INDUSTRIAL TRAINING REPORT I / II

TRAINING ORGANIZATION : **GTN Technologies (Pvt) Limited**

PERIOD OF TRAINING : **FROM: 31/07/2023**

TO : 15/12/2023

FIELD OF SPECIALIZATION :

COMPUTER ENGINEERING

VILAKSHAN V.

E/18/373

ACKNOWLEDGEMENTS

I extend my deepest gratitude to the Computer Engineering Department for their pivotal role in organizing the intern career fair, which served as a gateway to invaluable industry connections. Special appreciation is directed towards Dr. Asitha Bandaranayake, Senior Lecturer and Mrs. Nadeesha Adikari, Lecturer whose dedication to preparing us for interviews and inviting esteemed companies to the career fair was instrumental in bridging the gap between academic learning and professional application.

My sincere thanks are also owed to the National Apprentice and Industrial Training Authority (NAITA), along with the Industrial Training and Career Guidance Unit of the Faculty of Engineering at the University of Peradeniya. Their comprehensive sessions on industry preparation and adherence to professional guidelines have been a cornerstone of my practical education.

Lastly, I am profoundly thankful to GTN, a leading fintech company, for providing me with the opportunity to undertake my internship. I am particularly indebted to my former tech lead Mr. Chaminda Hettigoda , Head of Engineering Transaction Technologies, and Mr. Jaminda Batuwangala , Head of Fintech and API Integration, for their guidance and support. A special note of thanks to Mr. Mahesh Pathira, Senior Software Engineer, who not only supervised my work but also offered invaluable mentorship that shaped my career trajectory.

This acknowledgment encapsulates the collective efforts of all those who have contributed to my professional growth during my industrial training. Their support and guidance have been a beacon of inspiration and have significantly enriched my learning experience.

Vilakshan V

E/18/373

Faculty of Engineering

University of Peradeniya

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATION

API	Application Programming Interface
fintech	Financial Technology
UAT	User Acceptance Testing
ETF	Exchange-Traded Fund
FX	Foreign Exchange

Chapter 1

INTRODUCTION

1.1 Training session

During my industrial training at GTN, located at Level 10, Parkland, No 33, Park Street, Colombo 02, Sri Lanka, from July 31, 2023, to December 15, 2023, I had the opportunity to engage with the company's main product, the Fintech API. In addition to my regular duties, I participated in a group AI competition hosted by GTN, where our team was honored with the second runner-up position, reflecting the collaborative skills and competitive spirit fostered during my internship.

1.2 INTRODUCTION TO TRAINING ORGANIZATION

Figure 1.1: GTN logo

GTN is a leading fintech company with a long history of success. It holds important licenses in many places around the world through its related companies. Our goal is to help brokers, banks, asset managers, and fintech companies by providing them with powerful and creative solutions for investing and trading.

GTN is made up of a diverse group of more than 300 skilled people working in places like Dubai, Singapore, South Africa, Sri Lanka, the UK, and the US. We all share the same goal: to make it easier for our clients to invest and trade. Our growth is supported by major investors such as the IFC, part of the World Bank Group, and SBI Ventures Singapore, a company related to SBI Holdings, one of the biggest financial groups traded on the Tokyo Stock Exchange. GTN is dedicated to changing how easy it is to access investment and trading opportunities, leading the way in financial technology.

1.2.1 History of the company

To be written

1.2.2 Vision

"We see a world where anyone can invest in anything, anytime".

1.2.3 Mission

"Empowering our partners to drive growth and unlock value for their end clients with superior trading and investment solutions".

1.2.3 Services Provided by GTN

API Product: GTN's API framework is a secure gateway that enables clients to seamlessly integrate their front-end systems with GTN's robust infrastructure. This allows clients to offer their customers trading and investment opportunities in 87 markets across multiple asset classes, while retaining control over their own applications and user experience. The framework supports a single-frame connection for global investments in 8 asset classes and provides the flexibility to set up numbered sub-accounts for enhanced client management.

Figure 1.2: GTN TRADE logo

GTN Trade: The GTN Trade platform is a comprehensive solution that seamlessly integrates with existing back-end systems and APIs, simplifying the process of entering global trading markets. It offers a unified account that manages trading, clearing, settlement, and custody, all within a single interface. With access to 87 markets, including the US and various frontier and emerging markets, GTN Trade leverages the company's extensive experience in crafting trading solutions to provide a streamlined experience.

Figure 1.3: GTN INVEST logo

GTN Invest: GTN Invest is a platform that has modernized the way investment and portfolio management services are delivered. By automating and digitizing these processes, GTN Invest harnesses the power of AI and robotic analytics to maximize returns. It provides access to a diverse range of financial instruments, including cash equities, ETFs, treasury bills, mutual funds, FX, and bonds, across 87 markets. The platform's front-end functionality is both customizable and scalable, enabling the swift launch of new features to meet evolving market demands.

1.2.4 Organization Structure

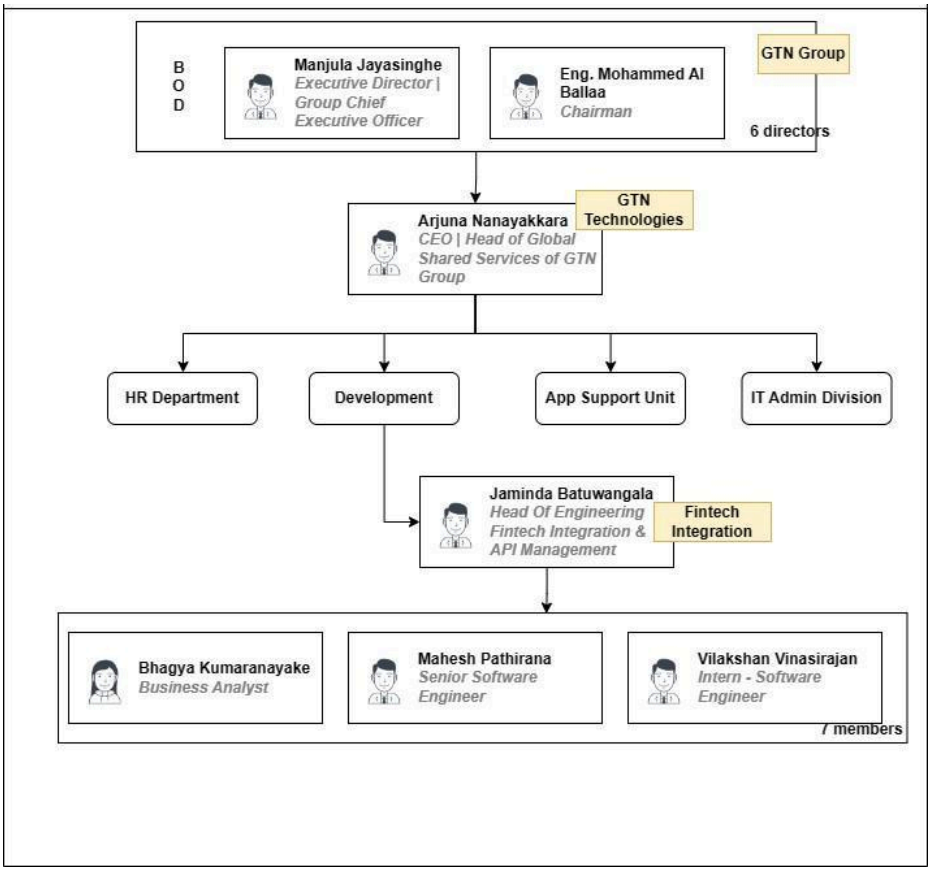


Figure 1.4: Organization structure

1.3 Summary of the training exposure

My industrial training commenced with a research-focused month, dedicated to learning and experimenting with Kubernetes, alongside identifying cost-effective AWS solutions for microservice architecture. This phase was characterized by a hands-on, trial-and-error approach that laid the foundation for my subsequent tasks.

The core of my assignment involved designing a solution diagram for the fintech API, a collaborative effort guided by the expertise of the Tech Lead and Senior Software Engineer. Following the design phase, I was responsible for deploying the application on Kubernetes. As my internship progressed, I delved into performance testing, monitoring error rates, and implementing enhancements to optimize our deployment.

In the final stages of my training, I took personal initiative to thoroughly understand the application's codebase, which enabled me to make necessary adaptations for Kubernetes deployment. By the conclusion of my internship, I had the privilege of contributing to a UAT ready Kubernetes environment, complete with automated monitoring and observability features. This experience not only honed my technical skills but also provided a comprehensive understanding of the practical aspects of software deployment and maintenance.

Chapter 2

Architectural Paradigms: Monolithic vs. Microservices in Modern Software Development

2.1 Introduction

In the realm of software development, the architecture of an application is akin to the blueprint of a building - it dictates the structure, defines the components, and determines how they interact. Two predominant architectural styles have emerged over time: the traditional monolithic architecture and the more contemporary microservice architecture. Each style presents its own set of principles, practices, and patterns that profoundly influence the development, deployment, and scalability of applications.

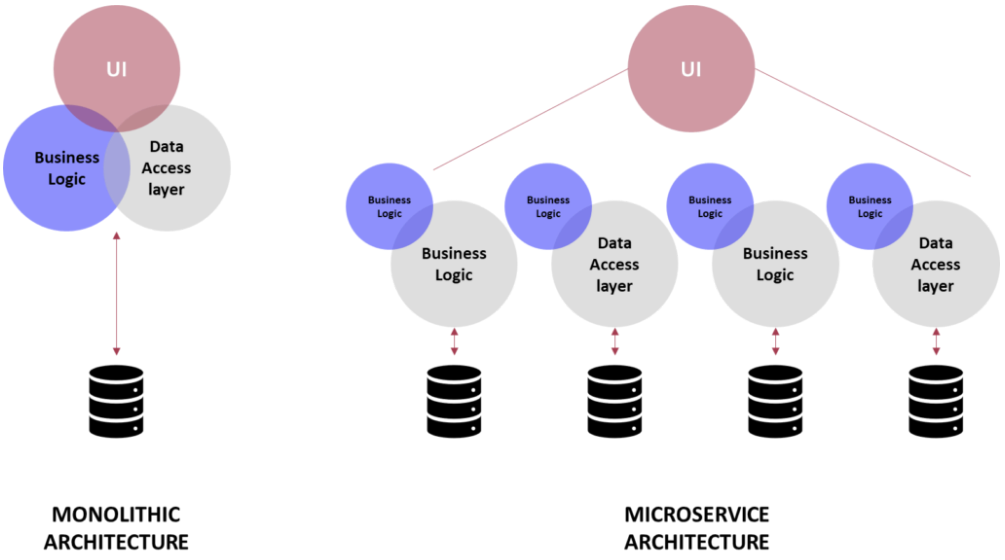


Figure 2.1: Monolithic vs microservice architecture

2.1.1 Monolithic Architecture

A Unified Approach Monolithic architecture is the classical approach to building software applications. It is characterized by a single-tiered software application in which different components are combined into a single program from a single platform. This unified unit is self-contained, meaning all components - ranging from the database operations to the client-side user interface are tightly packaged together.

The monolithic model offers several advantages, particularly for smaller applications or those in the early stages of development. Its simplicity allows for faster development cycles, as developers deal with a single codebase. Deployment is straightforward, typically involving the launch of a single executable or deployment of a single directory. Performance can also be optimized within the centralized codebase, as internal calls within the application are generally faster than inter-service calls in a distributed environment.

2.1.2 Microservice Architecture

A Modular Approach Microservice architecture, on the other hand, is an approach that structures an application as a collection of loosely coupled services. Each service, or microservice, is a self-contained unit with its own business logic and database, responsible for a specific function within the application. These services communicate with each other through well-defined APIs and protocols.

This architectural style is designed to overcome the limitations of monolithic designs, especially for large, complex applications that require scalability and flexibility. Microservices allow for independent development and deployment, which can significantly reduce downtime and improve productivity. They enable teams to deploy updates for specific parts of an application without affecting the whole, and they facilitate scaling by allowing individual services to be scaled as needed.

Evolving Architectural Needs The evolution from monolithic to microservice architecture reflects the changing needs of the software industry. As applications grow in size and complexity, the need for better scalability, flexibility, and maintainability becomes paramount. Microservices offer a solution to these challenges, allowing organizations to adapt quickly to market changes and technological advancements.

In the following subchapters, we will delve deeper into the characteristics of each architectural style, compare their advantages and disadvantages, and explore the factors that influence the choice between a monolithic and a microservice architecture for a given project.

2.2 The Benefits of Microservices Compared to Monolithic Systems

When it comes to modern software development, microservice architecture has become increasingly popular due to its numerous benefits over the traditional monolithic approach. This subchapter will focus on the distinct advantages that make microservice architecture a preferred choice for many organizations.

- **Scalability and Flexibility** One of the most significant advantages of microservices is their inherent scalability. Each microservice operates independently, which means that as demand increases, individual services can be scaled without the need to scale the entire application. This granular scalability is cost-effective and allows for more precise resource management.

- **Resilience and Fault Isolation** Microservices enhance the overall resilience of an application. Since each service is independent, the failure of one does not impact the others. This fault isolation ensures that system-wide failures are avoided, and uptime is maximized.

- **Technological Heterogeneity** Microservices allow for the use of different technologies across various services. Teams can choose the best technology stack for each service based on its unique requirements, leading to more innovative and optimized solutions.

- **Faster Development and Deployment** With microservices, development teams can work on different services simultaneously, leading to faster development cycles. Deployment is also quicker since only the modified services need to be redeployed, not the entire application.

- **Improved Maintenance and Updating** Maintaining and updating applications is more straightforward with microservices. Changes can be made to individual services without affecting the rest of the application, making the process more efficient and reducing the risk of introducing errors.

2.3 Opting for Microservice Architecture in Our Fintech API Project

2.3.1 Rationale for Transitioning to Microservices

Our project's transition to a microservice architecture was primarily driven by the limitations we encountered with our existing monolithic setup on EC2 instances. Initially, our production environment was supported by only two EC2 instances, which hosted four API applications. This configuration posed several challenges:

- **Single Point of Failure:** The entire system's reliability was compromised due to its dependence on a couple of EC2 instances. Any failure would render the applications inaccessible to clients.
- **Resource Contention:** The shared resources of a single EC2 instance led to performance bottlenecks. High traffic on one application could adversely affect the others, leading to suboptimal performance.
- **Inefficient Scaling:** Scaling up due to increased load on one application resulted in underutilized resources for the others. This inefficiency was not only costly but also did not align with our commitment to resource optimization.
- **Delayed Recovery:** The time required to spawn and prepare a new EC2 instance for all four applications in the event of a failure was unacceptable, leading to potential service disruptions.

2.3.2 Implementing a Microservice Solution

To address these issues, we adopted a microservice architecture, which allowed us to encapsulate each of the four applications within its own set of containers. This approach provided several benefits:

- **Improved Fault Tolerance**: By isolating services, we minimized the risk of system-wide outages. The failure of one service would no longer impact the availability of the others.
- **Efficient Resource Utilization**: Independent scaling of services ensured that resources were allocated based on individual application needs, eliminating waste and reducing costs.
- **Rapid Recovery**: The containerized nature of microservices facilitated quicker recovery times, as new instances could be launched independently and swiftly for each service.
- **Enhanced Performance**: The decoupling of applications allowed for dedicated resources, ensuring that high demand on one service did not affect the performance of the others.

2.4 Selecting AWS EKS for Our Fintech API Project

In the development of our Fintech API project, the choice of container orchestration service was pivotal. After careful consideration, we opted for Amazon Elastic Kubernetes Service (EKS) over Amazon Elastic Container Service (ECS) for several compelling reasons that align with our project's needs and goals.

- **Kubernetes Ecosystem and Community Support:** EKS is based on the widely-adopted Kubernetes platform, which benefits from a large, active community and a rich ecosystem of

tools and add-ons. This extensive support network can be invaluable for troubleshooting, innovation, and staying current with the latest advancements.

- **Flexibility and Portability:** EKS provides the flexibility to run applications both on AWS and on-premises, facilitating a hybrid cloud strategy. This portability ensures that you are not locked into the AWS cloud and can easily migrate workloads if necessary.

- **Scalability**: While both EKS and ECS offer scalability, EKS allows for more granular control over autoscaling. Kubernetes' Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) enable precise scaling of pods based on CPU, memory usage, or custom metrics.

- **Advanced Load Balancing**: EKS supports advanced load balancing capabilities through Ingress resources and the ALB controller, which can automatically manage the deployment of load balancers and target groups, streamlining traffic distribution and improving application availability.

- **Security and Compliance**: EKS leverages the strong security foundation of AWS while also providing the security features of Kubernetes, such as role-based access control (RBAC), network policies, and pod security policies. This combination helps maintain a secure and compliant environment for containerized applications.

- **Integration with AWS Services:** EKS integrates with a wide range of AWS services, providing a seamless experience for monitoring, logging, security, and more. This integration simplifies the management of containerized applications and enhances their performance.

2.5 Gaining Insights from Adopting Microservices

In the early stages, our project utilized AWS ECS for deployment. However, after thorough evaluation and team discussions led by our team leader, we decided to transition to AWS EKS. This move was informed by the benefits outlined in Subchapter 2.4 of our report.

This period was a significant learning curve for me. Coming into the project, my understanding of Kubernetes and microservices was minimal. But within two weeks, I had grasped the fundamentals of Kubernetes, AWS resources, and the distinctions between AWS ECS and EKS.

The knowledge I gained was encapsulated in a report comparing AWS EKS with ECS, which I presented to our team. This report not only affirmed our choice to proceed with microservices but also reflected the depth of understanding I had achieved during this explorative phase.

Chapter 3

Deploying and Managing Applications with Kubernetes

3.1 Introduction

Embarking on the journey of deploying and managing applications within a Kubernetes environment has been a cornerstone of my industrial training experience. This chapter recounts the hands-on process I undertook, starting from the initial steps of containerizing our application to the final stages of hosting it on a Kubernetes cluster. The narrative will unfold through a series of strategic actions: building the application into a JAR file, transforming that JAR into a Docker image, architecting a Kubernetes deployment solution, and ultimately, ensuring the application's smooth operation within the cluster. Each phase presented its own set of challenges and learnings, all of which contributed to my deeper understanding of microservices and container orchestration.

3.2 Containerization of Applications: From JAR to Docker Image

3.2.1 Building the JAR File

The initial phase of my project involved compiling executable JAR files for our suite of fintech applications: boService, foService, authService, and tradestreamService. These services, built on the Spring Boot framework, were reliant on several proprietary dependencies, specifically JAR files produced by our company's TRTE team. Upon attempting to run the applications locally, I was met with errors indicating missing dependencies. These were not available in public Maven repositories, necessitating a build from internal sources. To resolve this, I meticulously examined the pom.xml files of each application to identify the missing internal dependencies. For each dependency that was not publicly accessible, I cloned the corresponding Maven projects from our internal GitLab repository to my local machine. Here are the steps I followed:

Cloning the Repository::

```
git clone <repository-url>
```

Building the Maven Project:

```
mvn install -DskipTests
```

This command compiles the source code, and installs the resulting JAR file into my local Maven repository. After successfully building and installing the internal dependencies, I proceeded to

compile the main applications. With the dependencies resolved and the JAR files for the main applications built, I was able to run the applications on my local host, marking a successful conclusion to the initial setup phase of my project.

3.2.2 Implementing a GitLab CI/CD Pipeline for JAR Deployment

I was tasked with enhancing the efficiency of our development process by automating the build process of JAR files. To achieve this, I crafted a GitLab CI/CD pipeline. But I took on the challenge of refining our development workflow by introducing a manual trigger for our GitLab CI/CD pipeline. This strategic decision was made to prevent the pipeline from running unnecessarily with every code push, thus optimizing our resources.

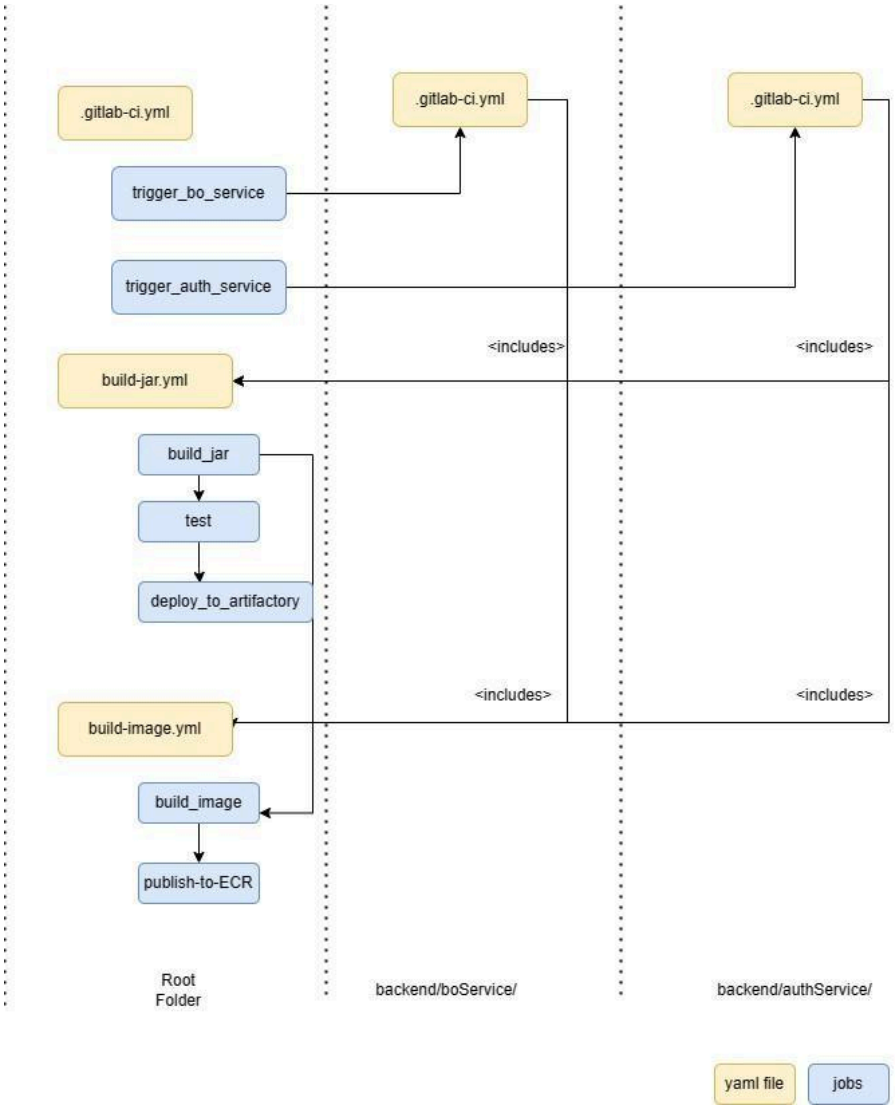


Figure 3.1: Automated JAR and Docker image deployment pipeline

Figure 3.1 provides a visual representation of the pipeline architecture I implemented. It's important to note that while the figure shows the process for two applications, the same setup was replicated across all four applications: boService, foService, authService, and tradestreamService. To avoid redundancy and maintain consistency, I utilized the GTN PE team's predefined templates for building the JAR file and deploying it to Artifactory. By including the template in the folder, we can call upon it from the application's .gitlab-ci.yml file.

By implementing this manual trigger, I have provided our team with the flexibility to run the pipeline at our discretion, aligning the build process with our development needs and maintaining control over the CI/CD process. I introduced a job to deploy our JAR files to Artifactory. Here's a simple explanation for why I chose to do this: Artifactory acts like a safe storage space for the JAR files we create during our build process. By using Artifactory, we make sure that these JAR files are stored in a place where they can be easily accessed later on. This is particularly useful when we want to build Docker images, as we can quickly grab the JAR files we need from Artifactory without having to rebuild them every time.

3.2.3 Containerization: Encapsulating the Application

I began by authoring a Dockerfile to locally build a Docker image and push it to AWS Elastic Container Registry (ECR). Recognizing the potential for further streamlining, I later integrated this process into our GitLab CI/CD pipeline, enabling the automated building of the image as soon as the JAR file was ready, followed by pushing it to AWS ECR.

Refer to Figure 3.1 in the build-img.yml file to understand the two critical jobs I established: build_image and publish_to_ecr. The build_image job is configured to execute upon the successful completion of the JAR building job discussed in the previous section. It utilizes the Dockerfile located in the root folder to construct the Docker image. Once the image is built, it is saved as a tar file, preparing it for the next phase. The subsequent job, publish_to_ecr, begins by authenticating with AWS and acquiring the necessary IAM role. This role grants the permissions required to deploy the Docker image resource to AWS ECR. By automating these steps, I not only ensured a consistent and error-free deployment but also significantly reduced the time between development and deployment.

3.3 Launching Applications in Kubernetes: Replica Management and Resource Allocation

When launching applications in Kubernetes, several critical decisions shape the efficiency and reliability of the deployment. Among these are the number of pod replicas, resource requests and limits, and scaling strategies. Here's an overview of the approach we took:

3.3.1 Minimum Replica Count

We decided on a minimum of two pod replicas for our main applications, as shown in Figure 3.2. This redundancy ensures high availability; if one pod fails, the other is immediately ready to handle incoming traffic, minimizing downtime. A single pod scenario would introduce a delay as Kubernetes waits for a new pod to spawn and become ready, which could impact service continuity.

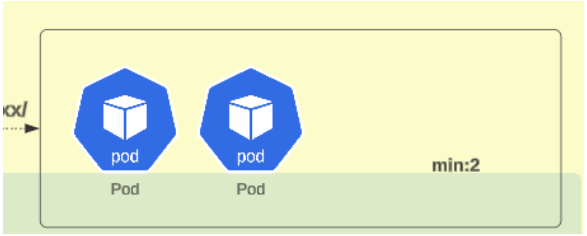


Figure 3.2: minimum 2 replicas for each application

3.3.2 CPU and Memory Resource Allocation

For CPU and memory resources, we chose values on the lower end of what's available on AWS Fargate, as our initial traffic expectations were modest. Specifically, we allocated 0.5 vCPU and a proportional amount of memory for our needs. In Kubernetes, requests are what the container is guaranteed to get, and limits are the maximum resources a container can use. By setting these values, we ensure that our applications have enough resources to run under normal conditions but are also restricted from using excessive resources which could affect other pods.

Our pods consist of two containers: the main application container and a Datadog agent container running as a sidecar. The Datadog agent collects resource metrics from the main application and sends them to the Datadog platform for monitoring. We allocated **450m** of CPU for the main application container and **50m** for the Datadog agent, as AWS Fargate allocates resources to a pod based on the sum of the resources requested by all containers within it.

3.3.3 Scaling Strategies

With AWS Fargate, we focused on horizontal scaling rather than vertical scaling. Since we set the same value for both requests and limits, there's no room for a pod to scale vertically by acquiring more resources than initially requested. Instead, we implemented horizontal scaling based on CPU utilization. Our configuration specifies a targetCPUUtilizationPercentage of **80%**. This means that if the CPU utilization reaches 80%, Kubernetes will automatically start another pod instance to balance the load, ensuring that our applications remain responsive under increased demand.

Chapter 4

Mastering Deployment on AWS with Kubernetes

4.1 Introduction

The chapter begins with an exploration of AWS resources. Instead of setting up IAM roles and policies from scratch, I leveraged the existing roles provided by the PE team. My task was to fine-tune these roles by adding the necessary permissions that would allow our fintech team to deploy our applications securely and efficiently.

Moving forward, I established our network infrastructure within AWS. This involved creating a VPC, configuring NATs and subnets, and ensuring all resources were correctly tagged. These steps were critical in laying down a secure and organized network for our applications. At the heart of the deployment process was the management of the Kubernetes cluster. I evaluated EKS Fargate Pods against Managed Node Groups, choosing the best options for our application needs. This section also covers the setup of ingress controllers, the use of kubectl tools, and the authentication mechanisms within Kubernetes. Additionally, I integrated the API Gateway to manage incoming API traffic effectively. This was a key component in bridging the gap between user requests and our Kubernetes services.

Lastly, I implemented AWS Lambda functions for authentication. These serverless functions were crucial in authenticating and authorizing user requests, adding a robust layer of security to our deployment.

4.2 Foundations of AWS and IAM

I was introduced to the AWS Management Console, a pivotal tool for managing AWS services. My initial task was to understand the deployment of Kubernetes on AWS. I initially deployed on AWS ECS Fargate, and subsequently transitioned to AWS EKS. To facilitate this, I delved into the documentation, gaining insights into the intricacies of these services.A key challenge in the learning process was grasping the permissions linked to the IAM role named GMAAutomation. This role, created by the PE team, had fundamental privileges like reading and writing to S3 buckets and writing logs to CloudWatch. It served as a shared tool across all GTN groups for deploying resources in development or QA setups.

As Kubernetes was a new area for our company, it became necessary for me to update this role with additional permissions to enable the creation of a Kubernetes cluster. At first, I used an IAM user to do tasks manually, which helped me understand what permissions were needed. But this approach was temporary; the role was meant for GitLab pipeline jobs to automate deployments, not for manual execution by users.The additional permissions I included in the GMAAutomation role were chosen specifically to give the role the necessary abilities for Kubernetes tasks. This primary permission is: AmazonEKSClusterPolicy - AWS Managed Policy

4.3 Constructing a VPC Infrastructure for Kubernetes Deployment

I worked on setting up a network in AWS to run a Kubernetes cluster. This section explains the parts of the network and how they work together. VPC and Its Components. A VPC is the cornerstone of AWS networking, providing a private, isolated section of the cloud where you can launch resources in a defined virtual network.

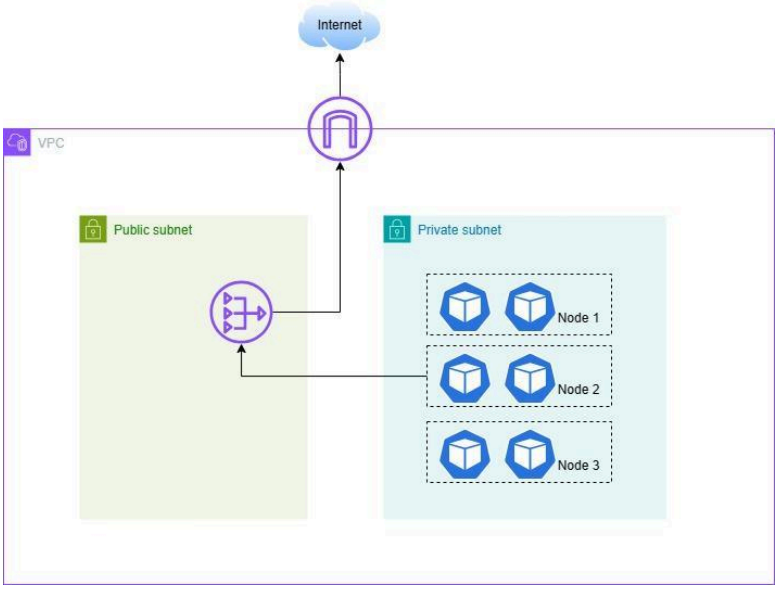


Figure 4.1 Initial Setup of Basic VPC and Components

The primary components of a VPC include subnets, Network Address Translation (NAT) gateways, routing tables, and VPC peering connections.

Subnets are pivotal in network design, dividing the VPC into distinct segments. There are two types of subnets:

- **Public Subnets** facilitate direct access to the internet via an internet gateway, allowing resources like load balancers to serve external traffic.
- **Private Subnets** are used for resources that should not be directly accessible from the internet. Here, I placed the Kubernetes nodes and node groups to ensure an additional layer of security, as these subnets do not have a route to the internet gateway.

NAT Gateways are employed to enable instances in a private subnet to initiate outbound traffic to the internet (for updates, patches, or downloads), while preventing inbound traffic from the internet.

Routing Tables define the rules for traffic direction within the VPC. Each subnet is associated with a routing table that dictates the flow of traffic from that subnet.

VPC Peering Connections allow for the networking of two VPCs, enabling resources in different VPCs to communicate as if they were in the same network. This was particularly useful for connecting the Kubernetes cluster with a database hosted in a separate AWS account's VPC.

When deploying the Kubernetes cluster, I adhered to AWS best practices by utilizing private subnets for the worker nodes and node groups. This strategic placement ensures that no external requests can directly reach our applications or nodes, fortifying the security of our system. The VPC peering connection was established to facilitate seamless communication between the Kubernetes cluster and the database service, both residing in private subnets across different AWS accounts. This setup allowed for secure and efficient data exchange necessary for the applications' functionality. Furthermore, the NAT gateway played a crucial role in the deployment process. It allowed the Kubernetes pods to retrieve images from external repositories, a vital step for the instantiation of containers based on these images. Figure 4.1 depicts the initial setup of my basic VPC and its components. However, the updated diagram will showcase the complete integration of multiple components that I eventually established.

In setting up the network for our Kubernetes cluster, I implemented **tagging** on the subnets within our VPC. This was a deliberate action to ensure the AWS Load Balancer Controller could automatically identify and use the correct subnets for creating an Application Load Balancer (ALB). In section X.X, I'll detail the process of enabling the AWS load balancer controller and explain the steps I took to deploy the load balancer.

I used specific tags
like `kubernetes.io/cluster/cluster-name` and `kubernetes.io/role/internal-elb` or `kubernetes.io/role/elb`, depending on whether we needed an internal or internet-facing load balancer. These tags are essential for the controller to recognize which subnets are available for the ALB and to make sure it's created in a subnet that matches our architectural and security requirements. By carefully tagging the subnets, I streamlined the ALB setup process, allowing the controller to do its job without manual intervention. This not only saved time but also reduced the potential for human error, ensuring the smooth and secure operation of our Kubernetes applications.

4.4 Kubernetes Cluster Deployment

4.4.1 EKS Fargate Pods vs. Managed Node Groups

Comparing the two approaches and discussing the rationale behind the choices made for application hosting.

EKS Fargate Pods Features:

- Serverless: Automatically manages the underlying infrastructure, scaling, and scheduling of containers.
- Isolation: Each pod runs in its own isolated environment, enhancing security.
- Pay-as-you-go: Costs are based on the actual amount of resources (CPU and memory) consumed by the pods.
- Simplified Operations: No need to select server types, decide when to scale your clusters, or optimize cluster packing.

AWS EKS Managed Node Groups Features:

- Control: Offers granular control over EC2 instances, including instance types and configurations.
- Spot Instance Integration: Supports using EC2 Spot Instances for cost savings on non-critical workloads.
- Auto-Scaling: Can automatically adjust the number of nodes based on workload demands.
- Customization: Allows for custom AMIs, enabling specialized hardware or software configurations.

EKS Fargate does not offer certain capabilities that are available with AWS EKS Managed Node Groups.

- Faster Pod Start Time
- Image Caching
- Special Hardware Requirements (e.g., GPUs)

In Chapter X, I will explore the strategies I employed to overcome the previously mentioned challenges, including the reduction of pod startup times.

Given our scenario, we adopted a hybrid strategy:

Main Applications on EKS Fargate

For our core services—boService, foService, authService, and tradestreamService—we chose EKS Fargate Pods. This decision was driven by the need for independent resource allocation and the ability to autoscale each service based on its unique demands. The serverless nature of EKS Fargate simplifies operations by abstracting away the server management and ensuring that each application has dedicated resources, thus providing consistent performance without the complexity of managing underlying servers.

Supporting services on Managed Node Groups

Conversely, our supporting services, which include metrics collection, AWS Secrets, alb controller and other components, are deployed on AWS EKS Managed Node Groups. I will discuss more about this in later chapters. These services have a consistent resource consumption pattern and do not necessitate frequent scaling. Therefore, a single, minimal EC2 instance within a Managed Node Group suffices for their needs, offering a cost-effective solution without the overhead of unused capacity. We have 15 support service pods, and I selected the t2.medium instance because it can support up to 17 pods on EC2. The Figure 1.2 will illustrate this information.

Instance Type	Maximum Pods
t2.2xlarge	44
t2.large	35
t2.medium	17
t2.micro	4
t2.nano	4
t2.small	11
t2.xlarge	44

Figure 4.2 : Instance Types and Maximum Pods Supported

To accommodate this dual approach, we've segregated our workloads into different namespaces: the default namespace is configured to utilize EKS Fargate for our main applications, while the other namespace is designated for Managed Node Groups, which host our add-on services.

4.4.2 In-Depth Exploration of Kubernetes Services Implementation

In Chapter 1.3, I outlined the deployment strategies and scaling policies for our applications. To facilitate user interaction with these applications, it was essential to implement Kubernetes services. Services in Kubernetes serve as a stable interface to manage access to pods, ensuring consistent communication regardless of the underlying pod changes.

The Role of Services in Kubernetes

Services are fundamental to Kubernetes networking. They allow us to abstract pod IP addresses from consumers and ensure that traffic is directed to the right pods. Let's delve deeper into the types of services available:

- ClusterIP: This default type restricts access to within the cluster, making it suitable for internal communications.
- NodePort: Opens a specific port on all nodes and forwards traffic to the corresponding pods, making it useful for external access when more complex ingress is not required.
- LoadBalancer: Seamlessly integrates with cloud providers' load balancers, providing an easy way to expose services to the internet.
- ExternalName: Maps a service to a DNS name rather than to a typical selector such as app or service.

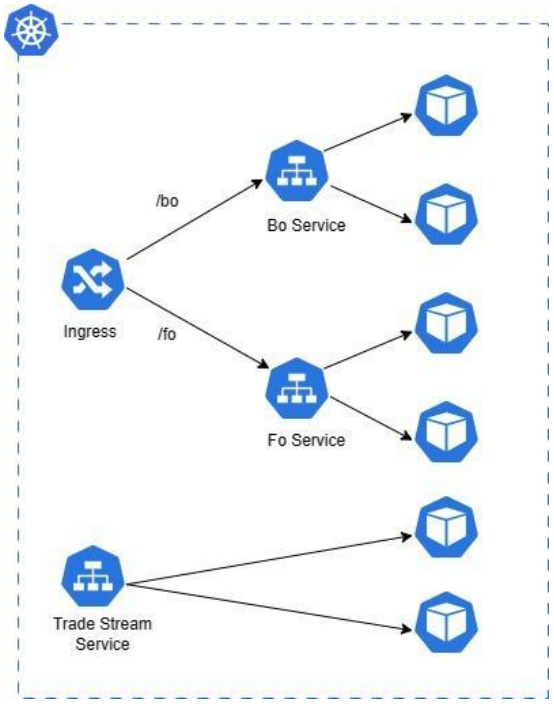


Figure 4.3 : Kubernetes deployment

For direct exposure of applications, I employed NodePort services. This approach allowed me to map an external port on each node to the pods' target port. Here's a more detailed look at the service definition for boservice.

```
apiVersion: v1
kind: Service
metadata:
  name: bo-service
  namespace: default
spec:
  ports:
    - port: 80
      targetPort: 8723
      protocol: TCP
  type: NodePort
  selector:
    app: boservice-container
```

In this configuration, any traffic that hits port 80 on the service will be forwarded to port 8080 on the pods selected by the `app: boservice-container` label. The `nodePort: 30007` is the port on the nodes where external traffic can reach the service.

For the trade streaming application, which utilizes WebSocket connections, I chose a Network LoadBalancer due to its ability to handle long-lived connections. By setting the service type to LoadBalancer and annotating it as internet-facing, I ensured that the service is exposed to the internet, bypassing the need for API Gateway and thus reducing costs.

For other applications, I implemented an Ingress controller, which provided more granular control over traffic routing using path-based rules. The installation of the ALB controller, as referenced in subsection 4.2, meant that the Ingress resources would trigger the creation of an Application LoadBalancer. The integration with AWS services meant that target groups were automatically created and associated with the correct subnets and instances. This automation is part of the power of Kubernetes on AWS; it simplifies the process of connecting pods to the AWS load balancing ecosystem. In Figure 4.2, the Kubernetes deployment is shown.

By leveraging Kubernetes' service types and AWS's load balancing features, we created a resilient and scalable environment that could adapt to our applications' needs. I discovered that Fargate pods operate differently from traditional EC2-based Kubernetes setups. Fargate provides a serverless environment that manages the underlying infrastructure, allowing me to focus solely on the pods. This meant that for my Fargate deployments, I didn't need to use NodePort services since there were no nodes to expose.

Instead, I found that using a LoadBalancer or an Ingress was the right approach to expose services in Fargate. These methods are fully integrated with AWS's infrastructure, simplifying the process of routing and exposing applications. In contrast, when working with a conventional EKS cluster backed by EC2 instances, NodePort services are indeed created on all nodes. Each node listens on the designated NodePort, directing traffic to the appropriate pods. However, this is not the case with Fargate, where the serverless nature calls for different service types that align with its operational model.

4.4.3 EKS Add-ons: Enhancing Cluster Functionality

In the pursuit of optimizing our Kubernetes cluster on AWS EKS, I've integrated several add-ons that enhance its functionality, streamline operations, and bolster security. Here's a deep dive into each add-on and the rationale behind their inclusion

Fargate Fluent Bit is a log processor and forwarder which allows me to unify data collection and consumption for better observability. By deploying Fluent Bit as a logging solution on Fargate, I've ensured that all logs from our containers are efficiently collected, processed, and forwarded to Kinesis Firehose. The logs are then delivered to Datadog, a powerful visualization platform, providing us with actionable insights into our application's performance and user experience.

The **AWS Load Balancer Controller** manages AWS load balancers for a Kubernetes cluster. It simplifies the process of setting up and configuring AWS Application Load Balancers (ALB) for our Ingress resources. By using this controller, I've automated the creation and updating of ALBs, ensuring that our services are highly available and can scale with demand.

The **Metrics Server** is a scalable, efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines. It collects CPU and memory usage data, which is vital for the Horizontal Pod Autoscaler to make informed decisions about scaling our applications in or out based on demand. Integrating the Metrics Server was essential for enabling real-time monitoring of resource utilization, allowing us to maintain optimal performance levels without over-provisioning resources, thus optimizing costs.

External Secrets enable Kubernetes to access external systems' secrets, such as API keys, without storing them in the cluster. This add-on is particularly important for maintaining the security and integrity of sensitive information. For instance, we've used it to securely manage the Datadog API key, which is critical for our monitoring setup.

4.4.4 Deployment with Terraform

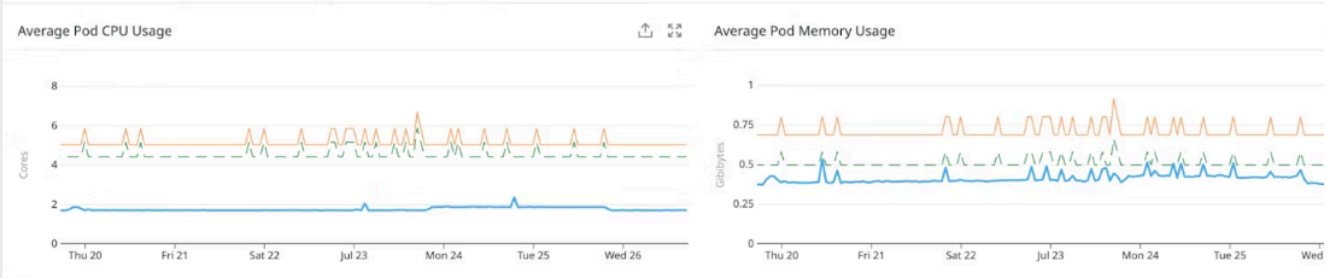
Deploying a Kubernetes cluster on AWS can be a complex task, especially when integrating diverse components like EKS managed node groups and Fargate profiles. Initially, I attempted to script the deployment using individual AWS modules in Terraform. However, this approach proved challenging, as I encountered difficulties getting both the managed node groups and Fargate profiles to function simultaneously within the same cluster.

To overcome these challenges, I turned to two robust solutions: terraform-aws-eks from terraform-aws-modules, and terraform-aws-eks-blueprints-addons from aws-ia. These repositories, maintained by Amazon, provided me with high-level abstractions and pre-configured modules that greatly simplified the deployment process. By leveraging these resources, I was able to focus more on productivity and less on the intricacies of implementation.

One of the key contributions I made to the terraform-aws-eks-blueprints-addons was the Grant IAM policy permissions to allow Fargate Fluent Bit to send logs to Kinesis Data Firehose. To execute the Terraform scripts, I utilized the GMAAutomation role, previously discussed in Section X.X. This role was specifically designed to encapsulate the necessary permissions and streamline the deployment process to AWS, ensuring a secure and efficient setup.

The combination of these Terraform modules and the automation role has significantly improved our deployment workflow, allowing us to enjoy a more seamless and manageable Kubernetes experience on AWS.

4.4.5 Enhancing Kubernetes Monitoring with Datadog



- Metrics Collection with Datadog Agent Sidecar

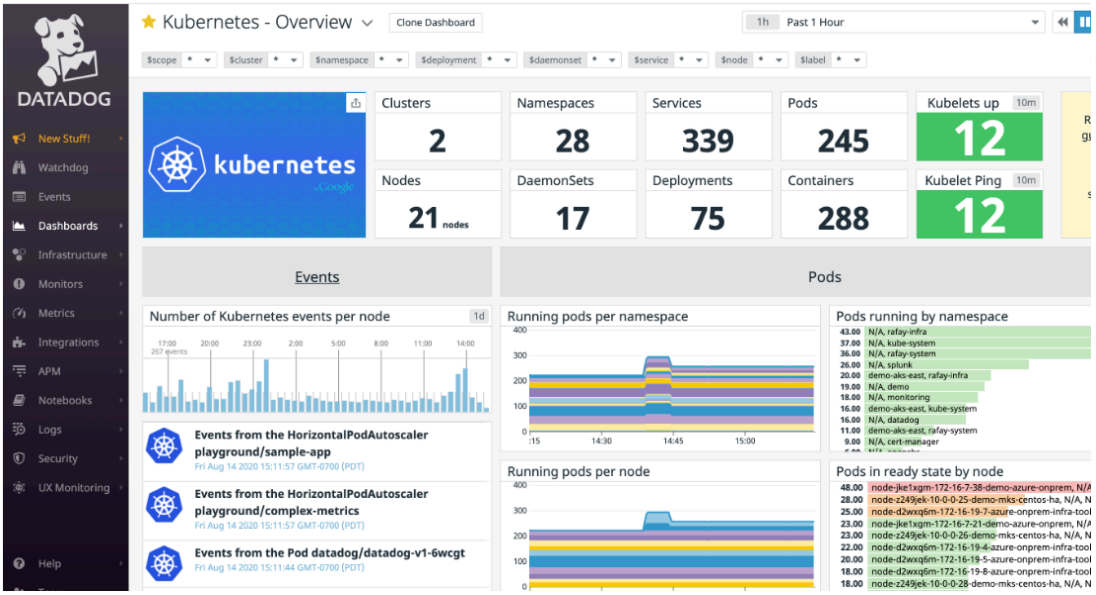
To capture critical performance metrics like CPU and memory utilization, we integrated the Datadog Agent as a sidecar container within our Kubernetes pods. This setup allowed us to collect detailed metrics from each pod and send them directly to Datadog. The sidecar approach ensured that the monitoring was tightly coupled with our application, providing real-time insights into resource usage and potential bottlenecks.

- Log Forwarding with Fluent Bit and Kinesis Firehose

Initially, we experimented with sending our logs to AWS CloudWatch. However, we encountered significant costs associated with log storage and retrieval. To optimize our expenses, we shifted to using Fluent Bit to forward logs to Amazon Kinesis Firehose. This change not only reduced costs but also streamlined the delivery of logs to Datadog for analysis. The transition from CloudWatch to Kinesis Firehose proved to be a cost-effective solution for our logging needs.

- Cluster-Level Insights with Datadog Cluster Agent

Deploying the Datadog Cluster Agent was a strategic move to gain a centralized view of our Kubernetes cluster. The Cluster Agent collects events, provides a comprehensive view of Kubernetes resources, and performs cluster checks. This level of monitoring is invaluable for maintaining the health and performance of the cluster, as it helps us identify issues at the cluster level before they impact our applications.



- Real-Time Trace Analysis with Datadog APM

we focused on real-time trace analysis to gain insights into the performance of each API endpoint. By leveraging Datadog's APM tools, we were able to capture and analyze traces, which provided us with valuable observability metrics.

Latency Metrics: We tracked the response times of our API endpoints, which is crucial for identifying performance bottlenecks. Latency metrics helped us understand the impact of our code changes and infrastructure adjustments on the user experience.

Flame Graphs: These visualizations were particularly useful for pinpointing issues within our stack. Flame graphs represent the nested calls that occur during a request's execution, with each block's width indicating the amount of time spent in that method or function. This allowed us to quickly identify which parts of our code were consuming the most resources and needed optimization.

By implementing these features, we ensured that our team could monitor our applications' performance in real-time, troubleshoot efficiently, and maintain a high standard of service reliability and user satisfaction.

4.4.5 Health Checks and Annotations

Health checks are vital for maintaining the reliability of services. They ensure that traffic is only sent to healthy pods. The annotations I used in the Ingress resource for health checks were:

```
alb.ingress.kubernetes.io/healthcheck-path: /actuator/health
```

```
alb.ingress.kubernetes.io/healthcheck-interval-seconds: '20'
```

```
alb.ingress.kubernetes.io/healthcheck-timeout-seconds: '15'
```

These annotations tell the ALB where to send health check requests (/actuator/health), how often (every 20 seconds), and how long to wait for a response before considering the check failed (15 seconds). By incorporating the Spring Actuator into my Java applications, I provided a robust health check endpoint that the ALB could use to assess the pods' status.

The implementation of these services and health checks was a critical part of our Kubernetes infrastructure. It ensured that our applications were not only accessible to users but also maintained high availability and performance.

In my deployment strategy, I've come to understand the importance of both liveness probes and Ingress health checks. Liveness probes are a Kubernetes feature that helps me ensure my containers are restarted if they become unresponsive. They're like an internal watchdog that keeps an eye on my application's health at the container level. On the other hand, I've used Ingress health checks, configured through annotations, to inform the AWS Load Balancer about the health of my services. These checks are crucial for routing traffic, as they help the load balancer decide which pods are healthy enough to receive traffic. I've learned that while Ingress health checks are great for traffic management, they don't replace the need for liveness probes. By implementing liveness probes, I add an extra layer of resiliency within the Kubernetes ecosystem, ensuring that any pod issues can be self-healed without relying solely on external infrastructure signals. So, in my setup, I've made sure to use both health check mechanisms. This dual approach gives me confidence that my applications are not only performing well but are also set up to recover quickly from any potential issues, ensuring high availability and a seamless user experience.

4.4.6 Kubernetes Tools and Authentication

Kubernetes command-line tool, kubectl, was utilized to interact with the AWS EKS cluster. The kubectl was configured to connect to the AWS EKS cluster by first setting up the necessary credentials and specifying the cluster context. This setup allowed for the management of cluster resources and the deployment of applications seamlessly.

The connection to the AWS EKS cluster was established through the kubectl command, which involved configuring the kubeconfig file. the AWS EKS command-line interface was utilized to directly set up kubectl. This method streamlined the process of configuring kubectl to communicate with the EKS cluster. By executing the command

```
aws eks update-kubeconfig --name <cluster-name> --region <region>
```

the kubeconfig file was automatically updated. This command fetched the necessary details such as the cluster API endpoint and the authentication token required for kubectl to connect to the EKS cluster.

Furthermore, the AWS-auth ConfigMap was employed to manage access to the EKS cluster. The ConfigMap, known as aws-auth, was crucial for mapping AWS IAM roles and users to Kubernetes RBAC permissions. It was edited and updated using kubectl edit configmap aws-auth -n kube-system, which allowed for the addition of new IAM roles and users.

Specifically, IAM user accounts other than the one associated with the GMAAutomation role, which was used initially to create the cluster, were granted access to the EKS cluster. This was achieved by adding entries to the aws-auth ConfigMap under the mapUsers section. Each entry included the user ARN, username, and groups, granting the specified IAM users the necessary permissions to perform cluster operations according to their assigned Kubernetes roles.

This subsection demonstrates the practical use of kubectl for cluster connection and the strategic employment of the AWS-auth ConfigMap for managing user and role-based access within an AWS EKS environment. These tools and techniques were essential for the effective management and operation of the Kubernetes cluster during the training period.

4.5 API Gateway Integration

4.5.1 Why Use AWS API Gateway Before the Application Load Balancer

The decision to implement AWS API Gateway before the Application Load Balancer, which routes traffic to the Kubernetes services, was driven by several crucial factors aimed at enhancing the overall system architecture's performance, security, and manageability.

- **Throttling and Traffic Management**: AWS API Gateway offers built-in throttling capabilities, which are vital for managing the rate of incoming requests to prevent overloading backend services. By setting rate limits and burst limits, the API Gateway ensures that the traffic directed towards the Kubernetes backend remains within manageable levels, preserving the system's stability and reliability.
- **Security with Lambda Authorizer**: Integrating a Lambda Authorizer adds an additional layer of security by enabling token-based authentication and authorization before requests reach the backend services. This function evaluates the incoming tokens, ensuring that only legitimate and authorized requests are allowed to pass through to the Application Load Balancer.
- **Integration and Routing Decisions**: The use of AWS API Gateway allows for sophisticated routing decisions. For instance, based on the configuration in the API Gateway, requests can be directed to different backend services. This is particularly beneficial in a microservices architecture where different components may require differentiated routing logic.
- **VPC Link Connection**: The necessity for a VPC Link stems from the requirement to connect the API Gateway, a public service, to the private Application Load Balancer managed within an AWS VPC (Virtual Private Cloud). The VPC Link acts as a bridge, securely channeling traffic from the API Gateway to the internal load balancer. This setup is crucial for maintaining the private nature of the backend infrastructure while still allowing controlled access through the public-facing API Gateway.
- **OpenAPI Specification for API Management**: AWS API Gateway supports the OpenAPI specification, which is a standard for defining RESTful APIs. By using the OpenAPI specification to import JSON files, all the API's endpoints, including their integration settings and Lambda authorizer configurations, are defined in a structured format. This capability not only simplifies the API setup process but also enhances consistency and ease of management across different stages of the API lifecycle.

These features collectively motivated the choice to implement AWS API Gateway as a pivotal component of the infrastructure, positioned strategically before the Application Load Balancer to optimize the handling, security, and routing of requests to the Kubernetes services.

4.5.2 Implementation of AWS API Gateway with Lambda Authorizer and AuthService

In managing and securing the services deployed on Kubernetes, a crucial role is played by the AWS API Gateway, coupled with a Lambda Authorizer and the AuthService. This setup ensures a robust authentication and authorization mechanism is in place for accessing the services.

Initial Access Token Acquisition,the process begins with users obtaining an access token through the AuthService, which offers serverToken and clientToken endpoints. Upon submitting their credentials at either of these endpoints, users are granted an access token, which they must include in the header of subsequent service requests.

Role of Lambda Authorizer, the Lambda Authorizer is configured to intercept all service requests to ensure they are authenticated. For services other than AuthService, the Lambda Authorizer extracts the access token from the header of the incoming request and communicates directly with the AuthService via the Application Load Balancer to verify the token. This direct communication is possible because both the Lambda function and the Load Balancer are within the same VPC, thereby eliminating the need for a VPC Link in this particular interaction.

Token Verification and Response Handling, the AuthService, developed using Spring Boot, checks the validity of the token at its /auth/verifyToken endpoint. Typically, a successful verification returns a User object, indicating that the user is authenticated and authorized to proceed. This object includes essential user information and their consent status.

Request Authorization Decision, If the AuthService confirms the user's authentication by returning a User object, the Lambda Authorizer permits the request to proceed by routing it appropriately through the API Gateway. This decision is facilitated by returning a boolean value true from the Lambda function. Conversely, if the verification fails, the request is denied, and the user receives an authentication error.

Integration with VPC Link, For services other than AuthService, a VPC Link is utilized to secure and streamline communications between the API Gateway and the Kubernetes services accessed through the Application Load Balancer. This setup ensures that all traffic, except direct AuthService requests, passes through a controlled and secure environment.

API Gateway Configuration , Within the API Gateway, routes are meticulously configured to ensure that each request, except those directed at the AuthService, passes through the Lambda Authorizer. This guarantees comprehensive security checks and prevents unauthorized access to the system’s backend services.

Figure X: Detailed Workflow of the Lambda Authorizer in the API Gateway Integration, Highlighting Direct and VPC Link Communications

4.5.3 Utilizing OpenAPI Specification for AWS API Gateway Configuration

The implementation of AWS API Gateway endpoints and their associated configurations was greatly facilitated by the use of the OpenAPI Specification. This specification allows for a structured and standardized documentation of API interfaces, which can be directly utilized to set up and configure API endpoints, integrations, and authorizers in the AWS environment.

Endpoint Configuration with OpenAPI Specification:

The OpenAPI specification was used to define the HTTP API's endpoints within the API Gateway. This included specifying the available paths, the operations allowed on these paths, and the parameters they

accept. By employing the OpenAPI format, it was ensured that the API Gateway setup was not only consistent with industry standards but also fully compatible with the AWS infrastructure.

Integration Setup Using OpenAPI Components:

A significant part of configuring the API Gateway involved setting up the integrations, which dictate how requests are forwarded from the API Gateway to the backend services. Using the x-amazon-apigateway-integrations component in the OpenAPI specification, specific integration behaviors were defined. For instance, the snippet:

```
"integration1": {
  "payloadFormatVersion": "1.0",
  "connectionId": "${vpclink_id}",
  "type": "http_proxy",
  "httpMethod": "ANY",
  "uri": "${loadbalancer_listener_arn}",
  "connectionType": "VPC_LINK",
  "requestParameters": {
    "append:header.username": "$context.authorizer.username",
    ...
  }
}
```

This configuration specifies that the API Gateway should use a VPC Link (connectionType: "VPC_LINK") to forward requests to the backend via the load balancer. The integration is set to support any HTTP method (httpMethod: "ANY"), indicating flexibility in handling different types of requests. The requestParameters section enriches the forwarded requests with additional headers extracted from the context of the Lambda authorizer, such as the username and role, enhancing the backend service's ability to perform context-aware processing.

Lambda Authorizer Configuration:

The security of the API endpoints was a paramount concern, addressed by implementing a Lambda authorizer. Configured through the securitySchemes in the OpenAPI document, the authorizer setup specifies how the Lambda function should be invoked and what data it should expect:

```
"lambda-authorizer": {
  "type": "apiKey",
  "name": "Authorization",
  "in": "header",
  "x-amazon-apigateway-authorizer": {
    "type": "request",
    "identitySource": "$request.header.Authorization",
    "authorizerUri": "${authorizerUri}",
    "authorizerPayloadFormatVersion": "2.0",
```

```
    "enableSimpleResponses": true
  }
}
```

Here, the Lambda authorizer is configured to extract the API key (i.e., the token) from the Authorization header of incoming requests. The authorizerUri is dynamically populated, linking to the specific Lambda function deployed within the same AWS environment. This function authenticates the token and determines whether the request should proceed.

Terraform Integration for Dynamic Configuration:

Given the dynamic nature of cloud environments where resource identifiers (like URIs and connection IDs) can vary between deployments, the use of Terraform templates was crucial. These templates allowed for the OpenAPI specification to be templated and parameterized, enabling the injection of environment-specific values at runtime. This approach not only streamlined the deployment process but also ensured that the API configurations remained flexible and adaptable to different deployment scenarios.

Chapter 5: Innovating AI at GTNAIVOLUTION Challenge

5.1 Introduction

The GTNAIVOLUTION Challenge, a flagship event at our company, provides a platform for innovation and collaboration among teams aiming to push the boundaries of artificial intelligence. This year, the competition was especially significant for me, as it presented an opportunity to demonstrate my skills and learn from seasoned professionals, despite being a fresher.

5.2 Team Formation and Initial Challenge

The challenge began with 25 diverse teams, each brimming with unique ideas. Our original team, shown in Figure 5.1, consisted of eight members who were passionate about developing a state-of-the-art AI chatbot.. During the initial pitch to the panel, it became evident that several teams shared a similar vision. To optimize resources and ideas, the panel recommended a consolidation of teams with like-minded proposals. Consequently, two of us from the original group, myself and my colleague Karan, were selected to join a new team that represented a merger of four teams. This restructuring resulted in a formidable group of eight members tasked with a significant project—developing the GTNWISEBOT.

Figure 5.1 : The initial team of eight members

5.3 Development and Role Assignment

In the newly formed team, roles were assigned with the aim of leveraging individual strengths. Despite our relative inexperience, Karan and I were entrusted with critical roles in developing and engineering the chatbot. Our responsibilities were substantial, as we were to build the core functionality of the bot. To aid us, three senior team members were assigned to oversee the dataset preparation, ensuring we had robust data to work with. This data spanned various formats including CSV, Excel, PDF, and TXT files from different internal teams such as App Support, DevOps, and Security. These documents were essential for training our bot and were processed into vector embeddings stored in a Chroma database.

5.4 Technological Implementation

The technological infrastructure behind GTNWISEBOT was meticulously designed to ensure robust performance, scalability, and user responsiveness. The implementation involved several key technologies and architectural strategies that are depicted in Figures 5.2, 5.3, and 5.4.

Frontend and User Interface

Our first major task was developing the frontend of the application, which serves as the interactive face of our chatbot. Utilizing Streamlit, we created an intuitive and user-friendly interface that allows users to interact with multiple bots. Each bot is designed to assist users based on their specific roles and inquiries, whether they are new joiners needing guidance on internal projects or public users seeking information about our platforms. The frontend of our working bot is illustrated in Figure 5.2.

LangChain Infrastructure

Central to our application's functionality was the integration of LangChain, which links conversational retrieval and question answering capabilities with our backend systems. This framework was essential for enabling the chatbot to access a diverse array of internal documents and data, process them, and deliver accurate, contextually relevant responses. LangChain facilitated the integration of memory capabilities and API calls to OpenAI's GPT-3.5, which significantly enhanced the chatbot's ability to understand and respond to complex queries. This infrastructure is detailed in Figure 5.3.

Data Management and Processing Pipeline

The data management and processing pipeline played a critical role in ensuring that our chatbot was trained on high-quality, relevant data. We utilized AWS S3 buckets to store a vast array of internal documents in various formats such as CSV, Excel, PDF, and TXT. AWS Lambda functions were triggered to automate the conversion of these documents into vector embeddings. These embeddings were then stored in a Chroma database, enabling efficient data retrieval and manipulation to support the AI’s learning process. This setup not only streamlined the data handling process but also ensured that updates to the data were seamlessly incorporated into the system. The complete data processing pipeline is shown in Figure 5.4.

Conversational Retrieval and QA

A critical feature of our chatbot was its ability to perform conversational retrieval and question answering, which was powered by the LangChain integration. This capability allowed the chatbot to understand the context of queries, search through the vector database for the most relevant information, and construct responses that were informative and contextually appropriate. The combination of vector embedding and LangChain’s QA capabilities ensured that our chatbot could handle a wide range of queries with high accuracy and relevance

.

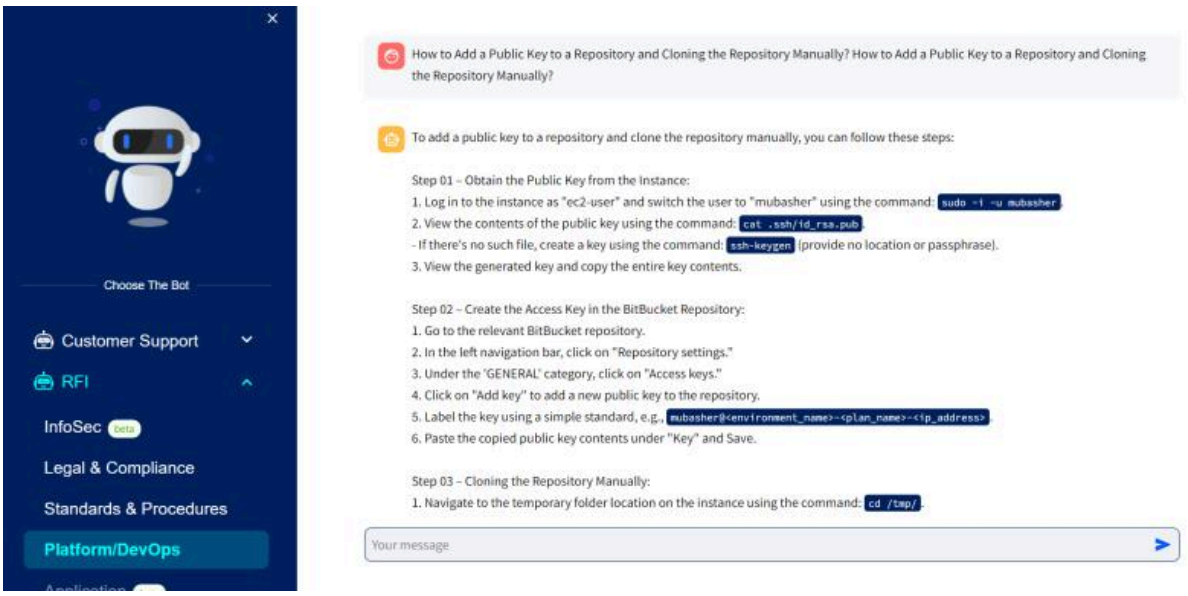


Figure 5.2 : Frontend Interface of GTNWISEBOT

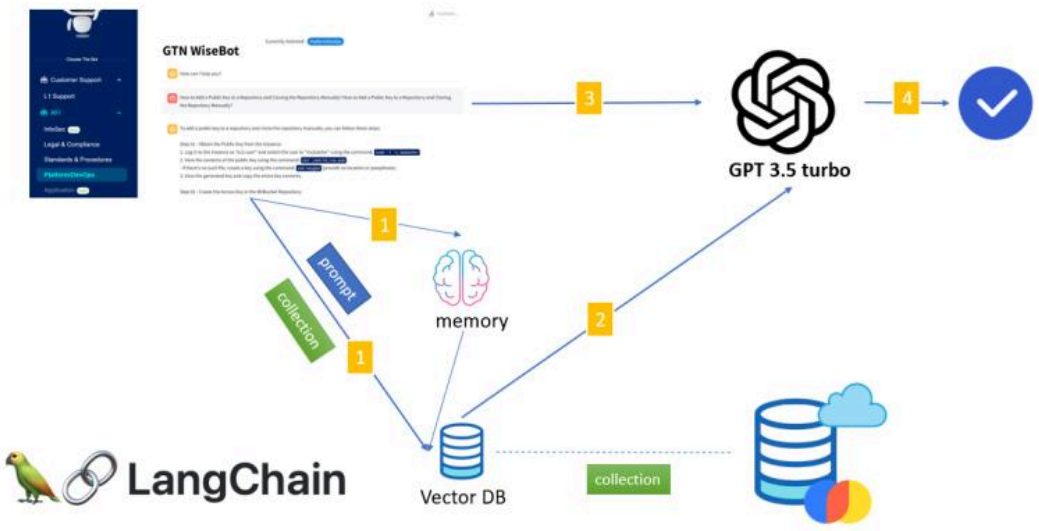


Figure 5.3: LangChain Infrastructure

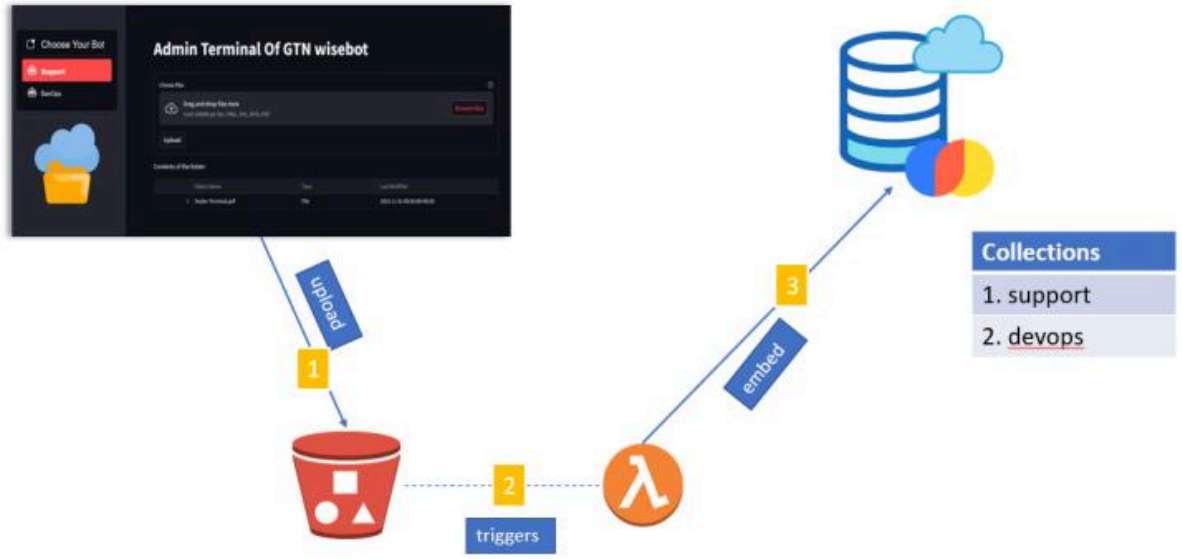


Figure 5.4 : Data Management and Processing Pipeline

5.5 The Grand Finale

The competition's final round was held at the Cinnamon Grand in Colombo, at the majestic Kings Court. It was a part of the annual general meeting, drawing an audience that included high-profile corporate members. Our presentation and live demonstration of GTNWISEBOT showcased its capability to intelligently handle queries and avoid irrelevant or out-of-context responses—a critical feature for a financial company like ours. The judges and audience were particularly impressed with the bot's discernment and the technical sophistication of our project.