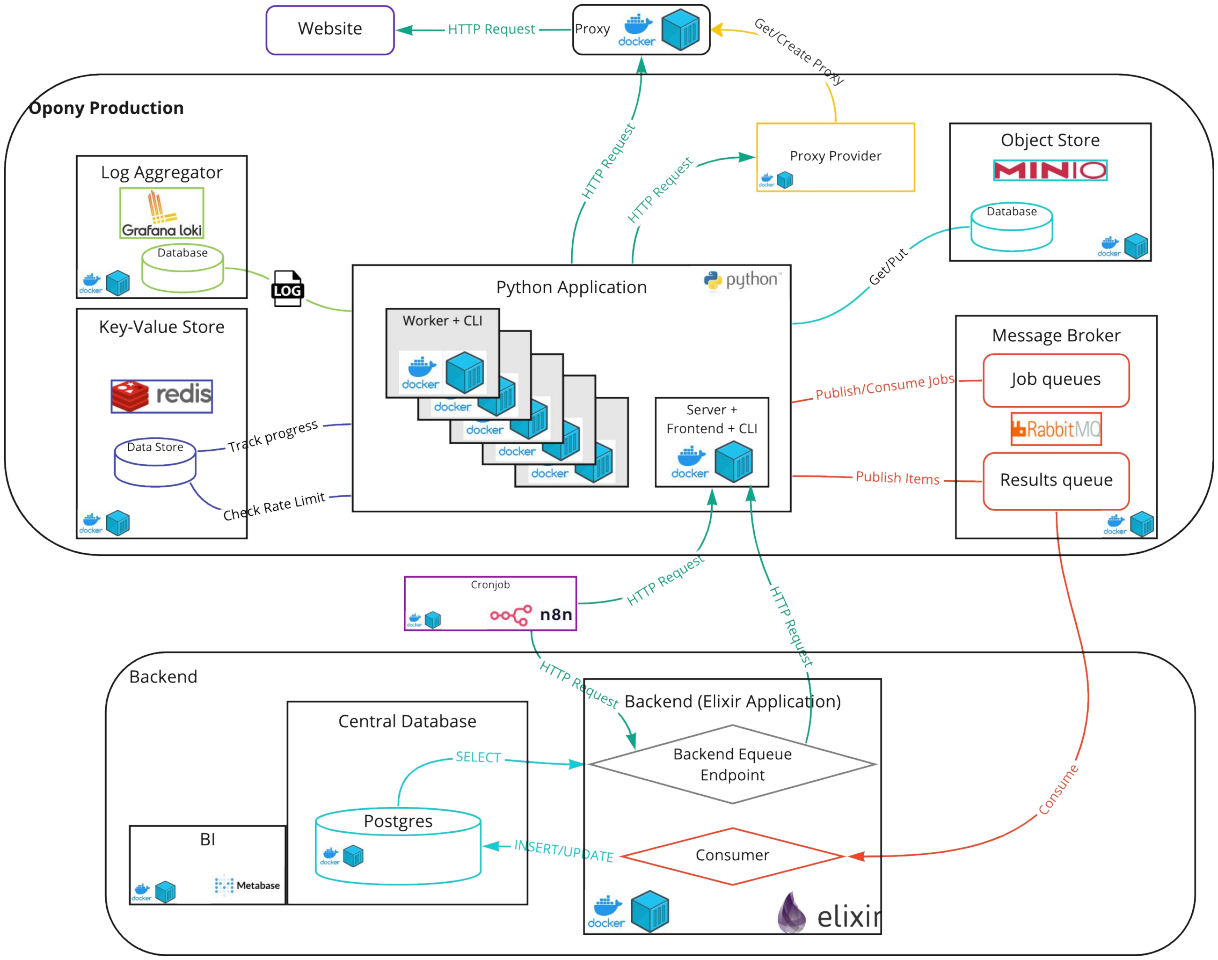
This ran both a worker and a server, each in a process, as shown in figure 21.

5.3.2.3 Homologation

The Development environment does not replicate completely the production environment. As mentioned, it does not run in Docker containers, the backend is not available and some services are replaced. Also, it is not meant to run bots for a prolonged period of time, but to develop bots quickly. Running bots for long periods in the Development environment would require the user's machine to stay awake during this period, which is not ideal. Finally and more importantly, access to the Development environment is restricted to the developer who has access to the local machine running Opony, i.e. this prevents someone else to check how the bot is running and how its data looks.

To solve these problems, another environment was defined: Homologation. Its objective is to serve as a collaborative testing environment, by approximating the bot to the Production environment and allowing many developers to interact with bots and validate their data. Homologation includes some of the production environment services shown in image 7, but has a limited number of worker containers. Its infrastructure is shown in image 23.

Figure 23 – Micro services overview in Homologation



Source: Personal Archive

The main differences here are:

- Only one worker container is available. This limits the speed of data collection, but since the Homologation environment is not meant to collect data in scale, this is not a problem.
- No log aggregation service is available. Since in Homologation there is not so much work being done by Opony workers, and therefore the amount of logs generated is of low quantity, a log aggregation service is not necessary. Instead, the Docker container logs can be accessed directly.

As mentioned in the work methodology section (4.3.4), Homologation was used to perform the final data validation of a given bot. The data validation report, shown in subsection 5.2.2 is available in a URL, allowing access to it by any developer.

Beyond the data validation URL, there are also other URLs available in the Homologation, all automatically created by the CI/CD publish (5.3) step. Examples of

URLs available in Homologation are: **Opony Server** URL, to allow the user to interact with Opony; **Message Broker** user interface URL; Database UI URL, where users can make SQL queries to visualize the collected data; **Backend** server URL.

Also, every URL contains the Git branch name in it, so it is possible to have an isolated Opony instance (running in the Homologation environment) available for each branch in the repository, allowing developers to make changes to Opony and easily test them just by pushing commits to Gitlab, which triggers the CI/CD pipeline and creates a fresh Opony instance in Homologation from scratch for that branch.

The Opony instance created is fresh since in Homologation no data is stored in disk, so starting Opony by pushing changes to Gitlab and triggering the CI/CD pipeline clears all data that was previously in that branch's Opony instance and starts everything from zero. Therefore, all messages in the **Message Broker** are lost, as well as all data in **Redis** and in the **Backend**. This allows a bot to be tested with no external interference, in an isolated way, making debugging much easier than in Production.

5.3.2.4 Production

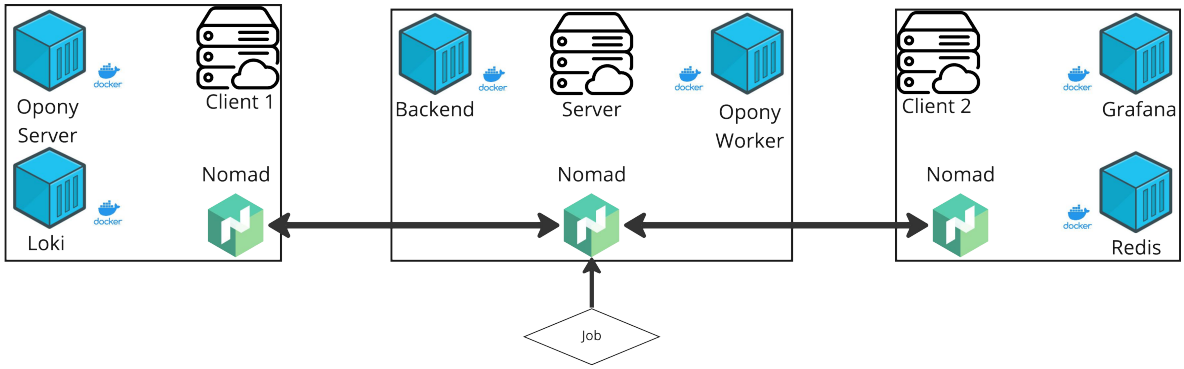
Finally, Opony in the Production environment runs all services shown in the image 7, which provides the full power and scalability of the Framework.

To run Opony Production, many Docker images are used. While some services are built in-house, as well as their respective Docker images, other services are open-source, and their Docker images are pulled from public repositories.

To run Opony containers using the Docker images, these must be pulled by the server(s) that will host the system. However, allocating the containers is not a simple task, as it requires: managing secret configuration in a safe way; configuring networks to allow communication between the services and access to services; allocating the containers in an efficient way not to waste the server's resources; defining restart logic in case the containers stop at some point due to an error; distributing containers throughout many machines since one is not enough for Opony; etc.

In order to do so, a container orchestrator/cluster tool is used: Nomad (NOMAD. . . , n.d.). Image 24 represents a cluster containing 3 nodes (e.g. machines): Client1, Server and Client2 - where Nomad is installed in each one of them, establishing communication between the machines and therefore establishing a cluster.

Figure 24 – Nomad Cluster



Source: Personal Archive

All nodes run Docker Containers, which in the case of Opony, might be **Redis**, **Loki**, **Grafana**, etc. To run a Docker Container in the cluster, a **Nomad job** must be defined. A **Nomad job** is a definition of how to run services. It might be formalized in a configuration file (called Nomad file) that tells Nomad how to run a given service.

Figure 25 – Nomad File

```
job "opony" {
  group "opony-workers" {
    count = 2

    restart {
      attempts = 3
    }

    task "worker" {
      driver = "docker"

      config {
        image = "opony-worker-docker-image"
        args = ["python", "-m", "opony", "agent", "--worker"]
        logging {
            config {
                loki-external-labels="application=opony"
            }
        }
      }

      resources {
        cpu        = 500
        memory     = 200
        memory_max = 500
      }

      env {
        ENVIRONMENT = "production"
      }
    }
  }
}
```

Source: Personal Archive

Image 25 shows a simple example of what a Nomad configuration file might look like. There are 3 main structures in the file: **job**, **group** and **task**. A job defines what will be run by Nomad once it receives these configurations. It might define many **groups**, each one to be run in a single node, but different **groups** might run in different nodes. Each group then defines **tasks**, which creates a unit of work, such as a Docker container.

In the example of image 25, the job **opony** asks Nomad to run all tasks of group **opony-workers** twice, as indicated by the attribute **count**. The group also defines a restart behavior, telling all tasks in the group to attempt restart 3 times, in case the task

fail for some reason.

The **worker** task defines what will be run. In this case, docker will be used to run a container using the image **opony-worker-docker-image**. The container will be started using the command **python -m opony agent –worker** and its logs will contain the label **application=opony**.

Also, the task defines **resources**, telling Nomad that it uses usually 500MHz of CPU and 200MB of memory, and if the 500MB memory threshold set by **memory_max** is reached, Nomad is allowed to terminate the task. This feature was very useful for Opony, since a memory leak happened at some point, making opony workers consume much more memory than expected. Nomad killed the leaked containers and avoided possible server crashes due to memory overflow.

Finally, the block **env** accepts environment variables to be set for the container. In the example of image 25, only the environment variable **ENVIRONMENT** is set, telling Opony to run in the production mode.

Once the **job** is ready for deployment, it can be sent to the Server node, as shown in image 24. The Server is the brain of the cluster, deciding in which node the tasks of the job should be run.

Nomad also provides an user interface, allowing users to deploy jobs, visualize resources used by nodes and tasks, see logs of tasks and interacts with tasks using a shell.

Opony was one of the first projects related to Jungsoft to be deployed using Nomad. Deploying Opony using it was a join effort of Jungsoft and the author of this document, and it served as a proof of concept of the new orchestration tool and validated that it was indeed very powerful and relatively easy to use. After Opony, many other projects were also migrated to Nomad by Jungsoft, and it is currently Jungsoft's standard deployment tool.

5.4 FINAL COMMENTS

This chapter presented a deep explanation of how the solution was implemented. It explained which microservices Opony uses, how the **Python Application** work, how Opony interacts with the **Backend**, how it can be used by developers through the CLI and the Opony server UI and how it can be deployed in many environments. The next and final chapter concludes the work done, explaining how Opony improved the data collection solution of the client.

6 CONCLUSION

This chapter concludes the report, giving final remarks on the work done.

The objectives presented in section 1.1 were successfully implemented, where the old system was successfully replaced by the new one.

Many developers are using/have used Opony to create more than a hundred bots performing more than a million daily requests and providing millions of daily data points.

The number of bot developers in the team has increased over time, where it was reported that the tool is easy to use and powerful, even for programmers with not much experience. The same feedback was given in favor of Opony when the developers that worked both on the old data collection tool and on Opony were asked to compare the 2 systems.

The data provided by Opony is reliably and continuously inserted in the central database. Opony handles errors very well, being able to endure code bugs, errors in services used and server downtime periods with minimum loss of data and performance. The data is used constantly by the backend to provide structured information to many products of the client.

The system can be easily observed and monitored through many tools, such as Grafana, Metabase and Slack, allowing quick response time for problem-solving and a centralized view of the distributed system, used to support technical and business-level decisions. A quick response time for bugs in bots is crucial to provide good data to clients continuously and has been proven to be a reality since bots are many times fixed before they run 2 times with errors, a feat which was rarely possible in the old system.

The client is very satisfied with the implemented solution and will continue using it in the next years. Jungsoft's goal - to help clients to build and validate products and then hire a replacement team when the time comes - was also accomplished. Opony is now used by developers hired by the client, while the author of this work and Jungsoft have minimum interference in it and soon will have none.

REFERENCES

AGILE - The Agile Process 101: Understanding the Benefits of Using Agile Methodology. [S.l.: s.n.]. Available from: https://www.nvisia.com/insights/agile-methodology.

ALPINE - Linux distribution. [S.l.: s.n.]. Available from: https://www.alpinelinux.org/.

BEAUTIFULSOUP - a Python library that makes it easy to scrape information from web pages. [S.l.: s.n.]. Available from: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

BRIGHTDATA - Award-winning proxy networks, powerful web scrapers, and ready-to-use datasets for download. [S.l.: s.n.], 2022. Available from: https://brightdata.com/.

CELERY - Python task queue library. [S.l.: s.n.], 2022. Available from: https://docs.celeryq.dev/en/stable/getting-started/introduction.html.

CI/CD - a tool for software development using the continuous methodologies. [S.l.: s.n.]. Available from: https://docs.gitlab.com/ee/ci/.

CLICKUP - One app to replace them all. [S.l.: s.n.]. Available from: https://clickup.com/.

CURL. [S.l.: s.n.]. Available from: https://curl.se/.

DATA-PIPELINE - What Is a Data Pipeline? [S.l.: s.n.]. Available from: https://hazelcast.com/glossary/data-pipeline/.

ELASTIC - Search. Solve. Succeed. [S.l.: s.n.]. Available from: https://www.elastic.co/pt/what-is/elasticsearch.

ELIXIR - Elixir is a dynamic, functional language for building scalable and maintainable applications. [S.l.: s.n.]. Available from: https://elixir-lang.org/.

FASTAPI - a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. [S.l.: s.n.]. Available from: https://fastapi.tiangolo.com/.

GIT - A free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. [S.l.: s.n.]. Available from: https://git-scm.com/.

GIT-BRANCHES: List, Create, Switch to, Merge, Push, Delete. [S.l.: s.n.]. Available from: https://www.nobledesktop.com/learn/git/git-branches.

GITHUB - Harnessed for productivity. Designed for collaboration. Celebrated for built-in security. Welcome to the platform developers love. [S.l.: s.n.]. Available from: https://github.com.

GITLAB - From planning to production, GitLab brings teams together to shorten cycle times, reduce costs, strengthen security, and increase developer productivity. [S.l.: s.n.]. Available from: https://gitlab.com.

GOOGLEBOT. [S.l.: s.n.]. Available from: https://developers.google.com/search/docs/crawling-indexing/googlebot.

GRAFANA - Operational dashboards for your data here, there, or anywhere. [S.l.: s.n.]. Available from: https://grafana.com/.

HTTP - What is HTTP? [S.l.: s.n.]. Available from: https://sematext.com/glossary/http-requests/.

JUNGSOFT - We don't just write software, we build fascinating digital products. [S.l.: s.n.]. Available from: https://jungsoft.io/.

KOMBU - Messaging library for Python. [S.l.: s.n.], 2022. Available from: https://github.com/celery/kombu.

LOGQL - Log query language. [S.l.: s.n.]. Available from: https://grafana.com/docs/loki/latest/logql/.

LOKI - A horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. [S.l.: s.n.]. Available from: https://github.com/grafana/loki.

MEDIUM - Developing a distributed web scraper using Celery. [S.l.: s.n.], 2020. Available from: https://medium.com/nam-r/developing-a-distributed-web-scraper-using-celery-24c17df4cc63.

MESSAGE-BROKERS - Key Models, Use Cases Tools Simplified 101. [S.l.: s.n.], 2022. Available from: https://hevodata.com/learn/message-brokers/#intro.

METABASE - The BI tool with the friendly UX and integrated tooling to let your company explore data on their own. [S.l.: s.n.]. Available from: https://www.metabase.com/.

MICROSERVICES - What are microservices? [S.l.: s.n.]. Available from: https://microservices.io/.

N8N - The workflow automation platform that doesn't box you in, that you never outgrow. [S.l.: s.n.]. Available from: https://n8n.io/.

NOMAD - Orchestration Made Easy. [S.l.: s.n.]. Available from: https://www.nomadproject.io/.

PANDAS Profiling - A Python library that generates profile reports from a pandas DataFrame. [S.l.: s.n.]. Available from: https://github.com/ydataai/pandas-profiling.

PIPENV - Python Development Workflow for Humans. [S.l.: s.n.]. Available from: https://pypi.org/project/pipenv/.

POSTGRES - The World's Most Advanced Open Source Relational Database. [S.l.: s.n.]. Available from: https://www.postgresql.org/.

PYDANTIC - Data validation using Python type hints. [S.l.: s.n.]. Available from: https://github.com/pydantic/pydantic.

REACT - A JavaScript library for building user interfaces. [S.l.: s.n.]. Available from: https://reactjs.org/.

REDIS - The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker. [S.l.: s.n.]. Available from: https://redis.io/.

SCALING - Horizontal Vs. Vertical Scaling: How Do They Compare? [S.l.: s.n.]. Available from: https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling.

SCRAPY - An open source and collaborative framework for extracting the data you need from websites. [S.l.: s.n.]. Available from: https://scrapy.org/.

SENTRY - Take action on broken lines of code, crashes, and busted API calls with the only developer-first app monitoring platform built to give you answers - not clues. [S.l.: s.n.]. Available from: https://sentry.io/.

SMARTPROXY - Access competitive business data anywhere in the world with proxies. [S.l.: s.n.], 2022. Available from: https://smartproxy.com/.

TCP - What is Transmission Control Protocol TCP/IP? [S.l.: s.n.]. Available from: https://www.fortinet.com/resources/cyberglossary/tcp-ip.

TIMESCALE - Postgres for time-series. [S.l.: s.n.]. Available from: https://www.timescale.com/.

UDP - What is User Datagram Protocol (UDP/IP)? [S.l.: s.n.]. Available from: https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/.

WEB-BROWSER - What is a Web browser? [S.l.: s.n.]. Available from: https://www.techopedia.com/definition/288/web-browser.

ZYTE - Web Data Made Easy. [S.l.: s.n.]. Available from: https://www.zyte.com/.