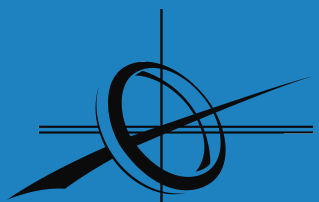
POLITÉCNICA



Universidad Politécnica de Madrid

ETSI SISTEMAS INFORMÁTICOS

Knag: A cloud native algorithmic trading platform

Proyecto Fin de Grado

Grado en Ingeniería del Software

Autor:

Alberto Zugazagoitia Rodríguez

Tutor:

Guillermo Iglesias Hernández

Curso 2022/23

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Knag: A cloud native algorithmic trading platform

Proyecto Fin de Grado

Grado en Ingeniería del Software

Curso académico 2022-2023

Autor:

Alberto Zugazagoitia Rodríguez

Tutor:

Guillermo Iglesias Hernández

To the FOSS community, for building software altruistically.

Abstract

Algorithmic and automated trading is out of the scope of retail investors, due to the technical complexity of it as well as the costs of development. Some brokers offer this service but at an upfront cost and a monthly fee. Users depend on the broker's platform and servers to execute the trades, giving the broker too much control over the user's data and trading activity.

The solution implemented in this project is a customizable FOSS automated trading platform that can be used by retail investor communities to implement their trading strategies and by individual investors to automate their trading activity.

The platform is designed to be highly scalable and available, developed with a cloud-first approach, and using the latest technologies in the field of distributed systems, such as Kubernetes and public cloud providers' specific services.

The specific implementation of the trading bot is out of the scope of this project. Nonetheless, the platform is designed to be easily extensible and customizable, through the use of docker containers.

Resumen

El trading algorítmico y automatizado está fuera del alcance de los inversores minoristas, debido a su complejidad técnica y a los costes de desarrollo. Algunos brokers ofrecen este servicio pero a un coste inicial y una cuota mensual. Los usuarios dependen de la plataforma y servidores del broker para ejecutar las operaciones, dando al broker demasiado control sobre los datos y actividad de trading del usuario.

La solución implementada en este proyecto es una plataforma de trading automatizado FOSS y personalizable que puede ser usada por comunidades de inversores minoristas para implementar sus estrategias de trading y por inversores individuales para automatizar su actividad de trading.

La plataforma está diseñada para ser altamente escalable y disponible, desarrollada con un enfoque cloud-first y usando las últimas tecnologías en el campo de los sistemas distribuidos, como Kubernetes y servicios específicos de proveedores de cloud públicos.

La implementación específica del bot de trading está fuera del alcance de este proyecto. No obstante, la plataforma está diseñada para ser fácilmente extensible y personalizable, mediante el uso de contenedores docker.

Contents

Abstract II

Resumen III

1 Introduction **1**

1.1 Context 1

1.2 Motivation 2

1.3 Objectives 3

1.3.1 Primary objectives 3

1.3.2 Secondary objectives 3

2 State of the art **5**

2.1 Cloud Native Applications 5

2.1.1 DevOps 6

2.1.2 Code Automation 6

2.1.3 Containerised Applications 7

2.1.4 Micro-services Architectures 8

2.1.5 Container Orchestration 9

2.2 Cloud Infrastructure 9

2.2.1 Traditional Infrastructure 9

2.2.2 Infrastructure as Code 9

2.2.3 Infrastructure as a Service 10

2.2.4 Platform as a Service 10

3 Project development **11**

3.1 Technology Stack 11

3.1.1 Containerisation 11

3.1.2 Kubernetes and Helm 12
3.1.3 Programming Languages in the Services 12
3.1.4 Programming Language in the Web Interface 12
3.2 System Architecture 13
3.2.1 Architectural Design 13
3.2.2 Development Environment 15
3.2.3 Deployment 16
3.3 Implementation 18
3.3.1 Users Service 19
3.3.2 Vault Service 21
3.3.3 Trading Bot Manager Service 21
3.3.4 Web User Interface 22

4 Results **23**

4.1 Resulting Product 23
4.2 Achieved Objectives 24
4.2.1 Primary Objective 1: Design a platform that is highly scalable and highly available 24
4.2.2 Primary Objective 2: Design a platform that is easily extensible and customizable 25
4.2.3 Secondary Objectives 25

5 Conclusions **27**

5.1 Conclusions 27
5.2 Social Impact 27
5.3 Future Lines 28

Bibliography **30**

List of Figures

1.1 Growth of Algorithmic trading as a percentage of market volume. Sourced from [1]. 1

2.1 Question about VCS use in JetBrains Developer Survey 2021. Sourced from [2]. 7

3.1 System Architecture Diagram 14

3.2 Development Environment diagram 15

3.3 GitHub Actions Workflows diagram 17

3.4 Users Service Simplified Class Diagram 19

3.5 Sequence Diagram of a User registration 20

4.1 Screenshots of the resulting webpage 23

Acronyms

API Application Programming Interface. 8, 10, 13, 15, 16, 18–23, 26

CD Continuous Delivery. 6, 7, 10, 26

CI Continuous Integration. 6, 7, 10, 17, 26

DSL Domain-specific language. 10

FOSS Free and open-source software. 3

HA High Availability. 9

I/O Input/Output. 12

IaaS Infrastructure as a Service. 6, 10, 18, IV

IaC Infrastructure as Code. 6, 9, 10, IV

IDE Integrated Development Environment. 19

JVM Java Virtual Machine. 12, 18

JWT JSON Web Token. 14, 19, 21

mDNS mutual Domain Name System. 15

OCI Open Container Initiative. 16

OS Operating System. 7, 8, 25

PaaS Platform as a Service. 6, 10, 16, 18, IV

REST Representational state transfer. 8, 13, 15, 19, 21

SaaS Software as a Service. 13, 23

SHA Secure Hash Algorithm. 17

TLS Transport Layer Security. 14

UI User Interface. 15, 20

VCS Version Control System. 6, 7, VI

VM Virtual Machine. 7, 10, 15

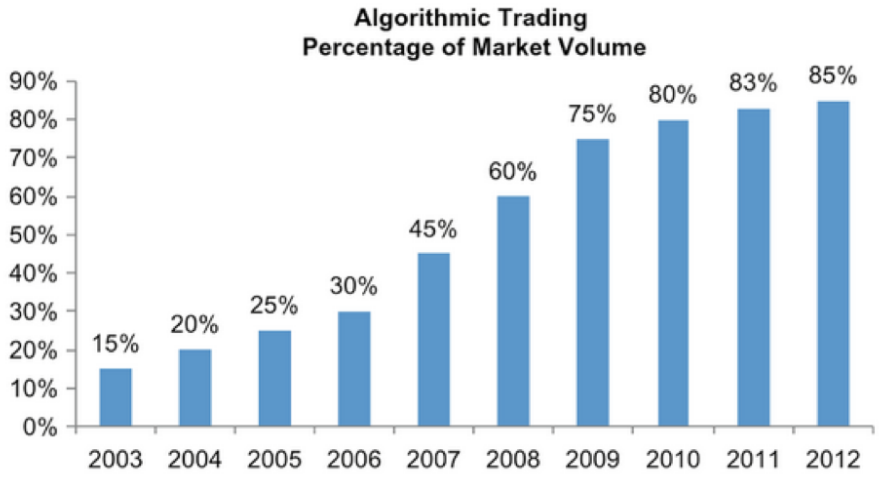
Chapter 1

Introduction

1.1 Context

Algorithmic trading is a method of executing orders using automated pre-programmed trading instructions accounting for variables such as time, price, and volume. This type of trading was developed to make use of the speed, accuracy and data processing advantages that computers have over human traders.

Figure 1.1: Growth of Algorithmic trading as a percentage of market volume. Sourced from [1].



According to data from Wall Street, around 60-73% of the total volume of US equity trading is algorithmic, with a similar percentage in Europe. [3] With most of the trading activity being done by algorithms, it is important for retail investors to have access to this technology.

Uneducated users are often attracted to algorithmic trading due to the promise of low effort money. However, algorithmic trading is a complex field that requires a deep understanding of the financial markets, trading strategies, and the programming of such strategies.

The open source community is replete with an extensive array of

valuable tools for algorithmic trading, such as backtesting frameworks, data analysis libraries, and even end to end trading bots [4]. However, these tools are often difficult to use, and require a deep understanding of the underlying technology. This makes it difficult for uneducated users to take advantage of these tools.

By using an automated trading platform, users can automate their trading activity, without having to develop their own trading bot. This allows them to focus on the trading strategy and the financial markets, without having to worry about the technical aspects of the trading bot. Expert users can also benefit from the platform, as it allows them to quickly test their trading strategies on multiple accounts, and to easily deploy their strategies to multiple exchanges.

Many automated trading platforms are available on the market. However, these platforms are either commercial and closed source, or open source but difficult to use.

Open source automated trading platforms present users with the opportunity to access and modify their source code. One example is *QuantConnect* [5]. While open source platforms like this offer trust and transparency, they are often difficult to use, and require a deep understanding of the underlying technology. This makes it difficult for uneducated users to take advantage of these platforms.

On the other hand, there are commercial and closed source automated trading platforms available in the market. These platforms, although not openly sharing their source code, provide users with a more user-friendly and straightforward experience. An example of such a platform is *Pionex* [6]. While closed source platforms may limit users' ability to customise or modify the underlying algorithms, they offer a more accessible interface and ease of use, which can be advantageous for uneducated users seeking simplicity in their trading endeavours.

The proposed solution intends to bridge the gap between open source and closed source automated trading platforms. The solution will be open source, allowing users to access and modify the source code. However, the solution will also be easy to use, allowing uneducated users to take advantage of the platform. This will be achieved by providing a user-friendly interface, and by abstracting away the technical aspects of the platform and the infrastructure underlying it.

1.2 Motivation

My interest and limited experience in algorithmic trading has led me to explore the available tools and platforms. I have found that the available tools are either difficult to use, or closed source. This has motivated me to create a platform that is both open source and easy to use. Friends and colleagues have also

expressed interest in algorithmic trading, but have been discouraged by either the complexity or the cost of the available tools. This has further motivated me to create a platform that is accessible to uneducated users.

A key motivation is also to facilitate the democratisation of algorithms, enabling uneducated investors to access and benefit from the expertise of expert communities. Traditionally, algorithmic trading has been either limited to individuals with specialised knowledge and technical skills or to those who can afford to pay for the services of such individuals or the companies employing them. Through this platform, the gap between expert and novice investors is bridged, allowing the latter to benefit from the expertise of the former.

Additionally, the open source nature of the project allows communities to operate the platform at no cost other than the cost of the cloud infrastructure, enabling those who don't pursue profit to offer the platform to users at no cost, paving the way for a user-centric environment.

1.3 Objectives

The goal of this project is to design the base for a customisable Free and open-source software automated trading platform that can be used by retail investor communities to implement their trading strategies and offload the technical aspects of running a trading bot. To achieve this goal, the following objectives are defined:

1.3.1 Primary objectives

1. Design a platform that is highly scalable and highly available, to ensure reduced downtime and to support a large number of users and strategies.
2. Design a platform that is easily extensible and customisable, allowing custom trading strategies and bots.

1.3.2 Secondary objectives

1. Research the state of the art in software architectures, to ensure the proposed architecture is up to date with the latest technologies and best practices.
2. Automate the deployment of the platform, to ensure a simple and straightforward deployment process.
3. Set up a sample implementation of the platform, to demonstrate the feasibility and good practices of the proposed architecture.

4. Provide a comprehensive development environment to further develop the platform with minimal cost.

The implementation of any trading bot is out of the scope of this project, as the focus is on the platform itself. Nonetheless, the platform should be easily extensible and customisable. A strategy developed by a user should be able to be deployed in the platform with minimal effort, and the platform should be able to scale to support a large number of users and strategies.

Chapter 2

State of the art

In order to fulfil the objectives of this project, it is necessary to understand the current state of the art in the applicable fields. This chapter will provide an overview of the technologies and tools used in the development of cloud native applications, as well as the infrastructure they run on, in order to provide context for the development of the project and follow the best practices in the field.

> Cloud-native software is defined by how you compute, not about where you compute. Writing software for the cloud demands that we treat change as the rule, rather than the exception. It is this that allows us to produce software that runs more reliably than the infrastructure that it is deployed to. [7]
>
> CORNELIA DAVIS

2.1 Cloud Native Applications

The term *Cloud Native* has no formal definition, however it has been coined by the development community to represent a new model of applications, ones that are built *on* the cloud and *for* the cloud [8]. Gannon et al. state that these applications share some key concepts [9]:

- They operate at global scale
- They must scale well to thousands of users
- The infrastructure is considered fluid and failure is expected.
- Update and testing happen seamlessly without damaging production-
- Security must be built into the services, due to the complexity of the architectures.

The main takeaway is that software must adapt to the current state of infrastructure, but in order to fully understand how *Cloud Native* applications are built, the techniques and tools used must be understood first.

2.1.1 DevOps

The term *DevOps* comes the combination of the traditional Development and Operations teams. It itself represents a series of techniques and practices aimed towards *bridging the gap* between this two main actors in software development. [10]

It could be conceived as an extension of *Agile* [11] software development principles to the complete software delivery pipeline. The main objective of *DevOps* is to deliver applications and services at a higher velocity. Enterprises following this model can evolve and improve software faster, more reliably, enabling them to scale easier.

In order to achieve this goal, several tools are widely used. The most common examples are: containerization, CI, CD, IaaS/PaaS, IaC. [12] All of them will be object of study for the development of this project.

2.1.2 Code Automation

One of DevOps main principles is maximum automation, from the moment the code is written, to the monitoring of the system in production. For the sake of reaching this automation, different tools are used.

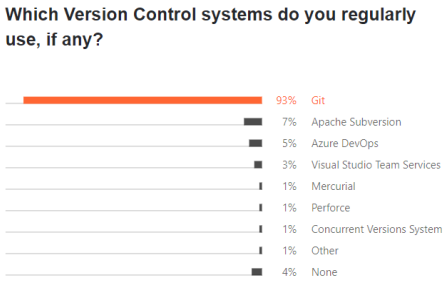
2.1.2.1 Version Control

> A source code control system [is]
> a giant UNDO key—a
> project-wide time machine. [13]
>
> Andy Hunt and Dave Thomas

Centralized VCS play a major role in software development pipelines, not only by keeping *control of versions* but also by simplifying a team-based programming effort. [14] Keeping track of changes and versions of the code in a central repository enables automation on that code.

Multiple open-source VCS tools have been used in the past, however, as we can see in Figure 2.1 there is now a clear preference by developers towards *Git*, this could be a consequence of *GitHub* popularity [15].

Figure 2.1: Question about VCS use in JetBrains Developer Survey 2021. Sourced from [2].



2.1.2.2 Continuous Integration/Delivery

Continuous Integration is all about integrating code changes as quick and safely as possible. Every time changes are made to the code base, automated builds will compile/check and test the code. [16] Frequent builds and testing allow for faster error detection. [17]

Continuous Delivery picks up where CI left off, extending past Unitary testing. Integration testing is performed then software is automatically deployed to a testing environment where Acceptance Testing can take place. When changes to the code succeed all the aforementioned tests, it is automatically deployed to production. [18]

The use of CD in software development pipelines provides shorter time-to-market, improved release reliability and continuous feedback on the code. [19]

2.1.3 Containerised Applications

The growth of the internet led to an increase in the need for small web servers, in order to provide hosting services to small customers. Having great amounts of low-end hardware is neither space or cost-efficient, therefore hosting providers quickly adopted a virtualisation model, allowing them to partition high-end servers into small virtual ones. Virtualised servers provided isolation at the process, network and filesystem levels. [20]

Hardware Virtualisation was present in most systems throughout the 2000s decade. Due to the performance overheads it brought with it, caused by running a full extra networking-filesystem-kernel stack on top of a hypervisor, OS-level virtualisation, commonly known as *containerisation*, emerged.

Container-based virtualisation preserves the isolation provided by traditional VMs while reducing the overhead created by the virtualisation of the OS. [21] The benefits from containerisation are not limited to performance, *containers* allow for increased portability, by packing needed libraries next to

the application, hence decoupling the software from the infrastructure it runs on. [22] Operation teams can deploy software with confidence that it will run faithfully regardless of the OS or hardware they are using.

Technologies such as *Docker* and the *Open Container Initiative* provide tool-sets to build applications into standardised containers and subsequently run them.

2.1.4 Micro-services Architectures

As a consequence of the market demand for dynamic software, the need for fast-paced updates is stronger than ever. Furthermore the vast figures of users accessing services at once requires them to scale flexibly and with ease. Monolithic architectures are no longer suitable, thereby the need for modular, small-sized, independent services manifests itself. [23]

Jaramillo et al. determined that micro-services are built upon the following premises [24]:

- They must be *Focused* on a single domain, be small and have single-responsibility.
- They must be *Loosely coupled*, no two services should need to be updated or deployed together.
- They must be *Language-agnostic* and communicate by exposing a standardised API, commonly a REST API.
- They have *Well-defined boundaries* amongst them, each one should work on a bounded context.

The result is a dynamic, cloud-ready architecture that enables easy horizontal scaling while being easier to develop, maintain and deploy. micro-services are usually developed by different teams, the loose coupling amongst them allows for independent development teams. Each service can be updated and deployed independently, isolating failures to their respective domain and reducing time to market.

Despite its advantages, micro-services architectures pose for a new set of challenges to achieve a successful implementation, some of them being [24]:

- *Failure isolation*, software needs to have greater granularity and lesser responsibilities in order to allow for it to fail in a controlled way.
- *Observability*, it is essential to monitor the status of every single peace of software in the system to detect anomalies.
- *Automation*, given the exponential and dynamic growth in the number of services automation is crucial.

- *High Independence*, services must be kept decoupled in order to allow for independent development.

2.1.5 Container Orchestration

Micro-services architectures can be composed often by thousands of container instances across different bare-metal hosts and regions. In order to maintain this cluster of containers fault tolerant, scalable and available *Container orchestration* platforms are used. These platforms are responsible for managing the state of the cluster, scheduling workloads, providing HA and fault-tolerance, enabling service discovery and simplifying networking amongst other responsibilities. [25]

Several container orchestration platforms are used for building and running enterprise applications, however, among the open source ones, Kubernetes is the industry reference. The reason behind its wide adoption is its abundance of features. Some of them are automated rollouts and rollbacks, storage orchestration, self healing, horizontal scaling, service discovery, load balancing and batch execution. [26]

2.2 Cloud Infrastructure

Aligned with cloud native application development, infrastructure has evolved to keep up with market needs. In the same manner applications abstracted components, infrastructure has abstracted itself from the hardware being used.

2.2.1 Traditional Infrastructure

Traditionally, infrastructure was offered in either opened or closed hardware configurations with a specific model and capacity of RAM, CPU, storage, and networking. Infrastructure providers would then provision this machines with the operating system the client requested. With the right amount of capital, it was not hard to build this infrastructure oneself, therefore most enterprises opted to host their own infrastructure to lower the costs. However, this model brought little flexibility to business, requiring them to deploy new infrastructure in order to scale their services. This process is both costly and laborious.

2.2.2 Infrastructure as Code

In order to set up new machines for expansion, system administrators would install the same stack of applications and the same operating system on many different machines, applying the same security and network policies. This process was done imperatively one machine at a time, developing many different

scripts in order to automate it. While this scripting proved useful for repetitive tasks, they were highly error prone and OS dependent. Moreover, the need for operation teams distanced the development teams from the infrastructure the software ran on.

As a result of DevOps practices, Infrastructure as Code emerged as the primary way to provision and manage infrastructure. Deployments are instead made declaratively using a Domain-specific language (DSL), abstracted from the OS and shell underneath. This provides many advantages, some of them them being [27] :

- All defined operations are done idempotently, allowing for a single configuration that can be changed in order to either mutate the state of a system or deploy a new one.
- Simpler deployments, as developers only need to define the desired state of the machine, leaving the implementation up to the provisioning tool.
- Infrastructure can be treated the same way as code and kept under version control in order to facilitate testing and change ownership.
- Deployments can be automated using CI/CD tools and allowing for system provisioning without requiring any human input.

2.2.3 Infrastructure as a Service

Cloud providers evolved their services by providing a layer of abstraction on top of infrastructure. Instead of the traditional approach of billing for the usage of a specific machine, which often translated to paying more than what's actually used, they provided the option to bill by resource usage.

Systems are offered using a multi tiered model, where differently *specd* and priced VMs are offered dissociated of the specific hardware. This VMs can often be provisioned and up-scaled in a few minutes using an API, expediting both automated up and down-scaling of services. [28] [29]

2.2.4 Platform as a Service

Extending the abstraction provided by IaaS, Platform as a Service (PaaS) abstracts the whole infrastructure. Developers are able to run software without provisioning and managing infrastructure.The cloud provider is responsible for the provisioning of resources, planning of capacity and maintenance of software.

Applications running on a PaaS model can easily scale without any interaction on the developers side. In contrast to IaaS where pricing was based on the number of instances running, in a PaaS model only used resources are billed. [28]

Chapter 3

Project development

3.1 Technology Stack

To fulfil the objectives of this project, a set of technologies must be chosen. These technologies must be up to date with the latest technologies and best practices, and they must be able to achieve the objectives of the project. This section will explain the technologies chosen for the development of the project, and the reasons behind the choice.

As explored in the state of the art, the project should be developed using a cloud native approach. This means that the application should be built using the latest technologies and best practices, and it should be deployed on a cloud infrastructure.

The very nature of cloud native applications is aligned with the objectives of this project. Cloud native applications are built *on* the cloud and *for* the cloud, and they are designed to be highly scalable and highly available. The system can be scaled to support a large number of users and strategies, and it can be deployed on a highly available cloud infrastructure to ensure high availability.

3.1.1 Containerisation

Containerisation is a key technology in a cloud native applications. It provides a standardised way to package and deploy applications, and it allows for the decoupling of the application from the infrastructure it runs on. This allows for the application to be run on any infrastructure that supports the containerisation technology.

The use of containers for trading strategies and bots also empowers users to develop their own strategies and bots, and to deploy them on the platform with minimal effort. Any technology stack they are familiar with can be used to develop the strategy or bot, and it can be packaged in a container and deployed on the platform, as oposed to commercial platforms that often limit users to their proprietary technology stack. This allows for the platform to be easily extensible and customisable, which is one of the objectives of the project.

3.1.2 Kubernetes and Helm

The Kubernetes container orchestration platform is the industry standard for cloud native applications. It provides a rich set of features, such as automated rollouts and rollbacks, storage orchestration, self-healing, horizontal scaling, service discovery, load balancing, and batch execution. Kubernetes is also open source, and it is supported by a large community of developers and companies. It is also cloud agnostic, meaning it can be deployed on any cloud provider, or even on-premises.

These features, along with the abstraction of the infrastructure provided by Kubernetes, make it the ideal choice for the deployment of the application. Kubernetes will be used to deploy the application, and to provide the scalability and availability required by the objectives.

Deploying and managing an application on Kubernetes can be a complex task. To simplify this task, Helm will be used to package the application into a set of charts that can be idempotently deployed on Kubernetes. Helm also provides a package manager that can be used to install and update the application.

The combination of both technologies will allow for a simple and straightforward deployment of the application, while also providing the scalability and availability required by the objectives.

3.1.3 Programming Languages in the Services

The programming languages used for the implementation of the microservices in this project are Java and Kotlin. Both languages are part of the Java Virtual Machine (JVM) ecosystem, which provides a rich set of libraries and tools for software development. The JVM ecosystem is also cross-platform, allowing the software to be run on any operating system that supports the JVM.

The choice of Java and Kotlin is based on personal preference, familiarity, and experience with the languages. The optimal choice of a programming language is highly dependent on each service and its requirements. However, with containerisation, the language used is not as important as it used to be since the software is decoupled from the infrastructure it runs on. Languages and frameworks that are more performant in terms of I/O could be used to improve the performance of the services, but the development time would be higher.

3.1.4 Programming Language in the Web Interface

The programming language used in the web interface is TypeScript. TypeScript is a superset of JavaScript that provides static typing and object-oriented programming features. TypeScript is compiled to JavaScript, which is then run

in the browser.

The use of the Vue framework is advantageous for a single page application, as it provides a simple and straightforward way to develop the application. It provides a rich set of features, such as routing, state management, and component-based development. Using a framework also allows for a more structured and organised codebase, which is advantageous for any software project.

3.2 System Architecture

In line with the motivation and objectives, and according to the state of the art, a platform such as this falls into the category of a Software as a Service platform. To provide a (good) SaaS product from a technical standpoint we need to achieve three key properties:

- Scalability
- Availability
- Security

As explored in the state of the art, these are some of the characteristics of a Cloud Native Application, therefore being a desired model for this application. So as to provide the aforementioned scalability and availability requirements the application will be designed using a micro-service architecture, in which services will be containerised and then orchestrated using the Kubernetes container orchestration system. The use of the latter enables automatic horizontal scaling of different modules of the software, based on system metrics such as CPU and RAM utilisation, taking care of the scalability aspect. It also accomplishes to solve the availability requirement, containers can be distributed amongst different *Availability zones* or even different providers to achieve data-center fault tolerance.

However managing a complex application on Kubernetes can prove to be a challenge, keeping track of versions, dealing with rollouts and rollbacks, even managing simple dependencies. Helm solves this problem by reducing complex applications to simple *Charts* that can be idempotently deployed. Acting as a package manager it also enables simple dependency updates [30].

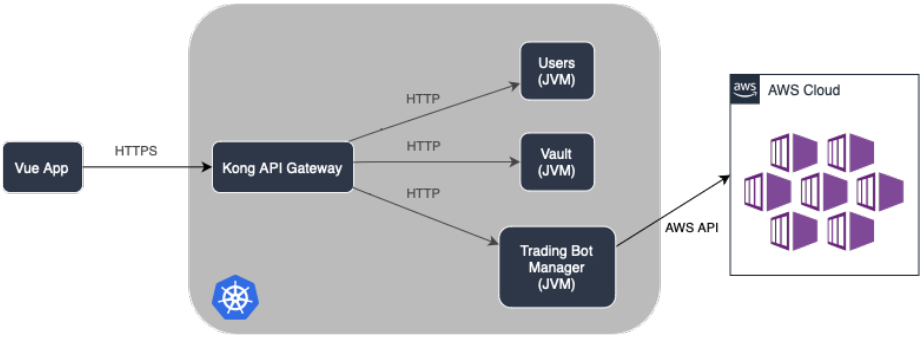
3.2.1 Architectural Design

Within the Kubernetes name-space, each service will provide their functionality through a REST API accessible only inside the cluster. To provide external access an API Gateway will act as a reverse proxy. The use of the API Gateway Pattern solves many of the complications introduced when using

a micro-services architecture like routing, traffic management and caching amongst others [31][32].

The Gateway is responsible for routing, load balancing, and serving as a single entry point for the client. It enables the decoupling of services from the client, simplifying the client's implementation. Authentication is enforced by the Gateway through signature verification of the client's JWT, ensuring authorized access to the services [33]. Additionally, the Gateway handles rate limiting and throttling to prevent system abuse. Moreover, the Gateway manages the termination of TLS, eliminating the need for services to have TLS certificates and offloading the computational cost associated with TLS to the Gateway.

Figure 3.1: System Architecture Diagram



A proper micro-service architecture is segregated by characteristics and composed by simple components [34]. The base structure for this application is the following:

- Users
- Vault
- Trading Bot Manager

The *Users* service is responsible for the authentication of clients. Login and register operations, as well as user data updates are the sole responsibility of this service. Users will be issued a JWT signed by an authoritative certificate that can be used for authentication against any service.

The *Vault* service holds symmetrically encrypted API keys to the trading brokers the users provide. This data is only decrypted with the user password, unknown to the system, and sent to be used by the trading bot.

The *Trading Bot Manager* is responsible for the provisioning and monitoring of bot instances in a separate public cloud resource. Independent cloud resources guarantee a separation between client instances and prevents resource contention.

On top of these services a web application, independent and agnostic to the services inner workings, communicates with the exposed REST API. The deployment and infrastructure of such UI is also discrete from the services.

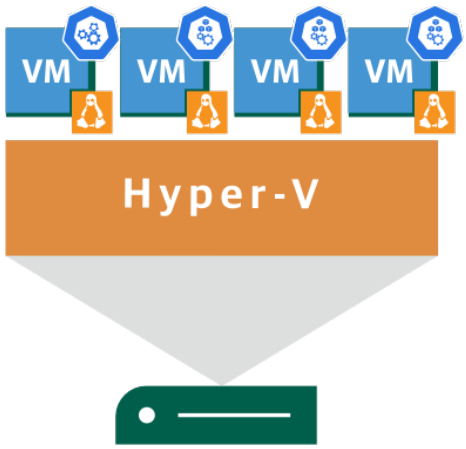
The layer separation between the business logic and presentation is advantageous, in case of a service malfunction the damage is contained to the specific service, as previously described in Gong et al. [35]

3.2.2 Development Environment

Considering the cost that would be incurred with the use of a public cloud provider for the development and testing of this project, a local development solution is imperative.

As a simple solution we can virtualise a Kubernetes cluster in our machine.

Figure 3.2: Development Environment diagram



This is done in three layers:

First off, to simulate a bare-metal Debian cluster Vagrant is used. Vagrant is an open source tool for provisioning development environments in a consistent and swift manner [36]. Using a simple scripting language a configurable number of Debian 11 node VMs are provisioned and started on an Hyper-V backend. The result is a cluster of machines, a *master* node and multiple *worker* nodes, configured with mDNS hostnames to facilitate access from the next *layer*. Due to some limitations in the networking configuration in Vagrant this script is tightly coupled to Hyper-V networking APIs, and as a result it can only be used in a Windows machine with the Hyper-V engine enabled.

On top of the previous *layer* or against any other cluster of physical or virtual machines running Debian 11, we use Ansible, an open source automation

tool for system configuration, software deployment and many other tasks using a human-readable language [37]. All of this is done requiring nothing but SSH for transport on the target machines, making it a *drop-in* solution for any infrastructure configuration. The execution of this tool results in a fully functional Kubernetes cluster with the master and slave nodes synchronised and a copy of the credentials file. If both of this *layers* are used, the resulting infrastructure would be the one represented in 3.2

Once a Kubernetes cluster is up and running, the third *layer* would be Helm. Helm *charts* can be deployed against the cluster to install and start the application. Any Kubernetes cluster either a local development one or a PaaS managed service can be used with Helm.

These three layers are independent and replaceable, but together provide a fully functioning development environment and could easily be adapted to be used in *production*. The specific instructions on how to deploy the application are available in the main knag repository.

To enable the application to be run on Kubernetes each of the services that compose it must be packaged in a OCI container image. In this project the Docker tooling will be used to generate said images. This images can be tested independently as well as together to guarantee the working of the system.

3.2.3 Deployment

Any cloud provider could be used for the Deployment of the platform.

Due to the cost incurred in the hosting and domain registration, the platform will only be hosted during the defence of this thesis.

The domain structure is as follows:

- knag.software, the root domain, serves the web interface.
- api.knag.software serves as the entry point for the API Gateway and the cluster.
- helm.knag.software is the helm repo for the application charts

A Helm chart is provided in the main repository [38]. The chart is composed of the following services:

- Users
- Vault
- Kong (api gateway)

As well as the corresponding persistence dependencies for each service.

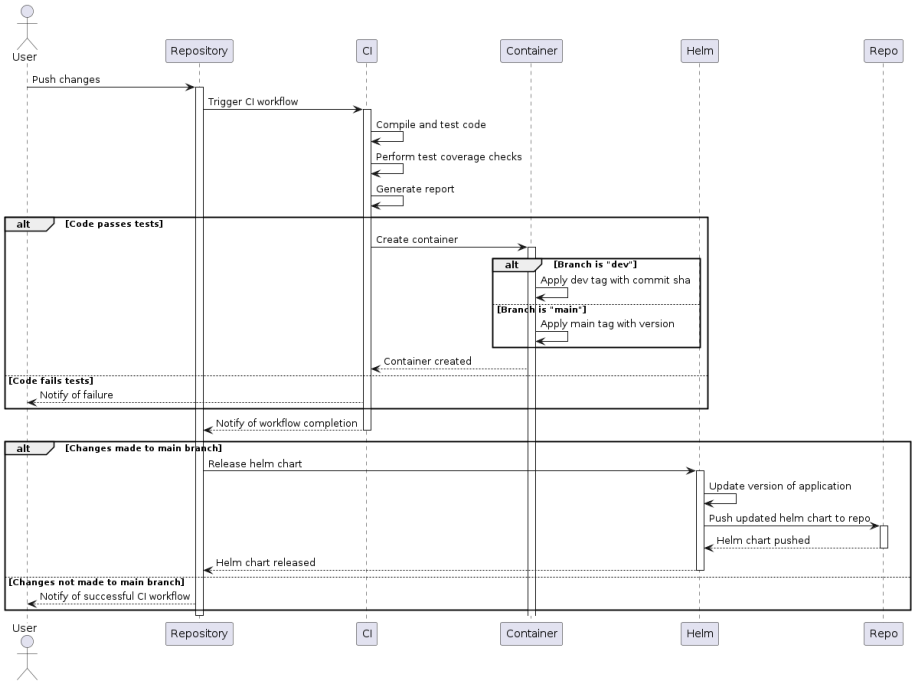
3.2.3.1 Automation on GitHub

The source code for all services is available on GitHub. The main chart with the documentation to deploy it is available on a repository, with every service available on a separate repository.

As a simple workaround for helm charts hosting, another repository is used and accesible through the helm.knag.software subdomain.

Each service repository includes an automated GitHub Actions CI workflow that performs several tasks. Firstly, it compiles and tests the code, followed by generating a container image and publishing it to GitHub's container registry. For changes made on the development branch, a tag is created using the SHA signature of the commit. On the other hand, if the commit is made against the main branch, an image with the specified service application version is published to the registry. Additionally, an updated helm chart is pushed to the chart repository mentioned earlier.

Figure 3.3: GitHub Actions Workflows diagram



The interface is hosted on GitHub pages under the same repository its source code resides. Another GitHub Actions workflow builds and deploys changes to the source code automatically.

3.2.3.2 Cloud Provider

To provide extra resilience and availability the application can be deployed using a mixture of cloud providers. The services can be deployed in different providers and regions to achieve fault tolerance. Persistence layers can be deployed on-premises or in a different cloud provider to achieve data locality and compliance with data protection laws. The web interface can be deployed in a content delivery network to provide a better user experience.

Overall the architecture is provider agnostic, and highly adaptable to the needs of the client. Hybrid cloud deployments are also possible, with the services deployed in a private cloud and the web interface deployed in a public cloud provider. A mixture of PaaS and IaaS services can also be leveraged to reduce the operational costs of the application.

As an example, the services can be deployed in a Kubernetes cluster in Azure, the persistence layers can be deployed in a private cloud and the web interface can be deployed in a content delivery network. At the same time the Bot instances can be run in a separate private cloud or Container PaaS service, such as Google Cloud Run.

This would provide a highly available and resilient application. In the extreme case of a complete provider or region outage, some services could still be available.

3.3 Implementation

The chosen API gateway for the micro-services is the Kong Ingress Controller [39], due to its cloud native design, Kubernetes first approach and its open source nature. It is deployed in a Kubernetes cluster and exposes the services through a single entry point. It can be configured using Custom Resources in Kubernetes, which allows for a simple and declarative configuration of the gateway.

All the services are implemented in JVM languages for the following reasons:

- Platform independence
- Security
- Correctness
- Memory managementmed

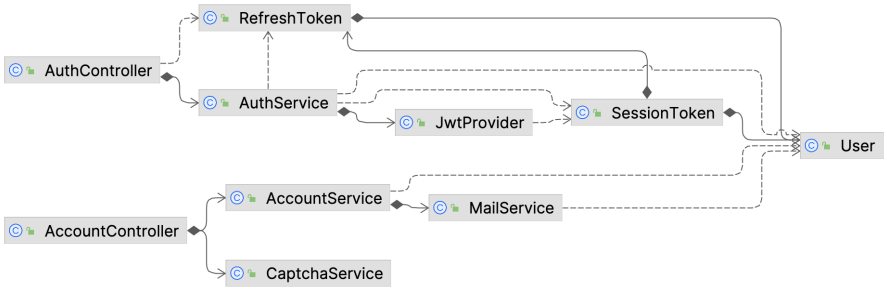
Having a similar stack in a micro-services architecture is not a necessity, since applications are packaged separately each of them can enjoy a completely different stack and even a different API.

For the development of these services the InteliJ Idea IDE was used, due to the suite of efficiency-enhancing features such as intelligent coding assistance, reliable refactorings, instant code navigation, built-in developer tools, web and enterprise development support, and more. [40]

3.3.1 Users Service

The users service is implemented using Spring Boot [41], a web framework for Java. It serves as a sample for the rest of the services, since it is the most complete one. It provides a REST API for the management of users and authentication.

Figure 3.4: Users Service Simplified Class Diagram



The code follows a clear separation of concerns between the service, controller, and repository layers. This promotes modularity, code maintainability, and facilitates collaboration among developers. The service layer handles the business logic, the controller layer handles request handling, and the repository layer manages data persistence. This separation enhances code organization, promotes code reuse, enables independent unit testing, and supports scalability and extensibility.

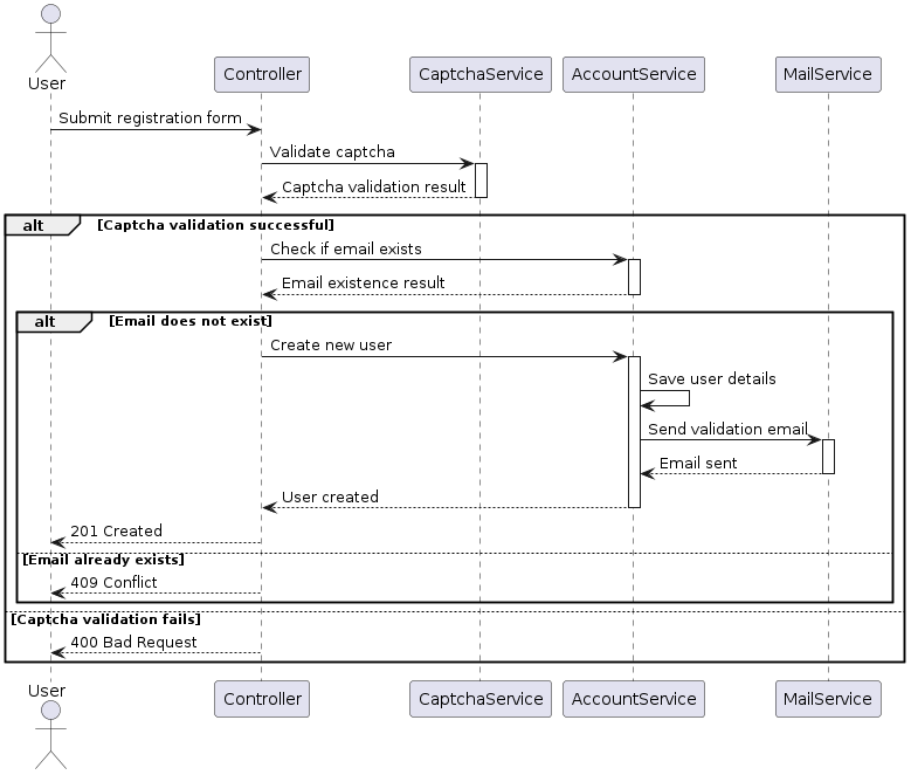
The persistence layer is implemented using Spring Data JPA, which provides an abstraction layer over the database, in this case MongoDB. This allows for a simple and clean implementation of the data access layer, without the need of writing boilerplate code.

The configuration of the service is done through environment variables, which are injected into the container at runtime. This is done to avoid the need of a configuration file and to facilitate the deployment of the service in a container orchestration system. Amongst the configuration variables we can find the database connection details, the certificate and key to sign the JWT tokens and the API key for the email service.

Users can register to the service after successfully completing a reCAPTCHA [42] challenge. This is done to prevent bots from registering to

the service and to prevent the service from being used to send spam emails. Afterwards a third party email service is used to send a verification email to the user, which contains a link and a token to verify the user's email address.

Figure 3.5: Sequence Diagram of a User registration



Passwords are hashed using the bcrypt algorithm, described by Provos et Mazières [43]. The algorithm stores salted hashes of the passwords along with the salt used to generate them. This allows for a simple and secure implementation of password hashing.

The service is documented using OpenApi [44], and a web interface is provided to explore the API and test it using the Swagger UI [45].

The application is packaged in the form of a container image using the Docker tooling. The image is then published to the GitHub container registry for consumption by the container orchestration system.

In the same repository a Helm chart is provided to deploy the service in a Kubernetes cluster. The chart includes a MongoDB instance as a dependency, which is also deployed in the cluster. This instance can be replaced by another one, either in the same cluster or in a different one, by changing the chart values.

3.3.2 Vault Service

The Vault service is responsible for the storage of the user's API keys. It is also implemented using the Spring Boot framework and provides a REST API for the management of the keys.

A MariaDB database is used to store the keys, which are encrypted using the user's password as a key. This is done to prevent the service from being able to decrypt the keys, since it does not have access to the user's password. The encryption is done using the AES-256-CBC algorithm, which is a symmetric encryption algorithm [46]. The key is derived from the user's password using the PBKDF2 algorithm with using HMACSHA256 as the hashing algorithm [47][48].

To verify ownership of the API keys, the user must provide the password used to encrypt them. This password is then used to decrypt the keys and send them to the user. The keys are never stored in plain text, they are only decrypted in memory and sent to the user.

As brute-forcing prevention the service requieres a valid JWT token to be sent along with the request, which is issued by the users service and verified using the certificate used by the users service to sign the tokens. The user id contained in the token is also used to verify that the user is the owner of the keys.

3.3.3 Trading Bot Manager Service

This service is responsible for the provisioning and monitoring of bot instances in a separate public cloud resource.

Due to the nature of the application, the bot instances must be able to run arbitrary code, therefore a containerised solution is the best fit. To achieve this the service would use a public cloud provider's API to create and manage containers in the cloud, such as Azure Container Instances or AWS Fargate. Instances would be created with a container image that contains the bot code and a API would be exposed to manage the instances. The bot code would be provided by the user and uploaded to a container registry, for example the Docker container registry, and then referenced in the instance creation request. The API key for the broker would be provided by the user and stored in the vault service, the bot code would not be responsible for retrieving it from the vault and using it to authenticate against the broker, this would be done by the bot manager service. Other users could instantiate the same bot code, but they would not have access to the API key, therefore they would not be able to authenticate against the broker, to do so they would need to provide their own API key.

Due to the time constraints of this project, as well as the provider lock-in that would be incurred, this service is not implemented, but the

architecture is described for future work.

3.3.4 Web User Interface

A web interface is provided to interact with the services. It is implemented using Vue as a Single Page Application. Vue is a progressive framework for building reactive user interfaces [49]. The data is fetched from the services through the API Gateway and displayed to the user. The user can then interact with the data and send requests to the services through the API Gateway, without the need to know the inner workings of the services.

For development the TypeScript language is used, which is a typed superset of JavaScript that compiles to plain JavaScript [50]. This allows for a more robust codebase and better tooling support.

The code is organised in components, which are reusable pieces of code that can be used to build the user interface. The components are then composed to build the user interface. This allows for a modular and reusable codebase. Each of these components is reactive, meaning that they update automatically when the data they display changes without the need to re-render the whole page.

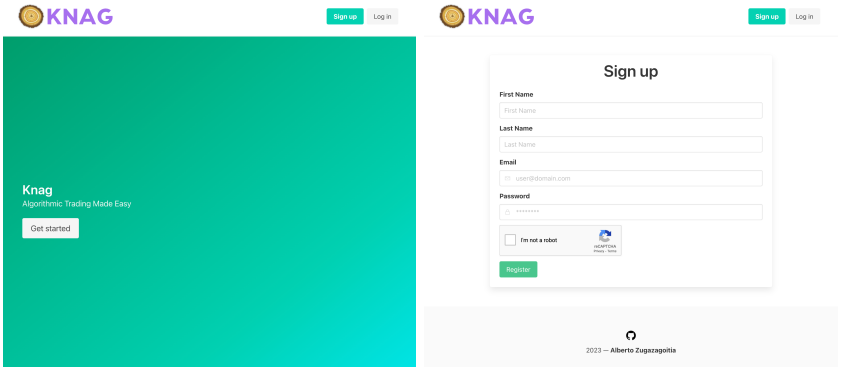
After each commits to the main branch of the repository, a GitHub Actions workflow is triggered to build and deploy the changes to GitHub Pages. This allows for a simple and automated deployment of the application.

Chapter 4

Results

4.1 Resulting Product

The resulting product is the structure of a platform that can be used to provide a SaaS product. The platform is composed of three services, a web interface and a container orchestration system. The services are deployed in a Kubernetes cluster and exposed through an API Gateway. The web interface is deployed in a content delivery network.



(a) Landing Page (b) Register Page

Figure 4.1: Screenshots of the resulting webpage

The reference implementation of the user service is provided, both as a microservice and in the web interface, as can be seen in figure 4.1. The register, login and email verification functions are implemented. The vault service is also implemented, but it is not used by the web interface. The trading bot manager service is not implemented, but the architecture is described.

For the deployment of the services in a Kubernetes cluster, Helm charts are available to streamline the process. These Helm charts are specifically designed to simplify the installation and management of the services within the cluster environment.

To facilitate easy access and distribution of the Helm charts, they are published in a dedicated Helm repository. This repository can be conveniently accessed through the subdomain helm.knag.software. By accessing this repository, users can conveniently browse and retrieve the necessary Helm charts to deploy the services.

To ensure consistency and reliability, each service independently publishes its own container image to the GitHub container registry. These container images act as the building blocks for the services and encapsulate their respective functionalities. The Helm charts then utilize these container images as the foundation for deploying the services in the Kubernetes cluster.

To simplify and automate the process of publishing both the container images and the corresponding Helm charts, a GitHub Actions workflow is provided. This workflow automates the publishing of the container images to the GitHub container registry, ensuring that the latest versions are readily available for deployment. Additionally, it also automates the publishing of the Helm charts to the Helm repository, enabling seamless access and deployment of the services.

In addition to the individual service Helm charts, an application chart is also provided. This comprehensive chart facilitates the deployment of all the services as a cohesive unit within the Kubernetes cluster. By utilizing the application chart, users can easily set up and manage the entire suite of services in a streamlined manner.

The complete source code is available both as an annex to this project as well as on-line at the following repositories:

- `https://github.com/zugazagoitia/knag`
- `https://github.com/zugazagoitia/knag-users`
- `https://github.com/zugazagoitia/knag-vault`
- `https://github.com/zugazagoitia/knag-web`

4.2 Achieved Objectives

4.2.1 Primary Objective 1: Design a platform that is highly scalable and highly available

Through the use of a cloud-native microservices architecture with a Kubernetes cluster, the platform is highly available [51]. This means that the platform remains accessible and operational even during peak usage periods.

The microservices architecture allows for the modularization of different components and functionalities of the platform, enabling flexibility, scalability, and fault isolation. Each microservice operates independently, ensuring that a failure in one component does not impact the entire system [51]. This architecture also enables efficient resource utilization, as services can be dynamically scaled up or down based on demand, optimizing performance and responsiveness [52].

By deploying the platform on a Kubernetes cluster, we can take advantage of its inherent features such as automatic scaling, load balancing, and self-healing capabilities. The cluster ensures that the platform can handle increased traffic and requests without compromising availability. In the event of a failure or disruption, Kubernetes can automatically redistribute workloads to healthy nodes, minimizing downtime and ensuring a seamless user experience.

Moreover, the use of a cloud-native approach allows for easy deployment and management of the platform across multiple cloud providers, ensuring geographic distribution and redundancy. This further enhances availability, as users can access the platform from different locations, and any localized issues or outages can be mitigated through failover mechanisms.

4.2.2 Primary Objective 2: Design a platform that is easily extensible and customizable

By leveraging docker containers for each bot/strategy, the platform is easily extensible and customizable. Users can easily create their own bots and strategies, and deploy them to the platform with minimal effort. A bot can be created by simply creating a docker image with the bot's code, configuring the bot's parameters, and deploying the bot to the platform. A container approach also ensures that each bot is isolated from other bots, ensuring fault isolation and security. Furthermore, the use of docker containers allows any library, Operating System and programming language to be used, since the container is self-contained and independent of the host system.

Any user that wanted to execute a trading strategy available on the platform would be able to do so with minimal effort. The user would simply have to select the desired strategy, configure the strategy's parameters, and deploy the strategy to the platform. The platform would then automatically deploy and run the strategy, and the user would be able to monitor the strategy's performance and results.

4.2.3 Secondary Objectives

4.2.3.1 Research the state of the art in software architectures

The exploration of the state of the art encompasses the latest trends and technologies in cloud-native applications, DevOps, automation, containerization, and microservices. It delves into concepts such as CI/CD, version control systems, Docker containerization, microservices architectures, and Kubernetes orchestration.

This exploration focuses on best practices, industry standards, and emerging methodologies to enhance scalability, flexibility, and resilience in the proposed automated trading platform. Automation in software

development processes is emphasized, streamlining tasks and reducing errors. Infrastructure-as-code and configuration management tools are discussed for efficient cloud resource provisioning and management.

Containerization and microservices architectures are highlighted for their benefits in application packaging, isolation, and portability. The advantages of Kubernetes as a container orchestration platform for automating container management, scaling, and resilience are also explored, contributing to the fullfillment of the primary objectives.

4.2.3.2 Automation of the deployment and sample implementation of the platform

Through the use of GitHub actions the deployment of the individual components of the platform is automated. This allows for a leaner and more efficient development process, as the deployment of the artifacts can be triggered by a simple git push. This also ensures that the deployment process is consistent and reproducible, as the same process is used for every deployment.

Using helm charts, the deployment of the platform is also automated. Each component of the platform is deployed as a helm chart, and the platform as a whole is defined as a helm chart that deploys the individual components. This allows for a simple and straightforward deployment process, as the user only has to run a single command to deploy the entire platform. The configuration of every service is located in a single file, which can be easily modified to suit the user's needs. Helm also allows for the creation of multiple environments, such as development, staging, and production, which can be used to test the platform in different scenarios, as well as enabling easy rollback in case of failure.

A sample implementation of a few services is also provided, to demonstrate the feasibility and good practices of the proposed architecture. The *users* service can be used as a reference for further development. A propper containerization and deployment process is defined, as well as a CI/CD pipeline. The service also provides a versioned Helm chart for easy deployment to a Kubernetes cluster. A OpenAPI schema is also provided, to ensure a consistent and well defined API contract, to be used by other services or clients.

4.2.3.3 Provide a comprehensive development environment

The propposed multi-layered development environment provides a fully functional Kubernetes cluster with the master and slave nodes synchronised, requiring minimal effort. The use of Vagrant and Ansible allows for a simple and straightforward deployment process, as the user only has to run a single command to deploy the entire cluster. The configuration of every service is located in a single file, which can be easily modified to suit the user's needs. The resulting infrastructure is also easily scalable, as the user can simply modify the configuration file to add or remove nodes from the cluster as needed.

Chapter 5

Conclusions

5.1 Conclusions

In conclusion, the development and implementation of the system architecture have yielded significant achievements in terms of scalability, availability, and security. The utilisation of open-source solutions played a crucial role in enabling the successful realisation of this project.

The adoption of a micro-service architecture empowered the system to scale effortlessly, allowing it to handle increasing loads and user demands. The use of Kubernetes for container orchestration provided automated horizontal scaling, fault tolerance, and optimal resource utilisation. These open-source solutions offered a robust foundation for building a scalable system.

Utilisation of open-source solutions not only contributed to the success of this project but also provided cost-effective and reliable tools that can be leveraged by other developers. The collaborative nature of open-source software fosters innovation and encourages the sharing of knowledge and resources, enabling developers to build upon existing solutions and create sophisticated systems.

The achieved results validate the effectiveness of the implemented system architecture, showcasing its scalability, availability, and security. The adoption of open-source solutions has played a pivotal role in enabling the development of this project, underscoring the value and impact of the open-source community in driving technological advancements.

5.2 Social Impact

The project's implementation has the potential to support the achievement of Sustainable Development Goal 8: Promote sustained, inclusive, and sustainable economic growth, full and productive employment, and decent work for all. The system offers benefits that can be leveraged by retail parties to enable investing and contribute to economic development in the following ways:

- Enhanced Financial Access: By providing an accessible and user-friendly platform, the system empowers retail investors to access investment opportunities that were previously limited to institutional investors.

This increased financial access promotes inclusive growth by enabling a broader population to participate in the investment ecosystem.

- Empowering Retail Investors: The system equips retail investors with the tools and resources they need to make informed investment decisions. By providing a base to invest with, the system democratises investment knowledge, enabling individuals to navigate financial markets more confidently and contribute to their economic well-being.
- Job Creation: The growth and adoption of the system can have positive implications for the job market. As retail investors actively participate in investing activities, there is an increased demand for professionals in related sectors such as financial advisory services, investment analysis, and technology development. This leads to job creation, supporting the goal of full and productive employment.
- Economic Stimulus: Retail investors play a vital role in stimulating economic growth by providing capital to businesses, especially small and medium-sized enterprises. The system's utilisation by retail parties can lead to increased investment in businesses, enabling them to expand operations, create job opportunities, and contribute to overall economic development.

5.3 Future Lines

As the system evolves, there are several exciting possibilities for future development and expansion to further enhance the platform's functionality and user experience. These future lines include:

- Introduction of Helper Services: To provide a comprehensive investing experience, the platform can incorporate additional helper services such as a billing service, enabling seamless financial transactions, and a statistics service that offers valuable insights and analytics for informed investment decision-making.
- Focus on Specific Algorithms and Customization: In order to cater to diverse investment strategies and objectives, the system can place a greater emphasis on specific algorithms tailored to different asset classes or investment styles. This customization will empower users to align the platform more closely with their individual preferences and risk appetites.
- Integration of ML-Based Algorithms: Machine learning (ML) techniques hold significant potential in the field of investment. By incorporating ML algorithms, the platform can leverage advanced data analysis and predictive modeling to identify patterns, trends, and investment opportunities that may not be readily apparent through traditional approaches.

- Expansion of Interfaces: Recognizing the growing importance of accessibility and convenience, the system can expand its range of interfaces. This can include the development of accessible interfaces to accommodate users with disabilities and the creation of mobile applications to enable on-the-go investing, providing flexibility and convenience to retail investors.

These future directions hold the potential to further empower retail investors and enable them to make informed investment decisions, while also enhancing the overall functionality and accessibility of the platform.

Bibliography

[1] M. Glantz and R. Kissell, "Chapter 8 - algorithmic trading risk," in *Multi-Asset Risk Modeling*, M. Glantz and R. Kissell, Eds. San Diego: Academic Press, 2014, p. 258. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124016903000081

[2] "Team Tools - The State of Developer Ecosystem in 2020 Infographic — jetbrains.com," https://www.jetbrains.com/lp/devecosystem-2021/team-tools/#Which-Version-Control-Services-do-you-regularly-use-if-any, [Accessed 19-Jun-2023].

[3] "Algorithmic Trading Market Size & Share Analysis - Industry Research Report - Growth Trends — mordorintelligence.com," https://www.mordorintelligence.com/industry-reports/algorithmic-trading-market, [Accessed 12-Jun-2023].

[4] "GitHub - wilsonfreitas/awesome-quant: A curated list of insanely awesome libraries, packages and resources for Quants (Quantitative Finance) — github.com," https://github.com/wilsonfreitas/awesome-quant, [Accessed 19-Jun-2023].

[5] "Design and trade algorithmic trading strategies in a web browser, with free financial data, cloud backtesting and capital - QuantConnect.com — quantconnect.com," https://www.quantconnect.com/, [Accessed 19-Jun-2023].

[6] "Pionex — Bitcoin Ethereum Auto buy low and sell high — Free Crypto Trading Bot — pionex.com," https://www.pionex.com/en/, [Accessed 19-Jun-2023].

[7] C. Davis, "Realizing software reliability in the face of infrastructure instability," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 34–40, 2017.

[8] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217300018

[9] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.

[10] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "What is devops? a systematic mapping study on definitions and practices," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2962695.2962707

[11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," Retrieved May 29, 2007, from http://www.agilemanifesto.org/, 2001.

[12] G. Bou Ghantous and A. Gill, "Devops: Concepts, practices, tools, benefits and challenges," *PACIS2017*, 2017.

[13] H. Andrew and T. David, "The pragmatic programmer," 1999.

[14] D. Spinellis, "Version control systems," *IEEE Software*, vol. 22, no. 5, pp. 108–109, Sep. 2005.

[15] P. Govekar and S. Budhkar, "Review on version control with git," 2018.

[16] M. Fowler and M. Foemmel, "Continuous integration," 2006.

[17] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, May 2014.

[18] V. Pulkkinen, "Continuous deployment of software," in *Proc. of the Seminar*, vol. 58312107, 2013, pp. 46–52.

[19] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002812

[20] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, April 2005.

[21] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*, March 2014, pp. 610–614.

[22] M. H. Syed and E. B. Fernandez, "The software container pattern," in *Proceedings of the 22nd Conference on Pattern Languages of Programs*, 2015, pp. 1–7.

[23] T. Salah, M. Jamal Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec 2016, pp. 318–325.

[24] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*, March 2016, pp. 1–5.

[25] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, Sep. 2017.

[26] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[27] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "Devops: Introducing infrastructure-as-code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 497–498.

[28] D. Rani and R. K. Ranjan, "A comparative study of saas, paas and iaas in cloud computing," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 4, no. 6, 2014.

[29] N. Serrano, G. Gallardo, and J. Hernantes, "Infrastructure as a service and cloud technologies," *IEEE Software*, vol. 32, no. 2, pp. 30–36, 2015.

[30] "Helm — helm.sh," https://helm.sh, [Accessed 14-Jun-2023].

[31] M. Song, C. Zhang, and E. Haihong, "An auto scaling system for api gateway based on kubernetes," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, 2018, pp. 109–112.

[32] A. Akbulut and H. G. Perros, "Software versioning with microservices through the api gateway design pattern," in *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, 2019, pp. 289–292.

[33] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7519

[34] "Microservices Pattern: Microservice Architecture pattern — microservices.io," https://microservices.io/patterns/microservices.html, [Accessed 12-Jun-2023].

[35] Y. Gong, F. Gu, K. Chen, and F. Wang, "The architecture of micro-services and the separation of frond-end and back-end applied in a campus information system," in *2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications(AEECA)*, 2020, pp. 321–324.

[36] HashiCorp, "Provision development environments," Nov 2022. [Online]. Available: https://www.vagrantup.com/use-cases/provision-development-environments

[37] RedHat, "How it works," https://www.ansible.com/overview/how-ansible-works, (Accessed on 04/11/2023).

[38] A. Zugazagoitia, "Knag." [Online]. Available: https://github.com/zugazagoitia/knag

[39] "GitHub - Kong/kubernetes-ingress-controller: :gorilla: Kong for Kubernetes: The official Ingress Controller for Kubernetes. — github.com," https://github.com/Kong/kubernetes-ingress-controller, [Accessed 13-Jun-2023].

[40] "Features - IntelliJ IDEA — jetbrains.com," https://www.jetbrains.com/idea/features/, [Accessed 12-Jun-2023].

[41] "Spring Boot — spring.io," https://spring.io/projects/spring-boot, [Accessed 14-Jun-2023].

[42] "reCAPTCHA — google.com," https://www.google.com/recaptcha/about/, [Accessed 14-Jun-2023].

[43] N. Provos and D. Mazières, "A Future-Adaptable password scheme," in *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. Monterey, CA: USENIX Association, Jun. 1999. [Online]. Available: https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme

[44] "Home - OpenAPI Initiative — openapis.org," https://www.openapis.org, [Accessed 14-Jun-2023].

[45] "REST API Documentation Tool — Swagger UI — swagger.io," https://swagger.io/tools/swagger-ui/, [Accessed 14-Jun-2023].

[46] M. J. Dworkin, *Advanced Encryption Standard (AES)*, 2001. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.197-upd1

[47] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898, Sep. 2000. [Online]. Available: https://www.rfc-editor.org/info/rfc2898

[48] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–15.

[49] "Vue.js - The Progressive JavaScript Framework — Vue.js — vuejs.org," https://vuejs.org, [Accessed 13-Jun-2023].

[50] "JavaScript With Syntax For Types. — typescriptlang.org," https://www.typescriptlang.org, [Accessed 13-Jun-2023].

[51] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019, pp. 176–185.

[52] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 2017, pp. 847–852.

[53] S. M. Mohammad, "Continuous integration and automation," *International Journal of Creative Research Thoughts (IJCRT), ISSN*, pp. 2320–2882, 2016.

[54] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302507