ECE 5904: Project and Report

Utilizing Docker and Kafka for Highly Scalable Bulk Processing of Electronic Thesis and Dissertation (ETDs)

Author: Dhanush Dinesh

Chair: Dr. Edward A. Fox

Committee Members:
  Dr. Creed Jones
  Dr. Nektaria Tryfona
  Dr. Prashant Chandrasekar

May 9, 2023
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
Abstract

This report discusses the utilization of Docker and Kafka for the bulk processing of Electronic Thesis and Dissertation (ETD) data. Docker, a containerization platform, was used to create portable Docker images that can be deployed on any platform making them platform-agnostic. However, managing a large infrastructure with interconnected Docker containers can be complicated. To address this, Kafka, an open-source, distributed message streaming platform, was incorporated into the pipeline to make each service independent and scalable.

The report provides a comprehensive discussion on how a pipeline was developed to maximize resource utilization and create a highly scalable infrastructure through the use of Docker and Kafka. Multiple Kafka brokers were deployed to ensure high availability and fault tolerance, and Zookeeper was used to track the status of Kafka nodes. Rancher was used to deploy the infrastructure on the cloud, which employs Kubernetes to manage the deployment and services.

The report also highlights the advantages of the current setup over previous workflow automation in terms of elapsed processing time and parallel processing of data. The system design includes a Kafka producer that produces ETD IDs to be processed and a segmentation container that acts as a consumer and polls the Kafka broker. Once the ETD IDs are received, the container starts processing, and the segmented chapters are stored in a shared Ceph file location. The process continues until all the ETDs are fully processed. This integration has the potential to benefit researchers who require large amounts of ETD data processed at a scale that was previously unfeasible, enabling them to make more robust and data-driven conclusions.
Contents

List of Figures v

1 Introduction 1
   1.1 Overview ...................................................... 1

2 Literature Review 3
   2.1 Cloud-based service provisioning ............................ 3
   2.2 Docker containers ............................................. 3
   2.3 Kubernetes .................................................... 3
   2.4 Apache Kafka .................................................. 4

3 Components 5
   3.1 Docker .......................................................... 5
   3.2 Kubernetes ...................................................... 6
   3.3 Rancher .......................................................... 6
   3.4 Flask API ......................................................... 8
   3.5 Apache Kafka and Zookeeper ............................... 8
   3.6 Reasoner Engine ............................................... 9
   3.7 Ceph ............................................................. 10

4 Design 11
   4.1 Different architectures .................................... 11
   4.2 Current architecture for bulk processing .................. 12
      4.2.1 Advantages of current design ....................... 13

5 Implementation 15
   5.1 Setting up of Kafka and Zookeeper ....................... 15
   5.2 Frontend UI .................................................. 16
5.3 Backend Flask API ................................................. 17
5.4 Segmentation .................................................... 18
5.5 Parse and clean .................................................. 19
5.6 Summarization .................................................... 19
5.7 Classification ..................................................... 21

6 Developer Manual .................................................... 23

7 Conclusion ............................................................ 25

8 Future Work .......................................................... 26

Bibliography ............................................................ 27
List of Figures

3.1 Containers vs. Virtual machines [20] ........................................... 5
3.2 Rancher architecture as used in Discovery cluster .............................. 7
3.3 Zookeepr and Apache Kafka [4] ................................................ 8
3.4 Workflow automation design [26] ................................................. 9

4.1 System diagram for information retrieval and analysis system 2020 [13] . . . 11
4.2 System architecture 2021 [26] .................................................. 12
4.3 Current architecture for bulk processing of ETDs .............................. 13

5.1 Zookeeper logs ............................................................................. 15
5.2 Bulk processing feature page UI ................................................. 16
5.3 Logs of the Kafka producer ......................................................... 17
5.4 Logs of segmentation producer and consumer service ......................... 18
5.5 Logs of parse and clean producer and consumer service ....................... 19
5.6 Logs of summarization producer and consumer service ......................... 20
5.7 Logs of classification producer and consumer service .......................... 22
Chapter 1

Introduction

Cloud computing has transformed the management and deployment of applications in recent times. Infrastructure as a Service (IaaS) providers like Amazon Web Service (AWS), Google Cloud Platform (GCP), and Microsoft Azure have gained immense popularity for their ability to handle infrastructure and decouple physical resources. Besides, containerization of code has also contributed to the creation of software applications that can be easily packaged and deployed in portable containers with specific environments, making them platform-agnostic. For this project, Docker, a well-known containerization platform, was utilized to create Docker images that can be deployed on any platform.

While Docker offers a convenient way to containerize software, managing a large infrastructure with numerous interconnected Docker containers can prove to be a complicated task. Electronic Thesis and Dissertation (ETD) data, which has seen an increase in usage in recent years, presented a challenge in our setup, with a vast amount of data (over 0.5 million) that required processing. To address this challenge, Kafka was incorporated into the pipeline to make each service independent. Kafka is an open-source, distributed event streaming platform that many companies rely on due to its high performance and low latency for data processing. Additionally, Kafka is designed to be scalable, highly available, and fault-tolerant.

The report focuses on the utilization of Docker in conjunction with Kafka to enable the bulk processing of ETDs, which can be beneficial for researchers. The report provides a comprehensive discussion on how a pipeline was developed to maximize resource utilization and create a highly scalable infrastructure through the use of Docker and Kafka.

1.1 Overview

The topics covered in the report are as follows

1. Introduction to the project
2. Literature review
3. Components
4. Design
5. Implementation
6. Developer manual
7. Conclusion
8. Future work
Chapter 2

Literature Review

2.1 Cloud-based service provisioning

Cloud based companies provide various layers of service:

1. Infrastructure as a Service (IaaS)
2. Platform as a Service (PaaS)
3. Software as a Service (SaaS)
4. Container as a Service (CaaS)

For this project, CaaS is the level of abstraction I have chosen which involves managing Docker containers and other services which are discussed more in Chapter 3.

2.2 Docker containers

Containers [18, 24] are lightweight, self-sustaining environments that are easy to build and deploy. Yadav et al. [29] found that Docker containers performed much better than virtual machines; they also conclude that Docker containers showed lower execution times compared to virtual machines. Docker containers provide better isolation and easier scalability. Docker is widely recognized as a leading platform for managing containers, and has played a pivotal role in driving the widespread adoption of container technology.

X. Wan et al. [28] explore the utilization of Docker for deploying microservices and optimizing their performance. Their paper introduces a distributed and incremental algorithm for deploying microservices, which is deemed to be more flexible and cost-effective due to its low resource usage.

2.3 Kubernetes

To support this project, I utilized the cloud.cs.vt.edu infrastructure [7] for deploying our services. The cloud.cs.vt.edu system in turn leverages Rancher [19], an open-source container management platform that provides an abstraction layer on top of a Kubernetes [3]
cluster. Rancher offers centralized logging, monitoring, workload management, and load balancing features, simplifying the management of containerized applications, and facilitating deployment across diverse environments.

Kubernetes [3], also referred to as K8s, is an open-source container orchestration platform that automates container deployment, management, and scaling. Originally developed by Google, it has become the standard for container orchestration and is widely employed by various organizations in their production environments.

### 2.4 Apache Kafka

Apache Kafka was evaluated for its performance [17, 27], and it was concluded that it was a more budget-friendly and efficient option for traditional batch applications. M. M. Rovnyagin et al. [22] described a comprehensive approach to provide fault tolerance for web applications even when instances of the application are down, using long polling of the request queue. This method was incorporated into the present work. Another paper [6] discussed a design approach for improving Kafka network fault tolerance using the pub-sub messaging mechanism, thereby making the Kafka broker more reliable.

Furthermore, a proof-of-concept (PoC) for developing an effective framework for continuous integration and continuous delivery to automate source code compilation was demonstrated [25], which serves as a useful reference for achieving the required CI/CD [21] in this project. These studies demonstrate the growing interest in using Kafka [12] for large document processing and real-time data processing, offering insights into designing and implementing Kafka-based systems for processing large documents. As data volumes continue to increase, Kafka-based systems are likely to become increasingly critical for efficient and scalable large document processing.

Overall, this literature review highlights the importance of an efficient and reliable infrastructure for processing, storing, and preserving ETDs, and indicates that Docker and Kafka provide significant benefits in terms of scalability, reliability, and portability. As cloud infrastructure continues to evolve, Docker and Kafka are likely to remain crucial components of the modern technological landscape.
Chapter 3

Components

3.1 Docker

Docker is a widely adopted containerization platform that simplifies building, running, and deploying containerized applications. Containers are lightweight and portable packages that include all the necessary environments and libraries to run the code independently. One of the major advantages of containers is their platform abstraction, which enables them to be deployed on any machine without the need for a guest operating system, unlike traditional virtual machines. In Figure 3.1, we can see the main difference between containers and VMs. Virtual machines virtualize the machine down to the hardware layer, while containers are virtualized on the software layer above the operating system level.

Docker provides a range of tools and services to help developers create, test, and deploy Docker applications. Some of the tools Docker provides are Docker Engine, Docker Compose, Docker Swarm, and Docker Hub. In this project, we have used all the above except Docker Swarm.

![Figure 3.1: Containers vs. Virtual machines](image)
1. Docker Engine [16] is the core component of the Docker daemon responsible for managing containers. It provides an API in the form of Docker CLI and Docker UI, which can be used to manage networking, create services, and attach volumes to containers.

2. Docker Compose [15] is a tool that helps developers deploy complex Docker applications. It takes care of the interconnections between various parts of the application and can be easily directed using a YAML file.

3. Docker Hub [23] is a public repository for storing and sharing Docker images, similar to GitHub for code. It provides a space to store images and collaborate with other team members.

### 3.2 Kubernetes

Large and complex applications deployed on the cloud with multiple containers running, which need to be interconnected and scaled according to demand, require a dedicated team to maintain. To address this issue, container orchestration systems such as Kubernetes (K8s) [3] have been developed.

Kubernetes is an open-source container orchestration system developed by Google to help deploy Docker containers, create services, manage web traffic with load balancing, and scale up applications. Kubernetes manages containers by grouping them into pods across different nodes, with each node potentially containing one or more pods, and each pod scalable independently.

Kubernetes manages networking and facilitates connections to persistent storage across a cluster of nodes. It was designed as a tool to make applications extendable, reducing developers’ tasks in managing network connections and enabling easy addition of new microservices with minimal changes to the existing infrastructure.

Kubernetes is popular because of its powerful management of containerized applications on the cloud, which reduces the burden on developers to maintain infrastructure, and enables them to focus on application development.

### 3.3 Rancher

Rancher [19] is an open-source platform for managing and deploying containers on top of Kubernetes. This provides a level of abstraction that is resource agnostic. The main benefit of using Rancher on top of Kubernetes is that it provides a simple and intuitive user interface for managing and deploying containers. It rivals the rich User Interfaces of the cloud
computing giants such as AWS [14] and GCP [9]. Rancher provides a centralized logging service that helps in debugging and troubleshooting.

Rancher uses Kubectl [2], which is a command line interface tool to control the Kubernetes cluster. Kubectl is essential for creating a CI/CD [21] pipeline. Figure 3.2 shows the relationship between Rancher, Kubernetes, Kubectl, Pods, and the Docker containers. A pod is an environment that supports a container; each pod can contain multiple containers. A pod is removed when all the containers in the pod are removed.

Figure 3.2: Rancher architecture as used in Discovery cluster

Figure 3.2 shows the Rancher architecture in the Discovery cluster [7].
3.4 Flask API

Flask API [8] is a web framework for building RESTful APIs using the Flask micro-web framework in Python. Flask APIs are simple and intuitive for building backend web applications with minimal code required to get started.

When creating the bulk processing page, a backend server was required to serve the request and queue the ETDs to start processing. An API was needed to trigger the Kafka producer to start the processing of ETDs.

3.5 Apache Kafka and Zookeeper

Apache Kafka [12] is an open-source distributed streaming platform developed by the Apache Foundation. Kafka is designed to handle high volumes of data, which makes it the best tool for building scalable and extendable data pipelines.

A Kafka broker works on a publish-subscribe mechanism. The producers produce the data and publish the message on the broker to a particular topic. The consumers poll the broker on a particular topic to fetch data. Kafka is designed to be highly scalable and to support real-time data processing. Hence we can have multiple Kafka brokers, to which the producers can publish, and consumers can subscribe. To manage this, Zookeeper [10] is used to maintain the different Kafka brokers.

![Diagram of Zookeeper and Apache Kafka](image)

**Figure 3.3: Zookeeper and Apache Kafka [4]**

Zookeeper provides distributed synchronization. As shown in Figure 3.3, Zookeeper manages Kafka by keeping track of the Kafka brokers in a cluster and the brokers’ state, partitions,
availability, and leadership status. Zookeeper elects one of the brokers to be the leader of a topic, which helps in keeping the data synchronized. Further, it keeps track of the configurations. One of the key configurations is that of topic and partitions. If there is a new topic created or updated, Zookeeper updates all the brokers with the new configuration. Another major benefit of using Zookeeper is that it manages the scaling of the Kafka cluster which helps to balance the load across the cluster and quickly scales as required.

This allows Kafka to handle node failures and recover from them quickly and automatically, ensuring that the system remains highly available even in the face of node failures.

3.6 Reasoner Engine

A reasoner engine was created by the Integration team in 2020 [13]. The engine utilizes an algorithm that mines a given set of services and goals to produce a workflow. The services and goals were stored in a database with entries such as description, input, output, and other details. The workflow was generated using a context-free grammar, and the pipeline was created using Apache Airflow [11]. This was the previous setup for creating workflow-automated pipelines as shown in Figure 3.4.

Figure 3.4: Workflow automation design [26]
3.7 Ceph

CEPH [1] is an open-source distributed storage platform that is designed to provide a highly scalable, reliable, and efficient storage solution. Ceph is used for high reliability and data replication across various containers. We use Ceph for persistence storage. This is important since Docker containers lack persistent storage, which can create difficulties across container instantiations and when trying to share storage across different containers. Hence Ceph is used, with storage attached as volumes to a container or to a set of containers.
Chapter 4

Design

The pipeline discussed in this report was developed on the CS cloud infrastructure that runs on Rancher which leverages Kubernetes container orchestration.

4.1 Different architectures

Before designing a new pipeline, I studied different architectures developed by different teams from previous years. In Figure 4.1, the system diagram for information retrieval and analysis shows how a student team had created a framework for ETDs to be indexed using Elasticsearch, and also had devised workflow automation [13] for creating pipelines.

As suggested by Prashant Chandrasekar in his dissertation “Continuously Extensible Information Systems: Extending the 5S Framework by Integrating UX and Workflows” [5], students aimed to create a system to support subject matter experts (SMEs) in achieving...
their goals. This involved designing a system that would create a pipeline when given a goal. This approach was achieved in 2021 by team 5 [26] in the course CS 5604: Information Storage and Retrieval. They implemented a Reasoner which produced workflows. A workflow was created for the goal of an end user. For example, if the end user wanted the classified output of the ETD, the Reasoner would create a workflow including a set of services that had to run before reaching the final goal of classified output. To translate the workflow into a pipeline, Apache Airflow [11] was used, as shown in Figure 4.2.

![Figure 4.2: System architecture 2021 [26]](image)

While the team was able to create a pipeline for a given goal, the system had a few challenges. One of them was that the pipeline was only able to handle one ETD at a time, which made it impossible to process our large repository of ETDs.

### 4.2 Current architecture for bulk processing

After conducting a literature review on processing ETDs in bulk, Dr. Fox and I decided to create a Kafka broker with multiple topics to act as an intermediary between two services. In the current setup, I have created a pipeline that processes ETDs in parallel. To achieve this, I created a Kafka broker that is managed by Zookeeper, which keeps track of Kafka configurations. The system design is shown in Figure 4.3. To enable the publisher-subscriber model to work, I created a Kafka producer as a Docker container to produce the ETD IDs
to be processed. Once the ETD IDs were produced, and the Kafka topic was configured, the producer would queue the ETD IDs on the Kafka broker, with the first pipeline service, segmentation, as the topic.

The segmentation container at this point acts as a consumer as it polls the Kafka broker on the segmentation topic. Once the ETD ID is received, the container starts processing. The segmented chapters are stored in a shared Ceph file location. Once all the chapters are segmented, the segmented chapters are uploaded to the database. At this point, once the ETD is fully processed, the container acts as a Kafka producer and configures the ETD with a new topic, text extraction, and puts it back into the queue.

The ETD propagates from segmentation into parsing and cleaning, where the text is extracted; from text extraction to summarization; and then finally into classification. This process continues until all the ETDs are fully processed.

This design has a lot of advantages compared to the previous designs.

4.2.1 Advantages of current design

1. There were several limitations in the previous workflow automation used to produce a pipeline for single ETDs, and hence we could not process our entire repository of ETDs.
2. Additionally, the previous setup had longer processing times compared to our current setup since our current setup has only one setup time. Once the containers are running, we can process as many ETDs as required.

3. The parallel processing of data was achieved in the current setup by spinning up multiple containers at a click of a button as compared to the serial processing of data in the previous workflow automation process.

4. A easy-to-use UI was created for researchers to process their set of ETDs.
Chapter 5

Implementation

In this chapter, I describe my efforts to conceptualize the design of creating a scalable and highly reliable pipeline for bulk processing of ETDs using Kafka as a broker between different microservices. The two main components of this project are the creation of producers and consumers.

5.1 Setting up of Kafka and Zookeeper

I begin by creating the Kafka broker. To set up the Kafka broker we require a Zookeeper, as we discussed earlier. The Zookeeper manages the configuration setting. The Zookeeper container is the first container to start. I used container.cs.vt.edu/dhanushnd/docker-images/confluentinc/cp-zookeeper:7.3.0-2 for the container.

![Zookeeper logs](image)

Figure 5.1: Zookeeper logs

The Zookeeper exposes port 2181 for communication with other containers. The Zookeeper runs on a cluster IP which is a virtual IP address. This cluster IP will later help in load
balancing when we have multiple clusters. The URL for the container on the cloud is https://cloud.cs.vt.edu/p/c-k7rk9:p-wjqx4/workload/deployment:etd:zookeeper.

Once the Zookeeper is started as shown in Figure 5.1, the Kafka broker starts. I have used the container.cs.vt.edu/dhanushnd/docker-images/confluentinc/cp-kafka:7.3.0-2 image for the Kafka container, and hosted it on the cloud. The container has exposed port 9092 for communicating with the other containers and is configured to communicate with the Zookeeper on port 2181. The URL for the Kafka container on the cloud is https://cloud.cs.vt.edu/p/c-k7rk9:p-wjqx4/workload/deployment:etd:broker-kafka.

5.2 Frontend UI

The simple frontend, as shown in Figure 5.2, was created for the user interface. This page is accessible only to the administrator persona. Here, the admin would have to provide the range of ETDs to be processed. Once the start ETD ID and end ETD ID are provided, they can submit the request. As our pipeline is asynchronous, we can provide multiple inputs. If the provided values are valid integers, it calls the bulk process backend API.

![Figure 5.2: Bulk processing feature page UI](image)

The code for this is present in the GitLab repository https://code.vt.edu/aaron2000/
5.3 Backend Flask API

The backend API was written in Flask. The API has one POST method. The POST method receives two inputs: Start ETD ID and End ETD ID.

For queuing the ETDs in Kafka, the producer generate all the ETD IDs between the start and end ETD IDs. If the user has provided a start ETD ID that is greater than the end ETD ID, the code swaps the start and end ETD IDs. The ETD IDs are queued on the Segmentation topic which is the first service in the pipeline as shown in Figure 5.3.


![Logs of the Kafka producer](image)

Figure 5.3: Logs of the Kafka producer


\(^1\)Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster [3].
5.4 Segmentation

In the Segmentation microservice, the main script keeps polling the Kafka queue on the topic “segmentation” to check if there is any ETD queued for segmentation. This was one of the most challenging of all the microservices as the repository does not contain any of the model weights necessary to run the code. The model weights required for running the code can be downloaded from the following link: https://drive.google.com/drive/u/1/folders/1Hk1z69m9qqEQ1QzVzWU3CFP4x8GW01XY.

When we run the image, the main.py file runs and polls for the “segmentation” topic. If there is any ETD ID in the segmentation queue, it first creates a new folder for the ETD ID and downloads the PDF of the ETD ID. Once the ETD is downloaded, the ETD is passed to the segmentation.py file which generates the segmented chapters and stores them in the /segmented_chapters folder.

Once the chapters are segmented, each chapter is uploaded into the database using the POST API, and a segmented metadata JSON file is stored in the /clean_chapters folder. The link to the repository is https://code.vt.edu/ano22/cs5604-team-4-segmentation.

The Docker image used for the container is container.cs.vt.edu/dhanushnd/docker-images/segmentation:v1-git and is currently hosted on cloud.cs.vt.edu at https://cloud.cs.vt.edu/p/c-k7rk9:p-wjqx4/workload/deployment:etd:bulkapi-segmentation
5.5 Parse and clean

In Parse and Clean, the ETD chapters are converted from PDF to text. The code for this was taken from the CS5604-f22-team-4-Summarization repository but I have created a new repository for Parse and Clean. In the new repository, I created the main.py which is responsible to poll for the topic “textExtraction”. This service takes the segmented chapters as the input, extracts the plain text, and stores the resulting text in the /parse_and_clean folder as .txt files as shown in Figure 5.5. Once the Parse and Clean is completed, the ETD ID is queued by Kafka under the “summarization” topic.

![Logs of parse and clean producer and consumer service](image)

Figure 5.5: Logs of parse and clean producer and consumer service


5.6 Summarization

For the summarization microservice, the code base is located at https://code.vt.edu/harishbabu/cs5604-f22-team-4. Since the code base had a lot of additions, I had to modify the summarization.py file and use the one in the “team5-use” branch. In this code
base, I added the main.py file with the runSummarization() function which prepares the input and output paths and calls the summarization.py file. The output of the summarization is stored in the /summarization folder.

Once summarization is done, we upload the summarized text to the DB using the uploadSummerizedChapter function which calls the API. The logs of the service are shown in Figure 5.6. I made modifications to the utils.py file to enable writing JSON for all the chapters. Previously, the code was only functioning for a single chapter.

![Figure 5.6: Logs of summarization producer and consumer service](image)

The Dockerfile was updated with all of the required libraries, and the CMD command was updated to run the main.py file with the config file. The main.py file acts as the master Kafka consumer for the topic “textExtraction2” and produces the next topic “classification” once the processing is done.

I merged the code I wrote back to the repository. The Docker image for this container is container.cs.vt.edu/dhanushnd/docker-images/summarization:v11 and is hosted on cloud.cs.vt.edu at https://cloud.cs.vt.edu/p/c-k7rk9:p-wjqx4/workload/deployment:etd:bulkapi-summmarization.
5.7 Classification

For the classification code, I added the main function which handles the Kafka queue for the topic “classification”. I made modifications to the code to classify all the chapters generated. The code base is located at https://code.vt.edu/ano22/cs5604-team4-classification/-/tree/main/Flask_app?ref_type=heads. Once all the chapters are classified, the classified objects are uploaded to the database, as shown in Figure 5.7.

I also wrote a custom Dockerfile for this repository as it was missing. This Dockerfile runs the main.py file, which is the master Kafka consumer for the topic “classification”.

The model weight file required to run the code is not present in the repository. It needs to be downloaded from the shared CEPH folder. The path where the model weight file can be found is /data/Team4/Classification/saved_models/chapter_scibert_8_final_weights_chapter.pt, and the link to the path is https://team4-container-1.discovery.cs.vt.edu/tree/data/Team4/Classification/saved_models.

Some of the challenges I faced during the integration were that I was not able to generate classification for each chapter, hence I made some code changes, so it would work for all the chapters that were generated.

The container for the classification is bulkapi-classification and the Docker image for classification is container.cs.vt.edu/dhanushnd/docker-images/classification:v10. (This image does not remove the folder in the shared volumes and is only for demonstration purposes.) The container is hosted on cloud.cs.vt.edu at https://cloud.cs.vt.edu/p/c-k7rk9:p-wjqx4/workload/deployment:etd:bulkapi-classification.

The last and final service is the clean-up script. The deleteFolder function helps clean up the files and is present in dhanushnd/classification:v12. It should be used in production runs.
Figure 5.7: Logs of classification producer and consumer service
Chapter 6

Developer Manual

1. Kafka set-up
   Use the following Docker command to spin up Kafka and Zookeeper.
   
   ```
   docker compose up -d
   ```
   
   After the containers have been deployed, there are a few steps that must be followed on the Rancher UI to access the containers.

2. Bulk API
   We need the bulk API backend to run to listen to the POST requests sent by the front end to run the producer script.
   
   ```
   docker build --platform linux/amd64 -t <bulkapi_name>:<tag_name> -f .
   docker tag <repo_name>/bulkapi_<name>:<tag_name> <bulkapi_name>:<tag_name>
   docker push <repo_name>/bulkapi_<name>:<tag_name>
   ```

3. Kafka Producer
   If you have any code changes to the producer you need to run the following code to build the image.
   
   ```
   docker build --platform linux/amd64 -t <producer_name>:<tag_name> -f <DockerFile_name> .
   ```
   
   Once you build the image, tag it to your repository and push it to the container registry. Once its done, you can use it in the cloud or locally.

4. Create the Kubernetes script from docker-compose (OPTIONAL)
   This step is run only if you make any changes to the docker-compose.yml file. The Kompose command converts a Docker compose file into a Kubernetes deployment script.
kompose convert -f ./deploy/docker-compose.yaml

5. Deploy to cloud
   Once you have all the deployment files ready, you can deploy it on the cloud by running
   the Rancher command. You should have Rancher installed and be signed in using the
   Rancher access key.

   ```
   rancher kubectl -n etd apply -f ./deploy
   ```

6. Cleanup deployment - local machine
   Run the “docker compose down” command to bring down the Kafka and Zookeeper
   services locally. (Run this command to cleanup deployment on your local machine.)

   ```
   docker compose down
   ```

7. Cleanup deployment - cloud
   Run the “kubectl stop” command to bring down the Kafka and Zookeeper services on
   the cloud.

   ```
   rancher kubectl -n etd stop -f ./deploy
   ```
Chapter 7

Conclusion

Developing a batch processing pipeline was crucial for efficiently utilizing the current infrastructure to handle the large ETD repository. Kafka, an open-source queuing system, was chosen due to its high reliability, scalability, and fault-tolerant features. The project successfully achieved the following objectives:

1. Established a robust pipeline for processing large amounts of data.
2. Integrated various services such as segmentation, text extraction, summarization, and classification into a single system.
3. Provided a simple user interface for researchers to test their services on a large dataset.
4. Reduced processing time by utilizing parallel processing and minimizing setup time.
5. Built a highly scalable system that requires low maintenance due to its continuous integration capability with code changes.
6. Developed a user-friendly development manual to aid in maintaining the infrastructure.
Chapter 8

Future Work

The current setup is an initial version of the pipeline. This could be improved to make the system more robust and user-friendly. Here are some suggestions for improvement:

1. Pipeline
   (a) The containers in the current setup are all configured on the “bigRAM” node, which limits the processing capability of each container. If hardware resources improve, it would be better to allocate each container to a dedicated node.
   (b) If resources increase, it would be beneficial to have multiple segmentation services, as it takes the longest time to process.
   (c) Additional parallelism could result if after segmentation, the processing of the chapters of an ETD could proceed in parallel, such as by having a queue for the pair: ETD ID, chapter number.

2. CI/CD
   (a) Due to time constraints, CI/CD pipelines were not created for each container. Implementing a CI/CD pipeline would simplify the life of developers.

3. User Interfaces
   (a) Enhancing the logging mechanism to capture backend processing activities would benefit the user.
   (b) Limiting page accessibility to only the admin would facilitate restricted access.
   (c) To enhance queue management efficiency for users, the addition of a new “Clear Queue” feature or the removal of IDs from the existing Queue feature could be considered.
Bibliography


