Viet Nguyen

ORDER FULFILLMENT MANAGER

School of Technology
2022

ACKNOWLEDGMENTS

I would like to thank all my teachers at VAMK and especially Dr. Ghodrat Moghadampour – supervisor for my thesis. I could not finish the thesis paper without him.

Next, I want to thank Mr. Emiliano Spinella for his guidance during my time as an intern at Syndeno.

Finally, I want to thank Anh Minh, a VAMK student, for his encouragement in the process of thesis writing.

Viet Nguyen

Hanoi, Vietnam

25.11.2022

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Viet Nguyen
Title	Order Fulfillment Manager
Year	2022
Language	English
Pages	49
Name of Supervisor	Ghodrat Moghadampour

The main objective of the thesis was to develop a module that enables the user to install needed tools and software on a Kubernetes cluster through the data given by the customer.

Triggering pipeline and deploying resources is a tedious job that requires several processes. Therefore, in real-world production environment, there is need to build a microservice to operate and automate that work.

This application fully achieved its aim. Order Fulfilment Manager succeeds in deploying needed software on a Kubernetes cluster. It could receive orders, processes, and saves the output to the database. The thesis provides an insight of how to integrate different types of technologies into integration script and how to communicate between Jenkins and a cloud platform.

Keywords Python, Apache Kafka, Kubernetes, Cloud Platform, Jenkins

CONTENTS

1 INTRODUCTION 1
1.1 Objective 1
1.2 Syndeno 2
2 RELEVENT TECHNOLOGIES 3
2.1 Python 3
2.2 Google Cloud Platform 3
2.2.1 Google Kubernetes Engine 4
2.3 MongoDB 4
2.4 Jenkins 4
2.5 Terraform 5
2.6 Apache Kafka 6
2.7 Docker 7
2.8 Kubernetes 8
2.8.1 Basic Object in Kubernetes 8
2.8.2 Benefits of Kubernetes 8
2.9 Helm 9
3 APPLICATION DESCRIPTION 10
3.1 Objective and Function 12
3.2 Prerequisite 12
3.3 Requirements Analysis 12
3.3.1 Must-have Requirements 13
3.3.2 Should-have Requirements 13
3.3.3 Nice-to-have Requirements 13
3.4 Main Processes 13
3.4.1 Simulate Apache Kafka Order 15
3.4.2 Pre-validate order 16
3.4.3 Validate and Process order 18
3.4.4 Save Output 20
4 DATABASE 23
4.1 Entity Diagram 23

4.2 Collection 24
4.2.1 Servers Collection 24
4.2.2 Products Collection 24
4.2.3 Orders Collection 25
4.2.4 Builds Collection 26
5 IMPLEMENTATION 28
5.1 Setting up before Running the Application 28
5.1.1 Deploy GKE cluster 28
5.1.2 Get GCP access token 28
5.1.3 Jenkins Pipeline to Deploy Grafana 29
5.2 Order Fulfillment Manager 32
5.2.1 Simulate New Order 32
5.2.2 Order New Consumer 33
5.2.3 Order validated process 37
5.2.4 Order processing process 40
6 TESTING 42
6.1 Simulate New Order 42
6.2 Order New Consumer 42
6.3 Order Validated Process 43
6.4 Order Processing Process 44
7 CONCLUSIONS 47
7.1 Future work 47
REFERENCES 48

LIST OF FIGURES AND TABLES

Figure 1. OFM architecture diagram 10

Figure 2. Order state flow chart diagram 11

Figure 3. Use case diagram 14

Figure 4. Class diagram 15

Figure 5. Simulate Apache Kafka Order sequence diagram 16

Figure 6. Consumer class 16

Figure 7. Pre-validate order sequence diagram 17

Figure 8. OrderManagement class 17

Figure 9. Validate and Process sequence diagram 18

Figure 10. Class DAO 19

Figure 11. OrderProccess class 19

Figure 12. Save output sequence diagram 21

Figure 13. Jenkins class 21

Figure 14. Entity relationship diagram 23

Figure 15. Deploy Grafana structure 29

Figure 16. The simulate_new_OrderInstall file structure 32

Figure 17. File structure of order_NEW_consumer 34

Figure 18. File structure of order_VALIDATED_process 38

Figure 19. Simulate Apache Kafka order 42

Figure 20. The order_NEW_cosumer output 42

Figure 21. Orders collection 43

Figure 22. The order_VALIDATED_process output 43

Figure 23. Incomplete build data saved in Builds collection 44

Figure 24. State change to PROCESSING 44

Figure 25. The order_PROCESSING_process output 45

Figure 26. Build and stage status in database 45

Figure 27. State changes to COMPLETE 46

LIST OF CODE SNIPPETS

Code Snippet 1. Servers collection 24

Code Snippet 2. Products collection 25

Code Snippet 3. Orders collection 25

Code Snippet 4. Builds collection 26

Code Snippet 5. Get access token 29

Code Snippet 6. kubernetes.tf 30

Code Snippet 7. variables.tf 30

Code Snippet 8. helm_release.tf 31

Code Snippet 9. grafana-values.yaml 31

Code Snippet 10. versions.tf 32

Code Snippet 11. Initialize new Kafka producer 33

Code Snippet 12. Send message to consumer 33

Code Snippet 13. order_NEW_consumer.py 34

Code Snippet 14. Consuming the data 35

Code Snippet 15. Convert JSON to Python object 35

Code Snippet 16. Class Order 36

Code Snippet 17. Initialize order object 36

Code Snippet 18. preValidateOrder() 36

Code Snippet 19. saveOrder() 37

Code Snippet 20. order_VALIDATED_process.py 39

Code Snippet 21. validate_NEW_order method 39

Code Snippet 22. process_VALIDATED_order method 39

Code Snippet 23. trigger_jobs method 40

Code Snippet 24. Main function order_PROCESSING_process 40

Code Snippet 25. check_order_status() method 41

Code Snippet 26. Retry FAIL/FAIL_PARTIAL order 41

LIST OF ABBREVATIONS

OFM	Order Fulfillment Manager
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
OOP	Object Oriented Programming
SMB	Small and Midsize Business
NoSQL	Not Only SQL
JSON	JavaScript Object Notation
Amazon EC2	Amazon Elastic Compute
HTML	Hypertext Markup Language
API	Application Programming Interface
UNIX	Uniplexed Information Computing System
RESTAPI	Representational State Transfer
VM	Virtual Machine
YAML	Yet Another Markup Language
IP	Internet Protocol
GUI	Graphical User Interface
CI/CD	Continuous Integration/ Continuous Deployment

1 INTRODUCTION

Cloud computing is a term that has gained popularity in recent years. With the exponential increase in data use that has accompanied society's transition into the digital century, individuals and organizations are finding it increasingly difficult to keep all their critical information, programs, and systems up and running on in-house computer servers. Hence, the developer should utilize this technology work more effectively. /1/

In real-world production environments, team members must work together throughout the entire software development cycle of an application, from coding and testing to deployment and ongoing monitoring. For that reason, different tools and software were built to decrease the workload of the developer. However, the integration of several technologies is a tedious task and requires an uncounted number of steps. Luckily, automation is the key to solve that problem.

Therefore, there is a need to create microservice to automate it. Order Fulfilment Manager was built to adapt to the need.

1.1 Objective

The aim of the thesis is to create an application to install needed tools and software on a Kubernetes cluster.

This thesis consists of seven sections. The first part is the objective, background, and brief introduction to the company. The second part is discussion of the technologies that were used for the application. The third part contains the requirement, description of the project, and the requirement analysis. The fourth is about the database. The implementation is examined in part five while the sixth part is for testing and the outcomes. The conclusions is the last part of this thesis.

1.2 Syndeno

Syndeno is a software company established in the Demium incubator in Valencia in February 2021. Their mission is to make the latest and most innovative data connectivity accessible to as many users as. Syndeno was created to democratize the access to new technologies and help Small and Midsize Business (SMB) in their digitalization process. They help companies transform the way they do business in the context of a rapid growth in data and automatization. The main client of the company is a start-up. /2/

Syndeno has 7 products, Syndeno for Kubernetes, Syndeno for Jenkins, Syndeno for Grafana, Syndeno for Databases, Syndeno for Apache Kafka, Syndeno for Apache Flink. Customers can choose the product that meets their need.

Normally, the load on servers is very high when a company has a high number of users on the platform. The Syndeno team has developed a solution, Apache Kafka, to streamline the load. /2/

2 RELEVENT TECHNOLOGIES

Order Fulfillment Manager uses various tools for DevOps and cloud computing. This section describes each tools in detail and the reason why it was chosen.

2.1 Python

Python is a computer programming language that is frequently used to create websites and software, automate tasks, and analyze data. It can be used to create a variety of programs and is not specialized for any particular problem. This versatility, combined with its ease of use for beginners, has made it one of the most widely used programming languages today. /3/

It is also a multi-paradigm programming language, which means it supports different programming approach. Python also provides object-oriented programming (OOP), which allows to break the program into the bit-sized problems that can be solved easily. /3/

Python was chosen for this project because it provides libraries for integrating with Apache Kafka, Jenkins, and MongoDB. Moreover, the syntax is not complicated, and the code is maintainable.

2.2 Google Cloud Platform

Google cloud platform (GCP) is a medium that allows users to easily access Google's cloud systems and other computing services. The platform provides a wide range of services that can be used in various cloud computing sectors, such as storage and application development. Google Cloud Platform can be used freely according to users’ needs. /4/

2.2.1 Google Kubernetes Engine

Google Kubernetes Engine (GKE) is a managed environment that allows you to deploy, manage, and scale containerized applications using Google infrastructure. Multiple machines are grouped together to form a cluster in the GKE environment. Users can use Google Cloud Platform Console or the gcloud command line interface to interact with Google Kubernetes Engine. /5,6/

Software developers frequently use Google Kubernetes Engine to create and test new enterprise applications. Administrators also use containers to meet the scalability and performance demands of enterprise applications such as web servers. /6/

2.3 MongoDB

MongoDB is an open-source database that uses a document-oriented data model and a non-structured query language. It is one of the most powerful NoSQL systems and databases around, today. /7/

MongoDB uses collections and documents rather than using tables and rows, as in the traditional relational databases. Documents are made up of key-value pairs, similar to JavaScript Object Notation (JSON). Collections are equivalent to relational database tables in that they contain sets of documents and functions. /7/

MongoDB was chosen because the application has unstructured data and MongoDB has no storable data type limits.

2.4 Jenkins

Jenkins is an open-source Continuous Integration server written in Java for orchestrating a chain of actions to achieve the Continuous Integration process in an

automated fashion. Jenkins supports the complete development life cycle of software from building, testing, documenting the software, deploying, and other stages of the software development life cycle. /8/

Jenkins can be used to automate and speed up the software development process. It integrates all development life-cycle processes, including build, documentation, testing, packaging, staging, deployment, static analysis, and much more. It also provides plugin to achieve Continuous Integration. For example, Git, Maven 2 project, Amazon EC2, and HTML publisher. /8/

Jenkins was chosen because it is open source so company could get started regardless of the budgetary constraints. Furthermore, Jenkins provides plugin suitable with the application.

2.5 Terraform

Terraform is a tool that defines both cloud and on-premises resources in human-readable configuration files that users can version, reuse, and share. It uses a consistent workflow to provision and manage all infrastructure throughout its lifecycle. /9/

Terraform uses application programming interfaces to create and manage resources on cloud platforms and other services (APIs). It can work with almost any platform or service that has an API. /9/

The core Terraform workflow consists of three stages:

- Write: Define infrastructure in configuration files
- Plan: Review the changes Terraform will make to the infrastructure
- Apply: Terraform provisions infrastructure and update the state file /9/

Provisioning infrastructure across multiple clouds improves fault tolerance and enables more graceful recovery from cloud provider outages. Multi-cloud deployments, on the other hand, add complexity because each provider has its own interfaces, tools, and workflows. Terraform allows users to manage multiple providers and cross-cloud dependencies with the same workflow. This simplifies management and orchestration for multi-cloud infrastructures on a large scale. /10/

Terraform was chosen for the project because code can be used to manage and maintain resources. It allows to store the infrastructure status and track the changes in different components of the system.

2.6 Apache Kafka

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables the user to pass messages from one endpoint to another. It is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. /11/

A single Kafka Server is called a Broker. The user can have a Kafka Cluster created in multiple brokers inside it. The broker received messages from the Producer, and it stores on a local disk. It also caters to a fetch request which is coming from the Consumers and provides the messages which are already written to the underline local disk. Based on the hardware, one Kafka Broker can handle 1000s of partitions and millions of messages per second. One partition is assigned to multiple brokers the owner being a single partition. /11/

Kafka's publish and subscribe pipelines are low-latency, high-throughput, and fault-tolerant, and it can process event streams. Kafka is popular because a company benefits greatly from event-driven architecture. This is due to the massive

amount of data generated and consumed by numerous services (internet of things, machine learning, mobile, microservices). /11/

Apache Kafka was chosen because of the flexibility. Kafka can be used for the majority types of contents and add different types of producers and consumers to the system. Therefore, if the business grows, there is no need rewrite the whole architecture.

2.7 Docker

Docker is an open platform for app development, shipping, and running. Applications are separated from infrastructure through packaging and running (potentially multiple) applications in loosely isolated environments called a container so user can deliver software quickly. /12/

Docker operates on a client-server model. The Docker client communicates with the Docker daemon, which builds, runs, and distributes Docker containers. Docker client and daemon can coexist on the same machine, or the user can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate with one another via a REST API, UNIX sockets, or a network interface. /12/

The reason Docker is growing popular is because it enables more efficient use of system resources. The cost savings will vary depending on what applications and how resource-intensive they may be, but containers invariably work out as more efficient than VMs. /12/

Docker was chosen because it provides a consistent and isolated environment.

2.8 Kubernetes

Kubernetes is a container orchestration platform. It can manage the entire lifecycle of individual containers, spinning up and shutting down resources as needed. The orchestration platform will launch another container if a container shuts down unexpectedly. Furthermore, Kubernetes also provides a mechanism for applications to communicate with each other even as underlying individual containers are created and destroyed. /13/

2.8.1 Basic Object in Kubernetes

The Pod object is the fundamental building block in Kubernetes, consisting of one or more (tightly related) containers, a shared networking layer, and shared filesystem volumes. Pods, like containers, are intended to be ephemeral - there is no expectation that a specific, individual pod will last a long time. /13/

A Deployment object is made up of a collection of pods that are defined by a template and a replica count (how many copies of the template we want to run). The user can either specify a specific replica count or use a separate Kubernetes resource to control the replica count based on system metrics like CPU utilization. /13/

A Kubernetes Service provides a stable endpoint that can be used to direct traffic to the desired Pods even when the underlying Pods change as a result of updates, scaling, and failures. Services determine which Pods to send traffic to base on labels (key-value pairs) defined in the Pod metadata. /13/

2.8.2 Benefits of Kubernetes

Kubernetes can be used to scale, and quick scaling up/down depends on the workloads. This is especially true if it consists of multiple services and requires scaling up and down depending based on various workloads to move to the

cloud. factors. Compared to VMs, containers provide an easy way to scale application. Moreover, Kubernetes prepares workloads to move to the cloud. If the cloud cannot be built now, building on containers and Kubernetes may be a good way to prepare for a future cloud migration. /14/

2.9 Helm

Helm is widely known as "the package manager for Kubernetes". Helm's initial goal was to provide users with a better way to manage all Kubernetes YAML files created in Kubernetes projects. Helm Charts are used to solve this problem. Each chart is a collection of one or more Kubernetes manifests - a chart can have child and dependent charts. This means that when users run the install command for the top-level chart, Helm installs the entire project's dependency tree. /15/

Charts allow to version manifest files too, just like with Node.js or any other package. The user can install specific chart versions, which means keeping specific configurations for infrastructure in the form of code. /15/

Helm natively supports Kubernetes, so users do not need to write any complex syntax files or anything else to begin using Helm. /15/

3 APPLICATION DESCRIPTION

In this section, the objective and function of the application will be discussed in detail. After that, the project requirement is analysed and categorized. Finally, use-case diagram, class diagram, sequence diagram, architectural diagram will be presented and explained to show the functionalities of the thesis.

Figure 1 below describes the architecture of the application.

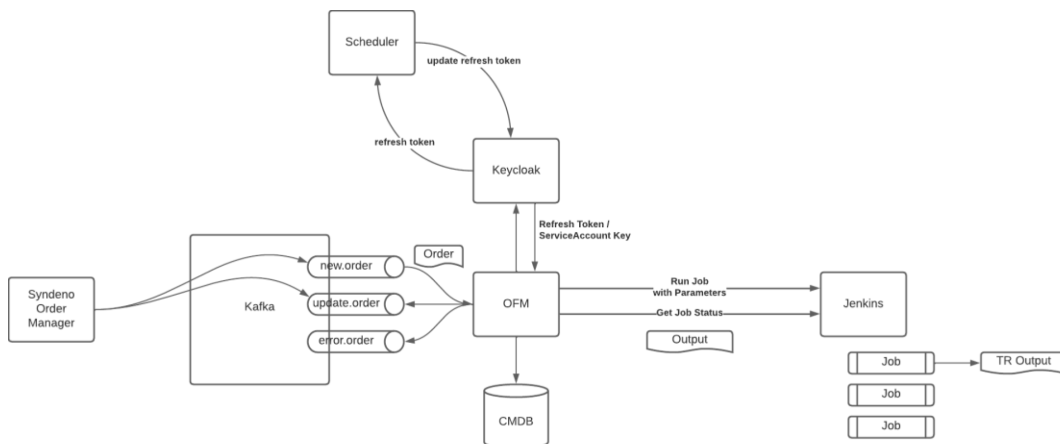

Figure 1. OFM architecture diagram

There are two main modules in the application: Syndeno Order Manager and Order Fulfillment Manager. Syndeno Order Manager was developed by my colleagues and its output is an Apache Kafka order, which will be consumed by OFM in the following step. Then, OFM saves the order in the database. After that, OFM communicates with Keycloak to get a refresh token to access GCP. Next, it will trigger Jenkins pipeline to deploy needed software by the data provides from the order as parameters. In the end, all the pipeline status and run time data is saved to the database.

Figure 2 below shows how the state of the Apache Kafka order changes from beginning till end.

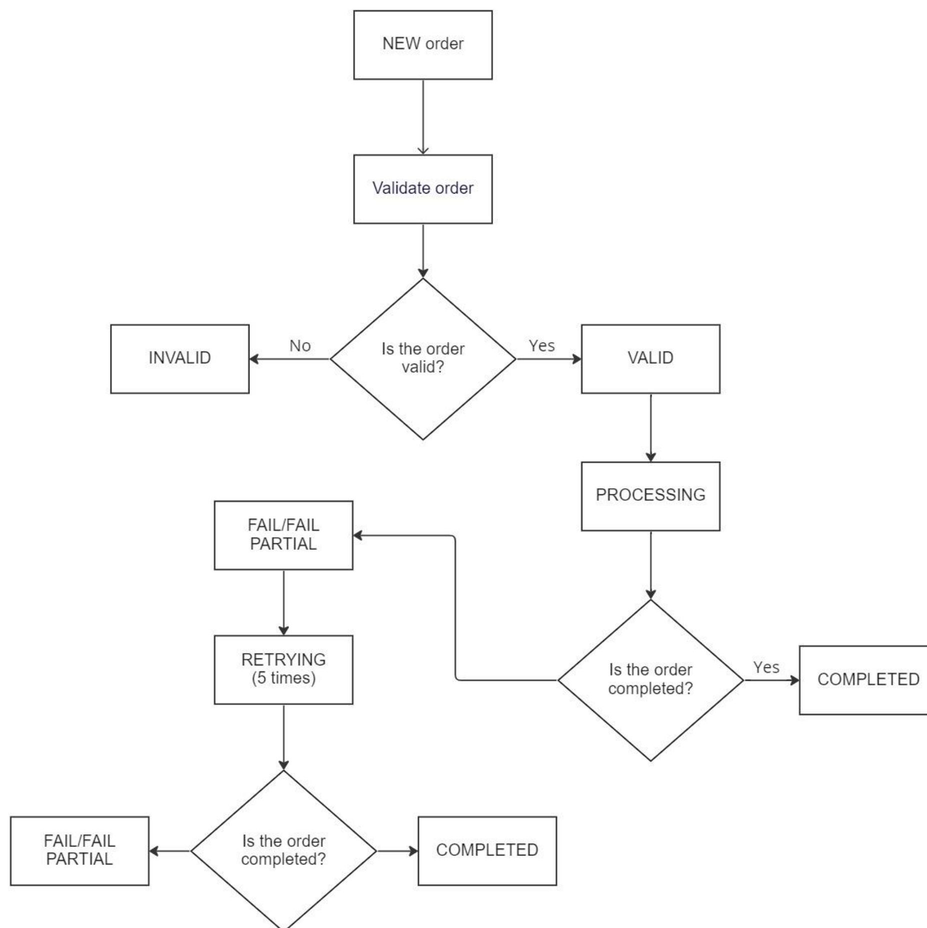

Figure 2. Order state flow chart diagram

When the order is pre-validated and saved to the database, it has the state "NEW". After executing the order_VALIDATED_consumer image, the order moves to "VALIDATING" state and decision "Is the order valid?". If it is no, the order has the "INVALID" state, and the program terminates. If it is yes, the order has the "VALID" state and move to "PROCESSING" state to start deploying application on a GKE cluster. Now, the user could execute order_PROCESSING_process image and the program encounters "Is the order completed?" logic. If the

application is deployed successfully on GKE cluster, the state “COMPLETED” is used and program ends. If not, the order will have “FAIL/FAIL PARTIAL” state and moves immediately to the “RETRY” state, where the Jenkins pipeline will be triggered again five times. Next, the program meets the logic “Is the order completed?” again and if nothing changes, the state would be set to “FAIL/FAIL PARTIAL” and the program terminates. However, if the application is deployed successfully, the state would be “COMPLETED” and the program ends.

3.1 Objective and Function

The main objectives of the project are to receive Apache Kafka order, process, and save its output to the MongoDB database. After receiving the order, the application will extract data (the name of application the user wants to deploy) and deploy the needed resources on a Kubernetes cluster on GCP. In the end, the build data after the Jenkins pipeline is done is saved to a collection in MongoDB.

3.2 Prerequisite

The following is required to run the application:

- GCP (Google Cloud Platform) account with a service account.
- Kubernetes cluster on GKE (Google Kubernetes Engine).
- Three Jenkins pipelines to deploy MySQL, MongoDB, Grafana on GKE cluster
- Bash script code to gain access to GCP through a service account

3.3 Requirements Analysis

The requirements can be divided into three categories: must-have requirements, should-have requirements, and nice-to-have requirements.

3.3.1 Must-have Requirements

The program must fulfil the following requirements:

- The resources are deployed successfully on the GKE cluster
- The build data is fully saved in the database
- The state of the Apache Kafka order in the database should change according to its status
- The application can pre-validate the Apache Kafka order
- The application can validate the Apache Kafka order and then trigger the right Jenkins pipeline

3.3.2 Should-have Requirements

The application should have the following requirements:

- Use a refresh token for authorization to GCP
- Config file for credentials of database and IP address

3.3.3 Nice-to-have Requirements

It is nice for the application to have the following requirements:

- Docker file for each process
- More Jenkins pipeline for different types of application
- GUI

3.4 Main Processes

The use case diagram, sequence diagram, class diagram will be used to show detail each process of the application.

Figure 3 summarizes the details of the project.

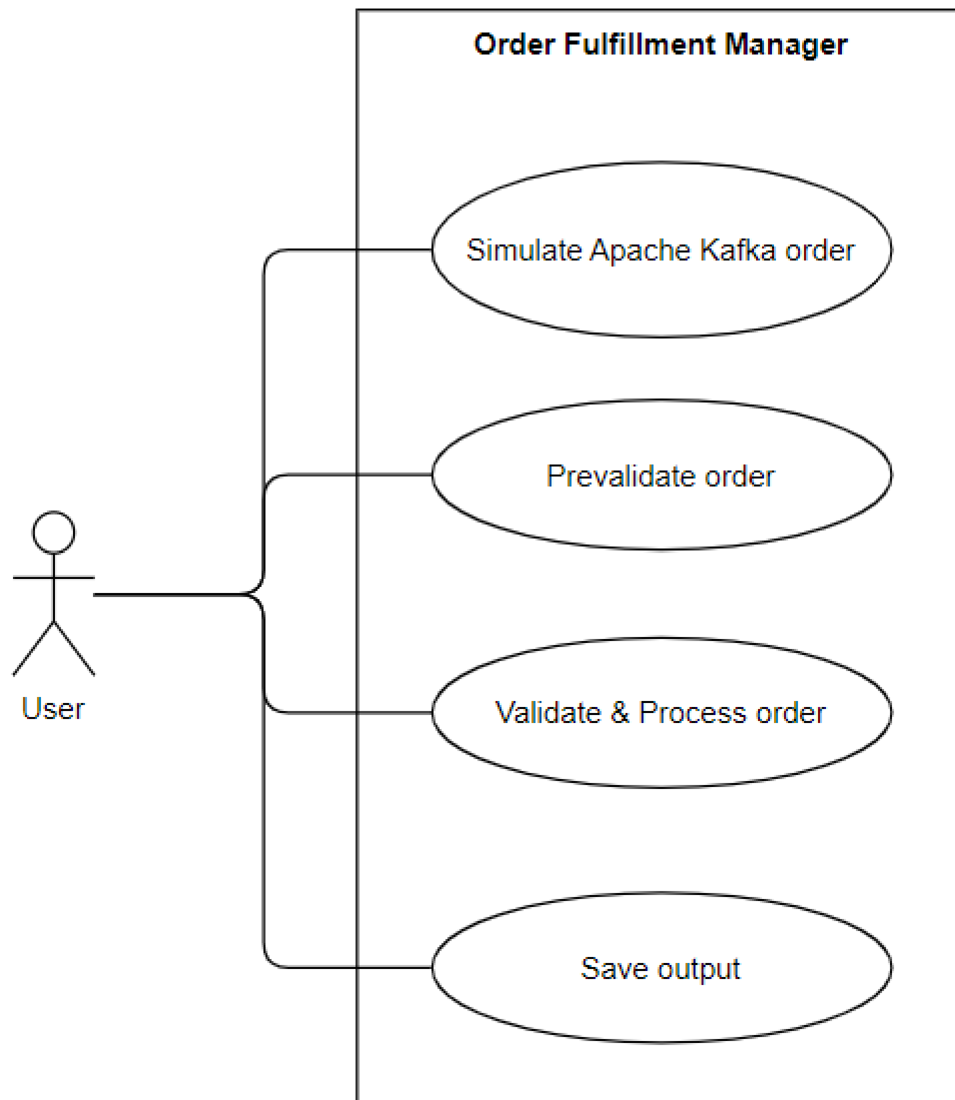

Figure 3. Use case diagram

The user is able to simulate an Apache Kafka order. After that, the order can be pre-validated to check if it meets the minimum requirement. Next, the user can validate and start processing the order to deploy resources on GKE cluster. Lastly, the user can display and save the output of build data.

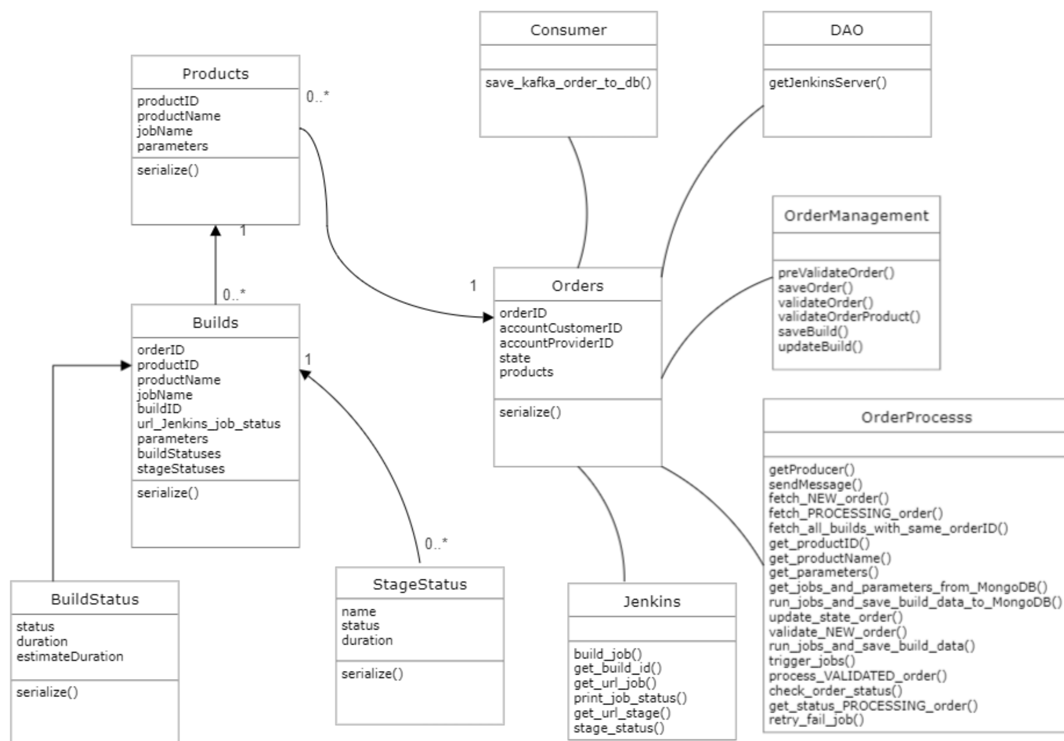

Figure 4. Class diagram

Figure 4 describes classes, methods, and properties which are involved in the application. Five classes (Orders, Products, Builds, BuildStatus, and StageStatus) contain variables which will be used in the database later. All other classes possess methods that help the project functions.

3.4.1 Simulate Apache Kafka Order

Figure 5 below shows the sequence diagram of Simulate Apache Kafka Order process.

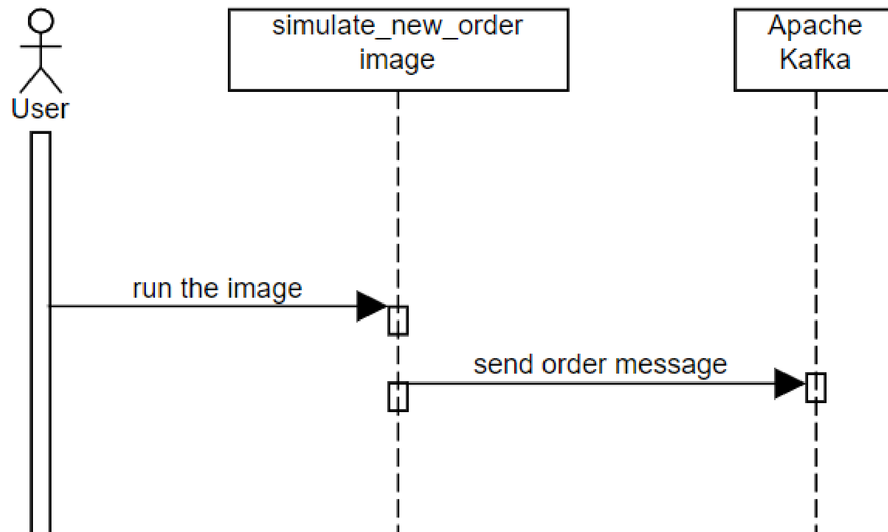

Figure 5. Simulate Apache Kafka Order sequence diagram

The user executes the simulate_new_order image (Figure 5) to simulate an Apache Kafka order.

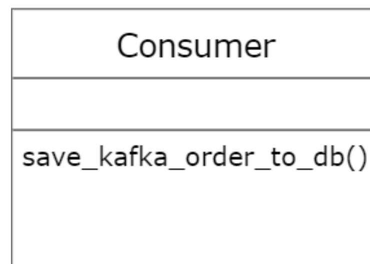

Figure 6. Consumer class

Consumer class (Figure 6) is used to save the Apache Kafka order to the database.

3.4.2 Pre-validate order

Figure 7 below shows the sequence diagram of Pre-validate order process.

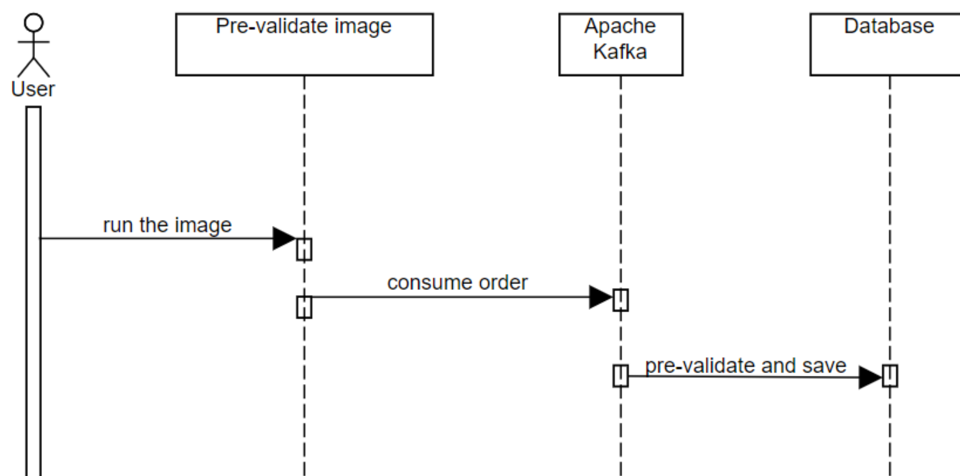

Figure 7. Pre-validate order sequence diagram

After running the Pre-validate image shown in Figure 7 (which names order_NEW_consumer in Docker) image, the application will pre-validate the order and save it to the database.

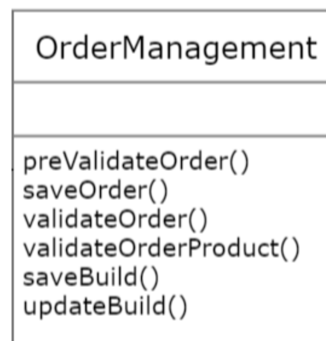

Figure 8. OrderManagement class

OrderManagement class (Figure 8) deals with order validation and save information to the database.

The method preValidateOrder() checks whether the order ID is filled or not. Meanwhile, validateOrder() and validateOrderProduct() investigate all the attributes of the order. If any attribute is missing, the order will change to status INVALID.

The method saveBuild() saves the build of order to the database. The last method of the class is updateBuild(), which is used to add the build data from Jenkins API to MongoDB.

3.4.3 Validate and Process order

Figure 9 below shows the sequence diagram of Validate and Process order process.

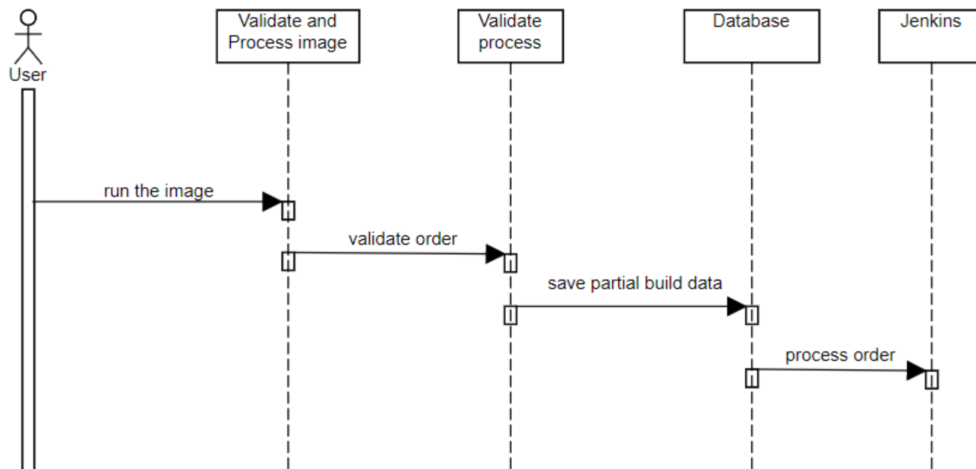

Figure 9. Validate and Process sequence diagram

The next process is Validate and Process order (the image was named order_VALIDATED_consumer). This will validate the order and save partial of build data to the database. After that, this image will trigger the needed Jenkins pipeline to deploy application on GKE cluster.

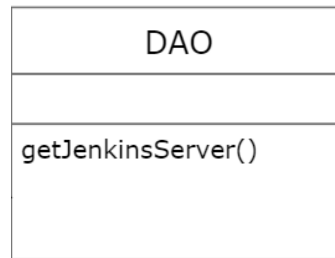

Figure 10. Class DAO

DAO will receive the credentials of Jenkins from the database (Figure 10).

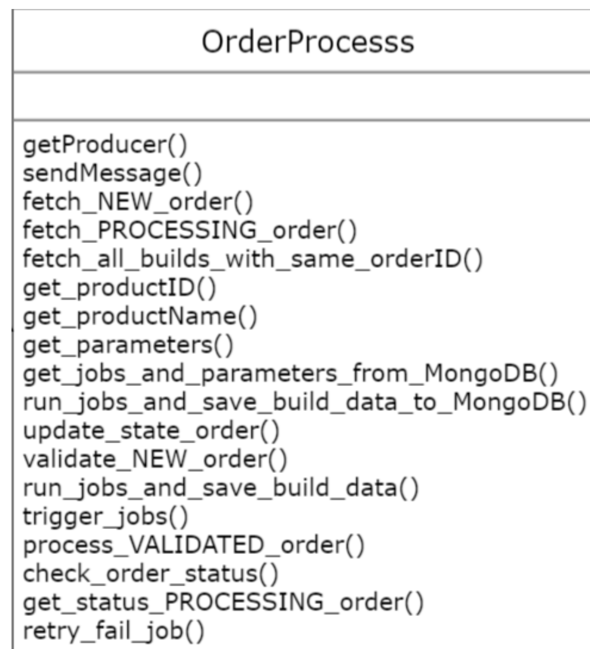

Figure 11. OrderProccess class

The OrderProcess class (Figure 11) has methods for validate order, process, and save output to the database.

The method validate_NEW_order() is called for validate the order with NEW state. Whereas, process_VALIDATED_order() is for the process order with VALIDATED state

The method update_state_order() would update the state of the order in the database. Meanwhile, trigger_jobs() authorizes to Jenkins server and trigger the pipeline to deploy resources on Kubernetes cluster. The retry_fail_job() method would rerun the Jenkins pipeline when the order has state FAIL/FAIL_PARTIAL.

The method getProducer() and sendMessage() receive and consume message from Apache Kafka.

The method fetch_NEW_order(), fetch_PROCESSING_order(), fetch_all_builds_with_same_orderID() fetch order with correspond state. After that, those data would be used for the following process of the application.

The method get_productID(), get_productName(), get_parameters(), get_jobs_and_parameters_from_MongoDB() get needed data from Products collection in MongoDB. These will be used for triggering Jenkins pipelines.

Method check_order_status() is used to define whether the order is FAIL, FAIL_PARTIAL, or SUCCESS.

Method get_status_PROCESSING_order() saves all the build data to the database.

Method run_jobs_and_save_build_data_to_MongoDB() would trigger the pipeline and save partial build data to the database.

3.4.4 Save Output

Figure 12 below shows the sequence diagram of Save Output process.

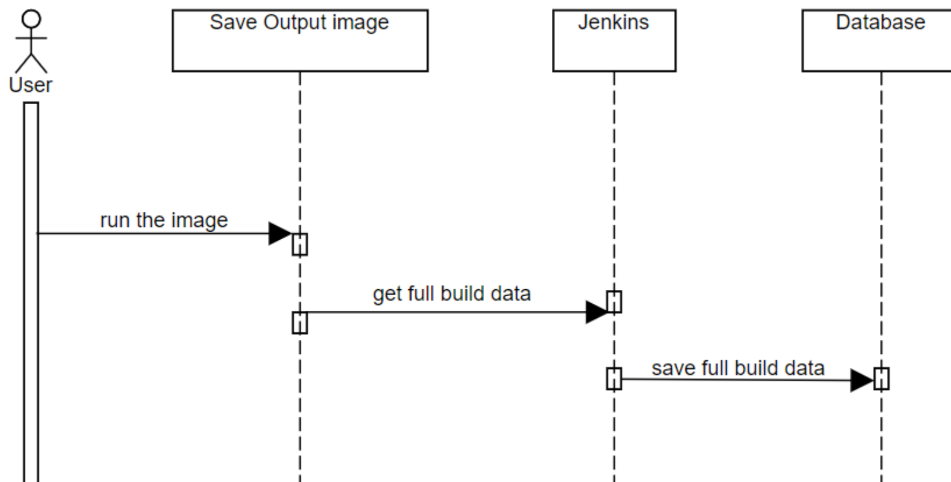

Figure 12. Save output sequence diagram

The last process is Save Output (image was named order_PROCESSING_consumer). It will get the build data from Jenkins API and save that to new collection in the database.

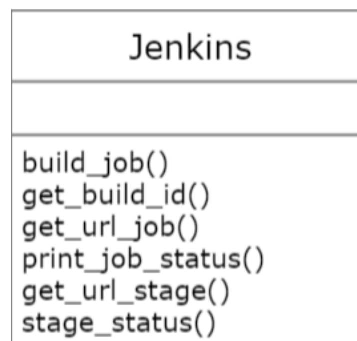

Figure 13. Jenkins class

Jenkins class will interact with Jenkins API to get the job status and stage status.

The method get_build_id(), get_url_job(), and get_url_stage() get the build id, the URL of the Jenkins pipeline API for job status and for stage status. Meanwhile, build_job() extracts the whole build data from Jenkins API and return that at the end of method. The print_job_status() method is used to print the status of the job in the terminal to test if it is right.

The method stage_status() could get useful information (name, status, duration) of each step of the pipeline.

4 DATABASE

MongoDB was chosen as the database for the project. There are four collections in the database: servers, products, orders, and builds. This section will discuss about the architecture and schema of the database.

4.1 Entity Diagram

Figure 14 below shows the entity relationship diagram.

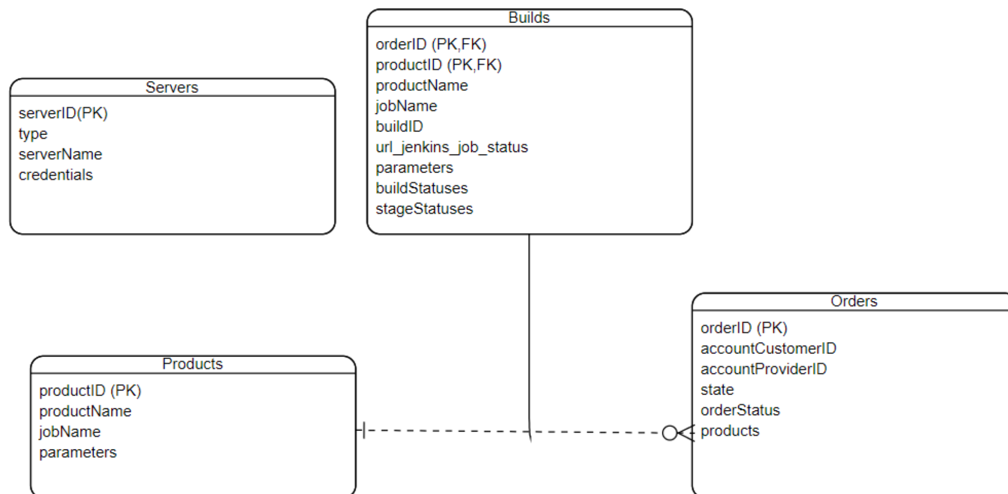

Figure 14. Entity relationship diagram

The Servers collection does not have any connection to other collections. The Products collection has productID as the primary key. Orders collection has orderID as primary key. For an order to exist, it must have at least one product and different product can be comprised of that order. Therefore, “one or many” was used to show relation between those order.

A product can be part of no orders, but it also can be a product of many orders. For that reason, “zero or many” notation was used in the diagram.

The “Builds” collection uses composite primary keys with orderID and productID.

4.2 Collection

There are four collections in the database: Servers collection, Products collection, Orders collection, and Builds collection. This section gives examples and explain the functionality of each collection.

4.2.1 Servers Collection

Code snippet 1 below shows an example of the Servers collection.

```
[{
	"type": "jenkins",
	"serverID": "jenkins-01",
	"serverName": "Your_Jenkins",
	"credentials": {
		"url": "http://localhost:8080/",
		"username": "admin",
		"password": "yourpassword",
		"token": "My-token"
	}
}]
```

Code Snippet 1. Servers collection

The Servers collection contains data about credentials for the Jenkins server. The key “type” contains data of which CI/CD tool the user wants to use (in this scenario is “Jenkins”). “Credentials” object has Jenkins URL, username, password, and token. Saving the credentials in the database is a good way to enhance security instead of writing that in the code.

4.2.2 Products Collection

Code snippet 2 below shows an example of the Products collection.

```
[{
	"productID": "P4",
```

```
        "productName": "grafana",
        "jobName": "deploy-grafana",
        "parameters": [{
                "name": "NAME_TF_WORKSPACE",
                "defaultValue": "deploy-grafana"
        }]
}]
```

Code Snippet 2. Products collection

The Products collection carries data about the product the user wants to deploy on a GKE cluster. Each product has a unique “productID”. The “jobname” key is the name of the Jenkins pipeline. The “Parameters” document value will be used as a parameter in the Jenkins pipeline.

4.2.3 Orders Collection

Code snippet 3 below shows an example of the Orders collection.

```
[{
        "orderID": "order-2",
        "accountCustomerID": "123",
        "accountProviderID": "999",
        "state": "COMPLETE",
        "orderStatus": "",
        "products": [{
                "productID": "P4",
                "productName": "grafana",
                "providerType": "GCP"
        }]
}]
```

Code Snippet 3. Orders collection

The Orders collection contains details about the order and the state of that order. The state will change according to the flowchart diagram. More details about the product is also showed in the “products” document.

4.2.4 Builds Collection

Code snippet 4 below shows an example of the Builds collection.

```
[{
	"orderID": "order-1",
	"productID": "P4",
	"productName": "grafana",
	"jobName": "deploy-grafana",
	"buildID": "510",
	"url_jenkins_job_status": "http://localhost:8080/job/de-
ploygrafana/510/api/json",
	"parameters": [{
		"name": "NAME_TF_WORKSPACE",
		"defaultValue": "deploy-grafana"
	}],
	"buildStatuses": [{
		"status": "FAIL",
		"duration": "8.409s",
		"estimateDuration": "42.864s"
	}],
	"stageStatuses": [{
		"name": "Declarative: Checkout SCM",
		"status": "SUCCESS",
		"duration": "568ms"
	}]
}]
```

Code Snippet 4. Builds collection

The Builds collection will carry details about the build data after the pipelines have been triggered. It shows the orderID, productID, productName, jobName, buildID, and an URL that uses to get the data from Jenkins API. The "buildStatuses" document shows the status of the pipeline, the duration and estimateDuration of pipeline. The "stageStatuses" document shows all the steps, as

well as the name, status, and duration of each step. All that data is taken from the Jenkins API.

5 IMPLEMENTATION

In this section, the implementation of the software will be described. There are several steps to set up before running the application. After that, the functionality and code snippet of Order Fulfillment Manager is discussed.

5.1 Setting up before Running the Application

This section will describe prerequisite before executing the project. There are three steps: Deploy GKE cluster, bash script to get GCP access token, and Jenkins pipeline to deploy application.

5.1.1 Deploy GKE cluster

A GKE cluster is prerequisite for Order Fulfillment Manager. Instructions found on the reference number 16 were used to create a GKE cluster. /16/

5.1.2 Get GCP access token

GCP requires authorization to be able to access and use resources. Therefore, we need to find a way to access GCP through the service account.

```
#!/bin/bash
key_json_file="syndeno-sandbox-dee00834aba3.json"
scope="https://www.googleapis.com/auth/cloud-platform"
jwt_token=$(./jwttoken.sh "$key_json_file" "$scope")
GCP_ACCESS_TOKEN=$(curl -s -X POST https://www.goog-
leapis.com/oauth2/v4/token \
    --data-urlencode 'grant_type=urn:ietf:params:oauth:grant-type:jwt-
bearer' \
    --data-urlencode "assertion=$jwt_token" |
```

```
    jq -r .access_token)
```

Code Snippet 5. Get access token

The code shown in Code Snippet 5 use the service's account key JSON file to get an access token to call Google APIs. We will save the access token to the GCP_ACCESS_TOKEN variable and later use it in the Jenkins pipeline.

5.1.3 Jenkins Pipeline to Deploy Grafana

Below is the structure of the folder that contains the code for deploying Grafana on the GKE cluster.

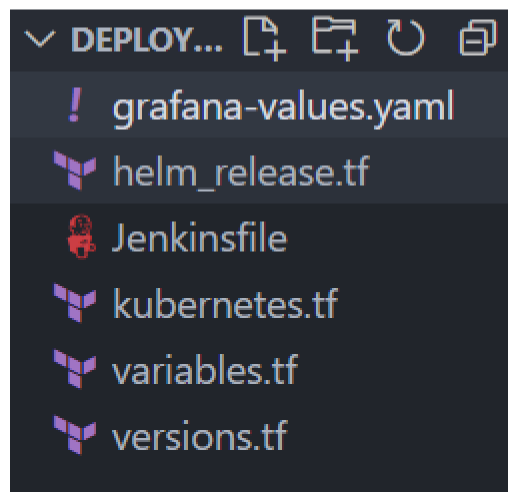

Figure 15. Deploy Grafana structure

The folder has six files kubernetes.tf, variables.tf, versions.tf, variables.tf, helm_release.tf, Grafana-values.yaml, and Jenkinsfile.

```
provider "google" {
  project      = "syndeno"
  region       = var.region
  access_token = var.access_token
}
data "google_client_config" "provider" {}
data "google_container_cluster" "my_cluster" {
```

```
  name     = "syndeno"
  location = "europe-west4"
}
provider "kubernetes" {
  host  = "https://${data.google_container_cluster.my_cluster.endpoint}"
  token = data.google_client_config.provider.access_token
  cluster_ca_certificate = base64decode(
    data.google_container_cluster.my_cluster.master_auth[0].clus-
ter_ca_certificate,
  )
}
```

Code Snippet 6. kubernetes.tf

The provider "google" is used to configure the credentials to authenticate with GCP. The provider "kubernetes" provides credentials to connect to the Kubernetes cluster.

```
variable "region" {
  default = "europe-west4"
}

variable "access_token" {
  type = string
}
```

Code Snippet 7. variables.tf

The file variables.tf contains variables to use in kubernetes.tf. Having a variable file is a good way to avoid repetitiveness in the code.

```
provider "helm" {
  kubernetes {
    host  = "https://${data.google_container_cluster.my_cluster.end-
point}"
    token = data.google_client_config.provider.access_token
    cluster_ca_certificate = base64decode(
      data.google_container_cluster.my_cluster.master_auth[0].clus-
ter_ca_certificate,
```

```
    )
  }
}

resource "helm_release" "grafana" {
  name       = "my-grafana-release"
  repository = "https://charts.bitnami.com/bitnami"
  chart      = "grafana"
  namespace = "viet"

  values = [
    file("${path.module}/grafana-values.yaml")
  ]
}
```

Code Snippet 8. helm_release.tf

We can release a Helm chart and customer it with Terraform.

```
fullnameOverride: "viet-grafana"
```

Code Snippet 9. grafana-values.yaml

The YAML file is used to override the default settings.

```
terraform {
  required_providers {
    helm = {
      source  = "hashicorp/helm"
      version = "~> 2.0.1"
    }
    google = {
      source  = "hashicorp/google"
      version = ">=3.52.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ">= 2.0.0"
    }
```

```
  }
  backend "gcs" {
    bucket = "tf-viet-testing"
    prefix = "tfstate-kubernetes"
  }
}
```

Code Snippet 10. versions.tf

Code snippet 10 provides the version that we need for the tools.

5.2 Order Fulfillment Manager

Order Fulfillment Manager is divided to four parts: simulate new order, order new consumer, order validated process, order processing process. The functionality, file structure, and code snippet are described below to give further details of the thesis.

5.2.1 Simulate New Order

The Simulate_new_OrderInstall image simulates an Apache Kafka order. This order will be used in other process of the application.

Figure 16 below shows the file structure.

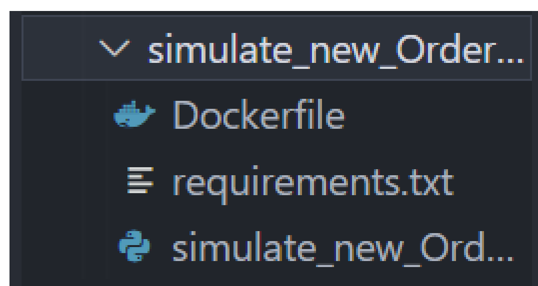

Figure 16. The simulate_new_OrderInstall file structure

There is one python file simulate_new_OrderInstall.py which contains the code. The Dockerfile and requirements.txt file is used to create docker image.

```
producer = KafkaProducer(bootstrap_servers=['172.17.0.1:9092'],
                            client_id='producer',
                              value_serializer=lambda x: dumps(x).en-
code('utf8'),
                              api_version=(0, 10, 1))
```

Code Snippet 11. Initialize new Kafka producer

Code Snippet 11 will initialize a new Kafka producer. The “bootstrap_servers” sets the host and port the producer should contact to bootstrap initial cluster metadata. “value_serializer” tells how the data should be serialized before sending to the broker. Here, we convert the data to json file and encode it to utf-8.

```
data = {
#the example of Apache Kafka order will be inserted here
}
producer.send('new_order_install', key=b'1003', value=data)
sleep(1)
```

Code Snippet 12. Send message to consumer

Code Snippet 12 will send the “Apache Kafka order” (save in “data” variable) message to the topic called “new_order_install”. A Kafka consumer will fetch the same message from the same topic name.

5.2.2 Order New Consumer

The order_NEW_consumer image will receive and pre-validate the Apache Kafka order that we simulate in simulate_new_OrderInstall. If the pre-validation is successful, the order is saved with state “NEW” to the database.

Figure 17 shows the file structure of order_NEW_consumer.

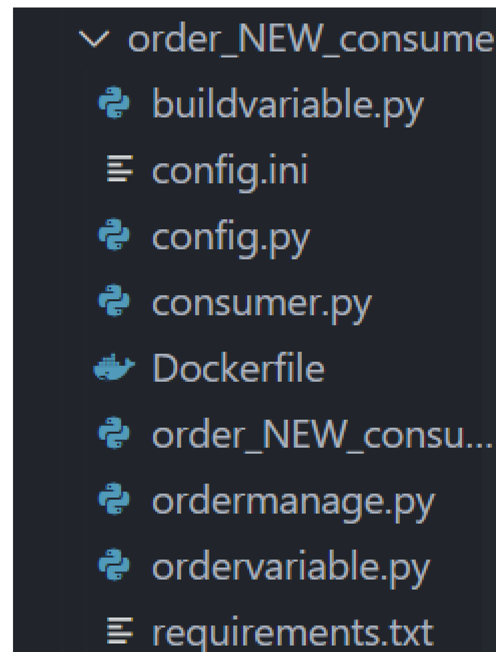

Figure 17. File structure of order_NEW_consumer

There are a total of nine files in the folder “order_NEW_consumer”.

```
if __name__ == "__main__":
    kafka = Consumer()
    Consumer.save_kafka_order_to_db()
```

Code Snippet 13. order_NEW_consumer.py

Code Snippet 13 shows the main program of order_NEW_consumer process.

```
class Consumer:
    def save_kafka_order_to_db():
        consumer = KafkaConsumer(
            'new_order_install',
            bootstrap_servers=['localhost:9092'],
            auto_offset_reset='earliest',
            enable_auto_commit=True,
            group_id='consumer',
            client_id='pythonOFM',
            value_deserializer=lambda x: x.decode('utf-8'),
```

```
            api_version=(0, 10, 2))
```

Code Snippet 14. Consuming the data

Code Snippet 14 will consume the data that we send in Code Snippet 12. “auto_offset_reset” handles where the consumer restarts reading after breaking down or being turned off. When set to “earliest”, the consumer starts reading at the latest committed offset.

```
for message in consumer:
        message = message.value
        orderOrg = json.loads(
                message, object_hook=lambda d: SimpleNamespace(**d))
```

Code Snippet 15. Convert JSON to Python object

Code Snippet 15 will convert JSON data to Python object so we can handle data easily in the future.

```
class Order:
    def __init__(self):
        self.orderID = ""
        self.accountCustomerID = ""
        self.accountProviderID = ""
        self.state = ""
        self.orderStatus = ""
        self.products = []

    def serialize(self):
        obj = {}
        obj['orderID'] = self.orderID
        obj['accountCustomerID'] = self.accountCustomerID
        obj['accountProviderID'] = self.accountProviderID
        obj['state'] = self.state
        obj['orderStatus'] = self.orderStatus
        obj['products'] = []
        for product in self.products:
            obj['products'].append(product.serialize())
```

```
        return obj
```

Code Snippet 16. Class Order

Code Snippet 16 shows constructor and serialize().

```
order = Order()
order.orderID = orderOrg.orderID
order.accountCustomerID = orderOrg.accountCustomerID
order.accountProviderID = orderOrg.accountProviderID
order.state = orderOrg.state
order.orderStatus = orderOrg.orderStatus
```

Code Snippet 17. Initialize order object

Code Snippet 17 initializes order object from Order class and assign attributes.

```
class OrderManagement:
    def preValidateOrder(self, order):
        if order.orderID == "":
            print("Missing order ID! Please enter it.")
            order.state = OrderStateEnum.INVALID
            return False
        else:
            print('Prevalidate is successful.\nYour OrderID is "' +
                  order.orderID + '".')
            return True
```

Code Snippet 18. preValidateOrder()

Code Snippet 18 shows preValidateOrder method. This will check if the order ID is filled or not. If the order ID is missing, the state will be set to INVALID and return False. If the order ID is filled, the program will return True.

```
Class OrderManagement:
    def saveOrder(self, order):
        client = pymongo.MongoClient(CONNECTION_STRING)
        db = client['SyndenoDB']
        orders_collection = db['orders']
        if self.preValidateOrder(order) == True:
```

```
            order['state'] = OrderStateEnum.NEW
            orders_collection.insert_one(order.serialize())
            print('The order "' + order.orderID +
                  '" has been saved to MongoDB!')
        else:
            print("Prevalidation is failed. Please fill in the order
ID!")
```

Code Snippet 19. saveOrder()

Code Snippet 19 will check if preValidateOrder true or not. If it is true, the order will have state NEW, and it is saved to the “orders” collection in database.

5.2.3 Order validated process

The order_VALIDATED_process image will check if the order is valid and then trigger the needed pipeline to deploy application on GKE cluster.

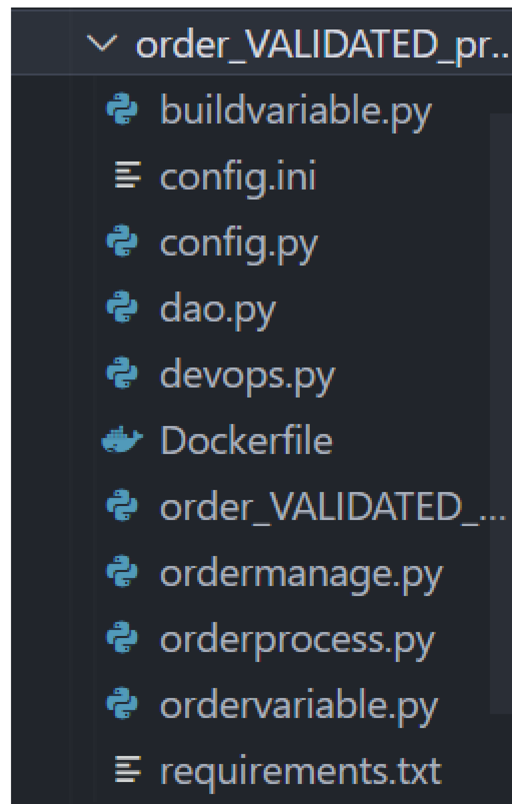

Figure 18. File structure of order_VALIDATED_process

There are eleven files in this process.

```
if __name__ == "__main__":
    process = OrderProcess()
    while(True):
        all_NEW_orders = process.fetch_NEW_order()
        print("Starting NEW order processing")
        for order in all_NEW_orders:
            validate = process.validate_NEW_order(order)
            if(validate):
                process.process_VALIDATED_order(order)
        print("Sleeping until next iteration\n")
```

```
        sleep(3)
```

Code Snippet 20. order_VALIDATED_process.py

Order_VALIDATED_process is the main Python file of the process. It will fetch all NEW orders in database, validate and trigger the needed Jenkins pipeline.

```
class OrderProcess:
    def validate_NEW_order(self, order):
        self.notify_VALIDATING_order(order)
        o = OrderManagement()
        self.order_state = o.validateOrder(order["orderID"])
        if self.order_state == OrderStateEnum.VALIDATED:
            self.notify_VALIDATED_order(order)
        elif self.order_state == OrderStateEnum.INVALID:
            self.notify_INVALID_order(order)
        return self.order_state
```

Code Snippet 21. validate_NEW_order method

Code Snippet 21 describes how the validate_NEW_order method works. It will use validateOrder method, which checks if all attributes of the order is filled. If that is true, the order will have VALIDATED state. If that is not the case, the order will have INVALID state and the program terminates.

```
class OrderProcess:
    def process_VALIDATED_order(self, order):
        if self.order_state == OrderStateEnum.VALIDATED:
            self.notify_PROCESSING_order(order)
            self.trigger_jobs(order)
```

Code Snippet 22. process_VALIDATED_order method

Code snippet 22 checks if the order has VALIDATED state or not. If it is true, the state will change to PROCESSING and then trigger Jenkins pipeline.

```
class OrderProcess:
    def trigger_jobs(self, order):
        db = DAO()
```

```
        jenkins_server = db.getJenkinsServer("jenkins-01")
        jenkins_obj = Jenkins(jenkins_server)
        jobs = jenkins_obj.get_job_and_parameters_from_MongoDB(order)
        for job_name in jobs.keys():
            parameters = jobs[job_name]
            self.run_jobs_and_save_build_data_to_MongoDB(
                job_name, parameters, jenkins_obj, jenkins_server, or-
der)
        print("All jobs has been triggered!")
```

Code Snippet 23. trigger_jobs method

Code Snippet 23 will get the credentials to Jenkins server from database. After that, it will get the job name from MongoDB and use it as parameter for run_jobs_and_save_build_data_to_MongoDB method. The pipeline will be triggered after that method.

5.2.4 Order processing process

The order_PROCESSING_process image will get the build status and stage status after the pipeline finished and save them to “builds” collection in MongoDB.

```
if __name__ == "__main__":
      process = OrderProcess()
      while(True):
      print("Starting PROCESSING order processing")
      all_PROCESSING_orders = process.fetch_PROCESSING_order()
      for order in all_PROCESSING_orders:
      process.get_status_PROCESSING_order(order)
      print("Sleeping until next iteration\n")
      sleep(3)
```

Code Snippet 24. Main function order_PROCESSING_process

Code Snippet 24 will fetch all PROCESSING orders in database and get the build data of each order.

```
class OrderProcess(Jenkins):
```

```
def check_order_status(self, order, all_buildstatus):
    if 'FAIL' not in all_buildstatus:
        self.notify_COMPLETE_order(order)
    elif 'SUCCESS' not in all_buildstatus:
        self.notify_FAIL_order(order)
    elif 'SUCCESS' in all_buildstatus and 'FAIL' in all_buildstatus:
        self.notify_FAIL_PARTIAL_order(order)
```

Code Snippet 25. check_order_status() method

Code Snippet 25 will check if any order has SUCCESS or FAIL. If none, the order will get FAIL_PARTIAL state. The state will decide next step of the application in the flow chart.

```
self.check_order_status(order, all_buildstatus)
if self.order_state == OrderStateEnum.FAIL or self.order_state == Order-
StateEnum.FAIL_PARTIAL:
for job_name in all_fail_job_name:
self.retry_fail_job(job_name, jenkins_obj, jenkins_server,order)
```

Code Snippet 26. Retry FAIL/FAIL_PARTIAL order

After checking status of order, if it has FAIL or FAIL_PARTIAL state, the program will rerun the pipeline with parameters.

6 TESTING

This section will show the output of the project and the testing. The result, change of order state, and collection in database of each process is described additional details.

6.1 Simulate New Order

The image below describes the output when we run simulate_new_OrderInstall process.

```
viego@vostro:~/Desktop/syndeno/Python/misc$ python3 simulate_new_OrderInstall.py
sending: key=b'1000' value={'orderID': 'order-7', 'accountCustomerID': '123', 'accountProviderID': '222', 'state': 'NEW', 'produ
cts': [{'productOfferingVersionID': '111', 'productOfferingID': 'PO-64323', 'parentProductOfferingID': '222', 'productOfferingTy
pe': 'LEAF', 'productID': 'P-456', 'productName': 'grafana', 'externalProviderIP': 'EP-00001', 'externalProviderLocation': 'Madr
id', 'providerType': 'GCP', 'externalProviderKeyCloakUser': '7676dcd5-03ce-446e-8045-7ba0dbf6e558', 'features': [{'featureID': '
55', 'featureName': 'Security', 'quantity': '2'}, {'featureID': 'PF-22320001', 'featureName': 'XXX', 'quantity': '3'}]}, {'produ
ctOfferingVersionID': '123', 'productOfferingID': 'PO-64323', 'parentProductOfferingID': '222', 'productOfferingType': 'LEAF', '
productID': 'P-2', 'productName': 'mysql', 'externalProviderIP': 'EP-00001', 'externalProviderLocation': 'Barcelona', 'providerT
ype': 'AWS', 'externalProviderKeyCloakUser': '7676dcd5-03ce-446e-8045-7ba0dbf6e558', 'features': [{'featureID': '99', 'featureNa
me': 'Security', 'quantity': '2'}, {'featureID': 'PF-22320001', 'featureName': 'XXX', 'quantity': '3'}]}], 'orderStatus': ''}
```

Figure 19. Simulate Apache Kafka order

The order could be seen through "value".

6.2 Order New Consumer

The output of order_NEW_consumer will be described below.

```
viego@vostro:~/Desktop/syndeno/Python$ python3 order_NEW_consumer.py
Prevalidate is successful.
Your OrderID is "order-7".
The order "order-7" has been saved to MongoDB!
```

Figure 20. The order_NEW_cosumer output

Figure 20 shows the output of the order_NEW_consumer. It pre-validates the order, displays the value of OrderID in the terminal and saves the order to database.

```
_id: ObjectId("61dec5de3f9fe32b6b6bf6c6")
orderID: "order-7"
accountCustomerID: "123"
accountProviderID: "999"
state: "NEW"
orderStatus: ""
products: Array
  0: Object
    productOfferingVer…: "111"
    productOfferingID: "PO-64323"
    parentProductOffer…: "222"
    productOfferingType: "LEAF"
    productID: "P4"
    productName: "grafana"
    externalProviderIP: "EP-00001"
    externalProviderLo…: "Madrid"
    providerType: "GCP"
    externalProviderKe…: "7676dcd5-03ce-446e-8045-7ba0dbf6e558"
    features: Array
      0: Object
        featureID: "PF-000001"
        featureName: "Security"
        quantity: "2"
      1: Object
        featureID: "PF-22320001"
        featureName: "XXX"
        quantity: "3"
```


Figure 21. Orders collection

The order was saved to Orders collection with state NEW.

6.3 Order Validated Process

The output of order_VALIDATED_process will be described below.

```
viego@vostro:~/Desktop/syndeno/Python$ python3 order_VALIDATED_process.py
Starting NEW order processing
sending to topic order.change: key=None value={'orderID': 'order-7', 'state': 'VALIDATING'}
The order "order-7" has been validated
sending to topic order.change: key=None value={'orderID': 'order-7', 'state': 'VALIDATING'}
sending to topic order.change: key=None value={'orderID': 'order-7', 'state': 'PROCESSING'}
Triggering job:deploy-grafana
Jenkins Build URL: http://localhost:8080/job/deploy-grafana/500/
Build ID: 500
The build of order "order-7" has been saved to MongoDB!

-----------------------------------

All jobs has been triggered!
Sleeping until next iteration

Starting NEW order processing
Sleeping until next iteration
```

Figure 22. The order_VALIDATED_process output

The state of the order changes: NEW -> VALIDATING -> VALIDATED -> PROCESSING. The program will trigger deploy-grafana pipeline. After that, the build of order is saved to database.

```
_id: ObjectId("620a8d756bb92c34ff8293ee")
orderID: "order-7"
productID: "P4"
productName: "grafana"
jobName: "deploy-grafana"
buildID: "500"
url_jenkins_job_st…: "http://localhost:8080/job/deploy-grafana/500/api/json"
⌄parameters: Array
  ⌄0: Object
      name: "NAME_TF_WORKSPACE"
      defaultValue: "deploy-grafana"
  ⌄1: Object
      name: "token"
      defaultValue: "My-token"
⌄buildStatuses: Array
⌄stageStatuses: Array
⌄terraformOutputs: Array
```

Figure 23. Incomplete build data saved in Builds collection

The incomplete build data is saved in Builds collection in MongoDB. The build status and stage status are empty now, but we will save those data in next process.

```
_id: ObjectId("61dec5de3f9fe32b6b6bf6c6")
orderID: "order-7"
accountCustomerID: "123"
accountProviderID: "999"
state: "PROCESSING"
orderStatus: ""
› products: Array
```

Figure 24. State change to PROCESSING

The state of the order changes to PROCESSING.

6.4 Order Processing Process

The output of order_PROCESSING_process will be described in the figure below.

```
viego@vostro:~/Desktop/syndeno/Python$ python3 order_PROCESSING_process.py
Starting PROCESSING order processing
BUILD STATUS:
Duration: 125.29s
Estimated duration: 103.935s
Job status: SUCCESS

STAGE STATUS:
Declarative: Checkout SCM, SUCCESS, 688
Declarative: Tool Install, SUCCESS, 107
Setup parameters, SUCCESS, 375
Get GCP access token, SUCCESS, 733
Git Checkout, SUCCESS, 686
Terraform Init, SUCCESS, 4596
Terraform Apply, SUCCESS, 113796

The job deploy-grafana is success
---------------------
sending to topic order.change: key=None value={'orderID': 'order-7', 'state': 'COMPLETE'}
Sleeping until next iteration
```

Figure 25. The order_PROCESSING_process output

The build status and stage status were shown in the terminal. The duration, estimated duration and job status results is in Build Status section. The stage status includes name of each step, the status, and the time it takes for each step. In the end, the order status changes to COMPLETE.


```
_id: ObjectId("620a8d756bb92c34ff8293ee")
orderID: "order-7"
productID: "P4"
productName: "grafana"
jobName: "deploy-grafana"
buildID: "500"
url_jenkins_job_st…: "http://localhost:8080/job/deploy-grafana/500/api/json"
> parameters: Array
v buildStatuses: Array
  v 0: Object
      status: "SUCCESS"
      duration: "125.29s"
      estimateDuration: "103.935s"
v stageStatuses: Array
  v 0: Object
      name: "Declarative: Checkout SCM"
      status: "SUCCESS"
      duration: "688ms"
  v 1: Object
      name: "Declarative: Tool Install"
      status: "SUCCESS"
      duration: "107ms"
  v 2: Object
      name: "Setup parameters"
      status: "SUCCESS"
      duration: "375ms"
  v 3: Object
      name: "Get GCP access token"
      status: "SUCCESS"
      duration: "733ms"
  v 4: Object
      name: "Git Checkout"
```


Figure 26. Build and stage status in database

Figure 26 shows that the build and stage status has been saved to database.

```
_id: ObjectId("61dec5de3f9fe32b6b6bf6c6")
orderID: "order-7"
accountCustomerID: "123"
accountProviderID: "999"
state: "COMPLETE"
orderStatus: ""
> products: Array
```


Figure 27. State changes to COMPLETE

The state of the order has changed to COMPLETE.

7 CONCLUSIONS

The main objective of the thesis was to develop a module that enables the user to install needed tools and software on a Kubernetes cluster through the data given by the customer. The implemented application succeeded in receiving Apache Kafka order, processing, and saving the output to the database. Besides, the resources were deployed properly on the GKE cluster.

Each process of the application has a docker file so it will be convenient to build and run a docker image. This ensures that the software is easy to move and maintain in the future.

The most challenging part in the project was the vast number of technologies. Each tool had to be learnt to find out how they work and how to apply them in the thesis work. However, thanks to lots of detailed tutorial on the Internet, this was done successfully. In addition, security was also a complicated task. A decision had to be made to choose the best way to authenticate to GCP, Jenkins, and MongoDB and make sure that no one could see the credentials for malicious purpose.

7.1 Future work

Although the application has achieved the needed requirements, there are many ways to improve the application. Firstly, a subtle GUI should be implemented so that users can run each process in the same program. That will enhance the usability of the application. The user could use the GUI to display output and collections in the database.

Furthermore, there should be more Jenkins pipelines to deploy more applications, such as MongoDB, MySQL, and other types of databases. The users could select the resources they like to deploy.

REFERENCES

1. Salesforce. Benefit of clouds. Accessed 13.11.2022. https://www.salesforce.com/ca/hub/technology/benefits-of-cloud/.
2. Syndeno. Accessed 25.10.2022. https://www.syndeno.com/en/about-us/
3. Coursera. What is Python used for? Accessed 13.09.2022. https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python
4. Whizlabs. Introduction to Google Cloud Platform. Accessed 13.09.2022. https://www.whizlabs.com/blog/google-cloud-platform/
5. Google Cloud. GKE overview. Accessed 14.09.2022. https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview
6. Bigelow, Stephen J. 2015. Google Kubernetes Engine (GKE). Accessed 25.10.2022 https://www.techtarget.com/searchitoperations/definition/Google-Container-Engine-GKE
7. Taylor, David. 2022. What is MongoDB? Accessed 24.10.2022 https://www.guru99.com/what-is-mongodb.html
8. Saurabh. 2022. What is Jenkins? Accessed 14.09.2022. https://www.edureka.co/blog/what-is-jenkins/
9. Terraform. What is Terraform? Accessed 14.09.2022. https://www.terraform.io/intro
10. Terraform. Terraform use cases. Accessed 15.09.2022. https://www.terraform.io/intro/use-cases

11. Kozlovski, Stanislav. 2017. A Thorough Introduction to Apache Kafka. Accessed 15.09.2022. https://betterprogramming.pub/thorough-introduction-to-apache-kafka-6fbf2989bbc1
12. Docker overview. Accessed 15.09.2022. https://docs.docker.com/get-started/overview/
13. Jordan, Jeremy. 2019. An introduction to Kubernetes. Accessed 16.09.2022. https://www.jeremyjordan.me/kubernetes/
14. Onyszko, Tomasz. 2021. Why should you use Kubernetes in 2022? Accessed 16.09.2022. https://www.predicagroup.com/blog/why-kubernetes-2022/
15. Santos, Lucas. 2021. What is Helm Chart? A tutorial for Kubernetes Beginners. Accessed 16.09.2022. https://www.freecodecamp.org/news/what-is-a-helm-chart-tutorial-for-kubernetes-beginners/
16. Terraform. Provision a GKE cluster. Accessed 13.11.2022. https://learn.hashicorp.com/tutorials/terraform/gke