Final Report

Team 5 - Infrastructure and DevOps

CS5604

Information Storage and Retrieval

Subject Matter Experts: Dhanush Nanjundaiah Dinesh and Satvik Chekuri

Adeyemi Aina

Amritha Subramanian

Hung-Wei Hsu

Shalini Rama

Vasundhara Gowrishankar

Yu-Chung Cheng

January 17, 2024

Blacksburg, Virginia, USA - 24061

Abstract

The core objective of this project is to construct an advanced system capable of efficiently searching and retrieving pertinent information from an extensive collection of electronic theses and dissertations. This system represents an enhancement over existing search and retrieval technologies, specifically targeting significant documents, such as academic textbooks, dissertations, and theses. This stands in contrast to conventional search engines, which excel at handling simpler content like websites or blogs.

The development of this system is divided into six major areas, with each area assigned to a dedicated group of graduate students. These areas include Knowledge Graph (Team 1); Search and Indexing (Team 2); Object Detection and Topic Analysis (Team 3); Language Models, Classification, Summarization, and Segmentation (Team 4); Integration (Team 5); and User Interaction (Team 6). The teams and their functions are structured to replicate the environment of a company engaged in new product development.

Team 5 is primarily responsible for integrating the collective efforts of all teams involved in the project. The team is crucial in managing access to the file system and database infrastructure, ensuring seamless collaboration and data accessibility. Additionally, we provide valuable assistance with APIs, enabling efficient communication between different system components. Moreover, we coordinate the utilization of containers within the CI/CD pipeline, facilitating a smooth development and deployment process.

The team is also responsible for building an efficient container cluster to support our CI/CD pipeline and maintaining containers for all teams. To ensure thorough processing capabilities, we are in charge of creating and maintaining APIs tailored to each team's individual requirements. Additionally, it's important to note that the database and file system are exclusively accessible by our team. Our team will handle all database access requests from

other teams via APIs.

The INT team expresses sincere gratitude for the invaluable contributions of the teams that came before us and the predecessors who paved the way. Their substantial efforts have laid the solid foundation upon which we are now constructing our ambitious project. We recognize and appreciate the collective dedication that has made this endeavour feasible and successful.

The team has made significant progress in migration from Discovery to Endeavour by successfully addressing knowledge gaps and leveraging various resources to enhance our understanding. Our meetings with the Subject Matter Experts (SMEs) provided valuable insights into project requirements and the intricacies of migrating from the old to the new cluster. We have established a robust CI/CD pipeline for efficient testing and deployment and have written a CI/CD guide for other teams to follow. The second version of the database schema has been created, and the migration of the necessary data fields from version 1 to version 2 has been completed. We have also added the list of 200 ETDs to version 2 of the database. The challenges faced with Kafka have also been addressed, and we have distributed a sample Kafka producer-consumer in Python to other teams. We have been involved in implementing the authentication service and have established APIs for the file system and database. This enables other teams to access and share data among themselves.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Overview

1.1 Collaboration and Communication

Our team of six members has established effective channels for communication and collaboration. Our Slack [1] workspace, "Infrastructure team for IR (Team 5)," has dedicated channels where we hold all of our internal and external communications and share any context. In order to facilitate collaboration among individuals from various teams or functional areas, we have established cross-functional channels within Slack. While an in-person meeting is the team's preferred format, Zoom [2] is available for those who might not be able to make it to the collaboration sessions. Each team has a dedicated liaison whose information is shared on the team page on Canvas. We utilize class sessions as synchronous liaison sessions to coordinate and engage with other teams. We have used these meetings to gather requirements and support other teams. All asynchronous communication occurs on our Canvas page, Slack channel, Discord channel, and email.

1.2 Problems Faced

In the first stages of setup, we encountered several challenges that required solutions. In this section, we address the following issues:

1. Restructuring the overall system architecture.

2. Identifying user needs from the relevant teams.

3. The majority of the members of our team had no prior experience working with DevOps [3] tools such as Kubernetes, Docker, etc.

4. While migrating containers from the Discovery cluster to the Endeavour cluster, we faced issues with bringing Kibana alive.

5. The service migration from Discovery cluster to the Endeavour cluster experienced image pull failures.

6. The services were restricted to accessing other pods through the Ingress [4] service in Kubernetes, due to the absence of an Ingress configuration, which prevented external network traffic from reaching its associated pod [5].

7. Elasticsearch experienced a security breach, resulting in data loss. We worked closely with the indexing team to improve the system's security and resilience.

8. The Continuous Integration/Continuous Deployment (CI/CD) setup [6] encountered issues with uploading images to GitLab and occasionally failed to execute the GitLab Runner [7].

1.3 Solutions Developed

In order to solve the above-mentioned issues, we spent time gathering user needs, understanding the project team's objectives, and then proceeded to build the solutions. Our subject matter experts, Mr. Satvik Chekuri and Mr. Dhanush Dinesh, as well as the teaching assistant for the course, Ms. Xiao Liang, were in touch with the team and actively laid out the process in addressing some of the issues and anticipating potential barriers in developing the solutions. They subsequently connected the team with Mr. Chris Arnold, who facilitated obtaining the logs collection system. We owe our deepest gratitude for their assistance in this. We also owe gratitude to the team from Fall 2020 for their work [8], which gave us some guidance regarding the solution and architecture design. The objective was to enhance Fall 2020 team's solution. We were able to come up with the following solutions in particular to the issues described above:

1. To address the architecture challenge, we decoupled the system's components and services with direct access to the file system and Postgres database into distinct microservices.

2. To understand the needs of other teams, we established effective communication channels with the relevant teams and conducted thorough requirements gathering. This enabled us to comprehensively understand the infrastructure team's objectives and set the foundation for the success of the project.

3. Addressing the knowledge gaps in infrastructure management utilizing DevOps tools like Kubernetes, Docker and Gitlab Runner, the team proactively studied various resources to acquire the necessary knowledge and held meetings with subject matter experts to gain a deeper understanding of these technologies.

4. To resolve the challenges with Kibana during migration, we took the following steps:

   (a) Updated the token from Elasticsearch to Kibana.

      i. Ran `bin/elasticsearch-service-tokens create elastic/kibana kibana-token` in the Elasticsearch container to generate a token.

      ii. Put the token into the Kibana YAML configuration.

   (b) Updated the port of the Ingress in Elasticsearch and the port of Elasticsearch in Kibana.

5. To resolve the image pull issue, the secrets and configurations in the Endeavour cluster were updated.

6. We reached out to the CS tech team, specifically Chris and Ben, to assist in identifying the Calico service crashing issue. They helped to restart the network policy service, Calico [9], to resolve this issue.

7. Due to the actions of the Computer Science (CS) technical team, which involved restarting nodes in our cluster, some non-persistent data was lost. This included the loss of the security token for Elasticsearch. To address this loss and enhance future resilience, the following was implemented:

   - Enhanced Storage: Persistent storage was mounted for Elasticsearch configurations, ensuring critical data is preserved beyond cluster node restarts.
   - Security Reinforcement: Robust security measures were implemented across both Kibana and Elasticsearch. This included:
     - Token Regeneration: A new security token was generated for Elasticsearch, replacing the compromised one.

- Identity Access Management: Usernames and passwords were established for all associated services, enabling granular access control and authorization.

8. The GitLab Runner was encountering issues with failing to clean up Docker resources, resulting in excessive storage usage. To address this, we established a cron job that routinely clears unused Docker resources, ensuring the smooth operation of the CI/CD VM. Additionally, we have assisted other teams in debugging their Dockerfiles, which were causing the creation of overly large container images, thereby hindering their upload to GitLab's container repository.

1.4 Future Work

Based on the progress achieved during this semester, several areas deserve attention and could potentially enhance the system. These include:

1. **Database Schema Enhancements**: Continuously monitoring and evaluating the database schema has proven beneficial. Future teams should consider this as an ongoing process to ensure the system remains aligned with evolving data requirements.

2. **Implement and Test Documented APIs**: Complete implementation of documented APIs required by other services and teams with open standards like Swagger, then provide the APIs for the respective services for test and production.

3. **Implement Outstanding Named APIs**: The team implemented 45 APIs based on requirements from other teams. The 9 remaining APIs are identified for future implementation. The API endpoints are listed in Table 1.1.

4. **Implement Test Routines for the Respective APIs**: Future efforts should include

implementing unit tests for each API to validate their functionality. By adopting this testing approach, the team can promptly identify and address any potential issues or downtime in the system. To streamline the testing process, an automated test routine can be developed through a robust CI/CD (Continuous Integration/Continuous Deployment) pipeline. This proactive testing strategy ensures the reliability of the APIs and contributes to the overall stability and resilience of the system.

5. **CI/CD Pipeline Implementation**: Our experience in establishing a CI/CD pipeline has been instrumental in improving development efficiency. Future teams could explore enhancing this further by automating additional steps in the deployment process to ensure quicker, error-free releases. Also, implement CI/CD pipelines for database setup and APIs. Future teams can focus on ensuring all teams implement a test routing and deployment pipeline using the sample CI/CD guide [10].

6. **Data Pipeline and Streaming**: Having accomplished significant milestones, the next focus for future teams is to advance our messaging infrastructure further. Specifically, we have successfully scaled up Kafka to fortify our messaging backbone and establish a more resilient message queue system. This achievement enables teams to handle various messages more efficiently, ensuring enhanced performance. Furthermore, we have strategically expanded our Kafka deployment to support these improvements. Future teams should prioritize their efforts on refining the system's performance, optimizing the management of increased request volumes, and maintaining efficient load distribution. These targeted initiatives will serve as a solid foundation for upcoming teams, guiding their work towards continuous improvement and innovation in our messaging capabilities.

7. **Populate and Test APIs Related to File System**: The APIs related to the file system were implemented, such as storing XML, page images, detected objects, and

functionalities to get ETDs and retrieve page images. However, the APIs have yet to be actively used by other teams on our Endeavour cluster. These APIs require testing and collaborative work. Additionally, there's a need to move chapter folders from the 200+ segmented ETDs into the directory of 500k ETDs under etdrepo. Moreover, continued collaboration with Teams 3 and 4 is essential to complete the tasks related to file system access and management.

API	Description	Request Method	Status
/api/v1/recommendation/infer	Takes in user-id to infer recommendations for the user.	POST	Named
/api/v1/recommendation/train	Takes in ETDs read of all user and ETD data to train a model for recommendation	POST	Named
/api/team2/etd-etd-neighbours	Post to ETD-ETD neighbors table	POST	Named
/api/team2/etd-etd-neighbours	Get data from ETD-ETD neighbors table	GET	Named
/api/team2/get-user-modules	Get data User-classes, User-queries-clicks, User-topics	GET	Named
/api/team2/get-object-metadata	Get data from Object Metadata - For Image captions	GET	Named
/api/v1/logs/<user-id>	API to fetch logs of a specific user by the user id, used by recommendation service	GET	Named
/v1/team-1/graphQL	Get data from various tables, complex queries and unions for Knowledge Graph service	GET	Named
/api/team2/etd-etd-neighbours	Get data from ETD-ETD neighbors table	GET	Named

Table 1.1: List of outstanding API endpoints for future work

Chapter 2

Literature Review

2.1 Container as a Service and Kubernetes

Container as a Service (CaaS) has gained significant attention in recent research, primarily due to its ability to provide a scalable and efficient environment for deploying and managing containerized applications. According to the work by Martin et al. [11], CaaS platforms offer simplified deployment models and facilitate the orchestration of container clusters, leading to improved resource utilization and enhanced application performance. Moreover, the research by Sharma and Khattar [12] highlights the role of CaaS in enabling seamless integration with existing infrastructure and emphasizes its potential for optimizing resource allocation in dynamic computing environments.

Kubernetes, an open-source platform for automating containerised application deployment, scaling, and management, has emerged as a prominent solution for orchestrating container clusters. The seminal work by Burns et al. [13] extensively discusses the design principles and architecture of Kubernetes, highlighting its role in providing robust container management

capabilities that we are implementing in the project. Furthermore, the research conducted by Arora and Chaudhary [14] emphasizes the scalability and fault-tolerance features of Kubernetes, making it an ideal choice for deploying and managing microservices-based applications in a distributed computing environment.

2.2 Elasticsearch

Elasticsearch has gained prominence as a robust and scalable search engine and analytics platform, revolutionizing the landscape of information retrieval and data analysis. Gormley and Tong [15] discuss the distributed architecture and advanced indexing capabilities of Elasticsearch, underscoring its ability to handle both structured and unstructured data efficiently. Additionally, Modi et al. [16] emphasize Elasticsearch's real-time search and analysis capabilities, showcasing its efficient querying mechanisms and versatile data visualization options across various application domains.

2.3 Jupyter Notebooks

Jupyter Notebook is an open-source web tool that enables users to produce and disseminate documents featuring live code, visuals, equations, and narrative text. The stewardship and maintenance of the Jupyter Notebook are carried out by the team at Project Jupyter [17].

Originating as a subsidiary project of IPython, Jupyter Notebooks were once part of IPython's own Notebook initiative. IPython is an evolving project, progressively embracing components that are independent of programming language. The term "Jupyter" is derived from the principal programming languages it supports.

2.4 GitLab and CI/CD

Continuous Integration and Continuous Deployment (CI/CD) play a pivotal role in modern software development by automating key stages in the development lifecycle. Implementing CI/CD practices enables accelerated delivery cycles and seamless application deployment with minimal manual intervention [18]. The CI/CD pipeline orchestrates essential tasks such as code integration, automated testing, and deployment, leading to improved software delivery efficiency.

GitLab, a widely recognized open-source repository management and collaboration platform, offers a comprehensive suite of tools supporting CI/CD pipeline implementation. Serving as a powerful Version Control System (VCS), GitLab provides public and private repository options, empowering development teams to efficiently manage source code and collaborate seamlessly across diverse projects [19]. By leveraging GitLab's integrated CI/CD capabilities, software teams can automate critical development tasks like code building, testing, and deployment. This streamlines the software delivery process and ensures the rapid, reliable delivery of high-quality applications [20]. In the context of this project, GitLab is our predominant Version Control System, employed to guarantee a more expedited, trustworthy, and collaborative process for software development.

2.5 Postgres SQL

In our effort to better understand PostgreSQL (Postgres SQL) and its use in database management, we explored the book "Practical PostgreSQL" [21]. This resource improved our skills in creating and managing table relationships and designing database schemas. We also learned the importance of selecting the right data types for each column to ensure

efficient and accurate data storage.

Additionally, our research extended to the practical side of running PostgreSQL in a Docker container [22]. Understanding how to deploy PostgreSQL in Docker provides a consistent database management environment.

Chapter 3

Components

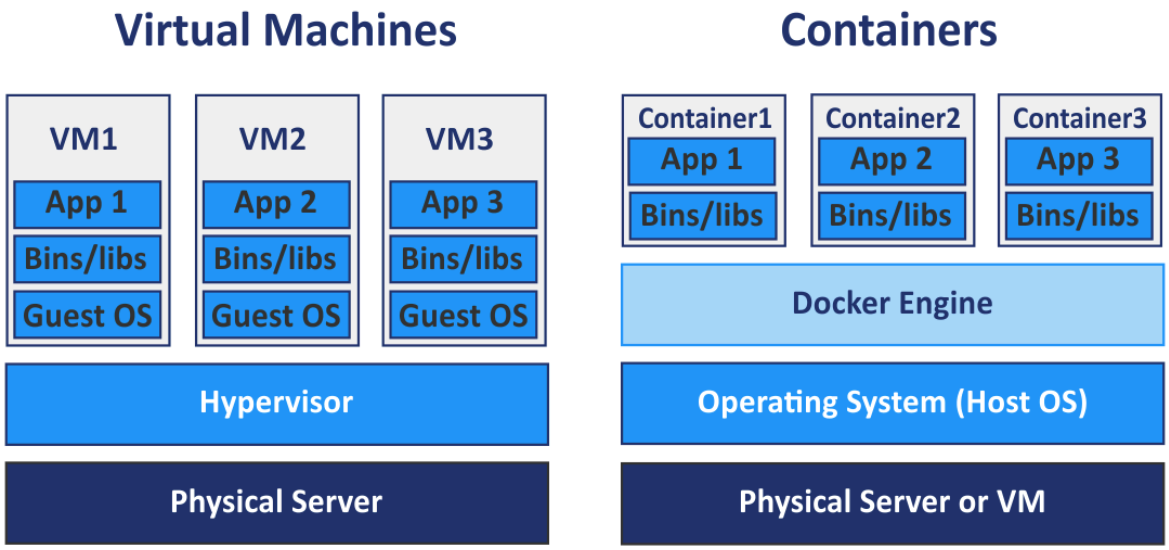
3.1 Docker and Containers

In the context of the project, the infrastructure provided by the CS cloud [23] is being utilized extensively. Docker, a well-known open-source platform, stands as the backbone, enabling developers to develop, deploy, manage, and update containers efficiently. These containers are standardized executable components that seamlessly integrate application source code, the operating system (OS), and all the necessary libraries and dependencies. This amalgamation ensures the code runs in various environments without any issues. The advantages of Docker are not restricted to developers; they extend to system administrators, making it an integral element of DevOps toolchains.

Containers, as illustrated in Figure 3.1, come into existence using customized images stored in both private and public container registries such as Docker Hub [24] and Google Container Registry [25]. Virginia Tech also maintains its proprietary container registry. These containers are pivotal in utilizing process isolation, and virtualization features inherent in the

Linux kernel, such as control groups (cgroups) and namespaces, which facilitate sharing OS resources amongst multiple application components. This is analogous to how a hypervisor allows various virtual machines (VMs) to share the resources of a single hardware server. The outcome is a swift and consistent deployment of applications, irrespective of the deployment milieu, promoting the broad adoption of container technology.

Figure 3.1: Container is on top of OS [26]



Our role in the DevOps team involves creating, deploying, and maintaining containers, aligning with the individual requirements of each team. To tailor to the specific needs of every team, we assist and require each team to construct a Dockerfile, a key element for swift deployment. A Dockerfile is essentially a script containing all the commands a user might invoke on the command line to build an image. Within the realm of the VT CS cloud, our deployment of all containers is executed on the Endeavour cluster.

3.2 Rancher

Rancher [27] is an open-source platform designed to orchestrate multi-cluster operations, resolving the administrative and security hurdles encountered when overseeing multiple Kubernetes clusters, regardless of the infrastructure. It facilitates the deployment of Kubernetes clusters and integrates them through centralized authentication and access control. It is instrumental in managing Kubernetes clusters and offers a beneficial UI for supervision and management.

The pivotal features that elevate Rancher's utility include:

1. **Authentication and Authorization** - It is crucial for teams to manage users effectively to confine administrative access to authorized individuals, ensuring that developers do not execute actions beyond their assigned tasks, such as modifying a cluster or removing a worker.

2. **Security Compliance** - Production-level Kubernetes clusters are required to uphold optimum security standards. Rancher distinguishes itself by offering a feature that scrutinizes the established clusters and conducts tests to evaluate their compliance with benchmarks.

3.3 Camelot

In Endeavour, we use Camelot as the primary storage infrastructure for Electronic Theses and Dissertations. While CEPH serves as a comprehensive open-source, software-defined storage system that operates as object, block, and file storage within a unified framework, it's important to note that the CEPH file system doesn't reside on the same private network as the Endeavour cluster [28].

Chapter 4

Requirements

4.1 General Requirements

This project revolves around the search functionality for Electronic Theses and Dissertations. Here's a concise overview:

The system will conduct searches on Electronic Theses, Dissertations and related digital objects. Searches can be performed on all the documents and their metadata. To enhance the search process, methods like classification and object detection will be applied. Additionally, there will be workflows for summarizing individual chapters. Data pre-processing will be employed, including tasks like identifying parts of speech and stemming. The utilization of a scalable indexing engine is preferred to ensure optimized information retrieval. Actual files will be stored in a file system, while metadata will be housed in a database. Access to data will be controlled at appropriate levels to maintain security and integrity. The project components, including web servers, databases, and Machine Learning models utilizing GPUs, will be integrated. Continuous Integration/Continuous Deployment (CI/CD) practices will

be employed for automated code deployment, along with implementing automated test cases through pipelines.

4.2 Team Requirements

Team 5 plays a pivotal role in ensuring seamless collaboration and functionality across all teams involved in the project. We are tasked with overseeing the operation of containers and APIs, ensuring they work efficiently in conjunction with other teams' components. This involves handling the deployment of containers and setting up APIs to access the database and file system. The first task was to set up Camelot to cater to all persistent storage needs. This includes connecting volumes to external locations, ensuring seamless accessibility to critical data, configuring the container cluster, and establishing pods, containers, and tailored environments to accommodate all teams' requirements. For this, we migrated deployments from the old cluster, named "discovery", to the new teaching cluster, named "endeavour", where we updated the dependencies between each deployment. This is crucial for Version 1 to be operational. Looking ahead, for Versions 2 and 3, the team must implement a robust file system organization, database schema, and content management system, facilitating seamless batch data communication using Kafka among containers and between teams. Thoroughly testing each team's containers within the cluster is crucial, ensuring optimal individual functionalities. Ultimately, the team's success will be measured by their ability to integrate and harmonize all teams' containers into a unified system supported by a robust CI/CD pipeline. This holistic approach will be instrumental in achieving the project's objectives effectively and efficiently.

4.3 User Personas

1. Team 1:
    - As an engineer, I want to define and maintain meaningful relationships between various data entities.
    - As a developer, I want to query the knowledge graph efficiently, pulling insights to be passed to the front end.
2. Team 2:
    - As an engineer, I want to effectively index vast amounts of data to ensure quick and accurate search results.
    - As an end-user, I want accurate search results returned quickly, with relevant recommendations and filters.
3. Team 3:
    - As a scientist, I need access to a platform with resources to perform object detection and topic modeling using Machine Learning models.
    - As a scientist, I need access to the training dataset for my Machine Learning model from the database.
    - As a scientist, I need access to the persistent file storage system and the database to store the results of the trained models.
4. Team 4:
    - As a scientist, I need access to the chapter text.
    - As a scientist, I need access to a platform with resources to summarize and categorize.

- As a scientist, I need access to the persistent file storage system and the database to store the results.

5. Team 6:

   - As an engineer, I want to deploy an intuitive front-end interface that provides users with clear navigation and smooth interactions.
   - As an engineer, I want to integrate with the back-end APIs efficiently, ensuring real-time data fetch and display.

Chapter 5

Design and Implementation

5.1 System Architecture

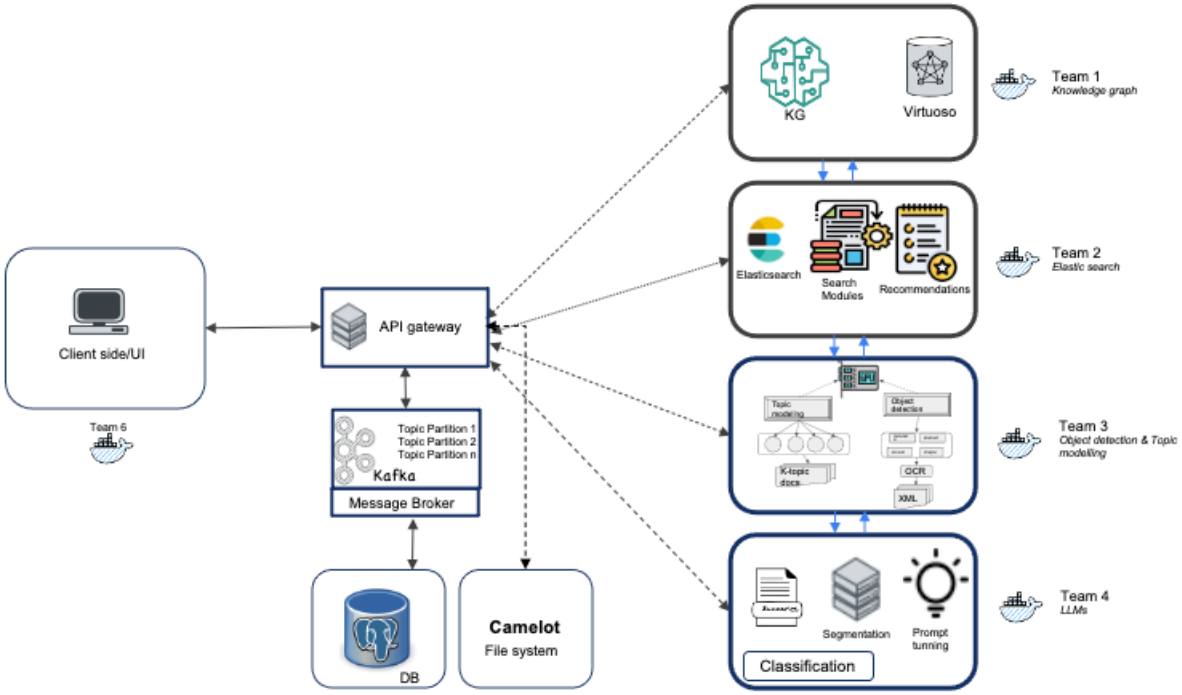


Figure 5.1: Overall system with each team's components interacting via APIs

The system architecture is essential for understanding how the different components interact. Based on refined requirements from the project objectives, the architecture differs from the previous implementation [21]. The design of the overall system is a Microservice architecture, and each team's service is a suite of independently deployable modular services, all communicating through an API gateway to other services as illustrated in Figure 5.1. The goal of the architecture is to isolate the various components, making them independent and reinforcing the architecture's security and organization, which is visually described in Figure 5.1.

5.1.1 System Overview

- Team-1 is responsible for managing the ETDs and overseeing the knowledge graph construction. To facilitate the knowledge graph team's data ingestion and channel data into the knowledge graph, an API service [29] will be implemented.

- Team-2 is tasked with querying various data sources, extracting pertinent information, and subsequently processing this data. The results are then made accessible via a search engine, specifically Elasticsearch [30], to deliver refined search outcomes. The Elasticsearch team also captures user interactions from searches and clicks and, therefore, logs the details for log analysis and recommendation.

- Team-3 has two objectives. Firstly, the service interfaces with the ETD data provided via an API [29]. Secondly, they leverage the data to train Machine Learning models. One set is for topic modelling, while the other employs object detection algorithms to identify and categorize elements.

- Team-4 has objectives that encompass applying training to Machine Learning models for classification, summarization, and prompt tuning. These subsequently commit the

outcomes to the storage framework as well.

- Team-5: The integration team orchestrates all interactions through the API [23]. Database engagements and intra-team service interactions are channelled exclusively via API calls. Additionally, team 5 manages the continuous integration and delivery mechanisms for streamlined deployments to the cluster.

- Team-6: The front-end team is tasked with crafting a uniform user interface and overseeing the design and presentation of all user-facing screens. They are charged with delineating the procedural flow for all user engagements and can retrieve and display results sourced from the backend.

- Universal Objectives: Every team is mandated to employ version control [31], ensuring consistent maintenance and crafting test routines for their respective services. Furthermore, each is responsible for encapsulating their services within containers [24] for streamlined deployment.

5.1.2 Detailed Design

It is crucial to grasp the holistic view of the system and its services, including a detailed understanding of how individual components behave and interact with each other. This understanding is exemplified in Figure 5.2, depicting the process flow architecture from development to deployment and the various components utilized.

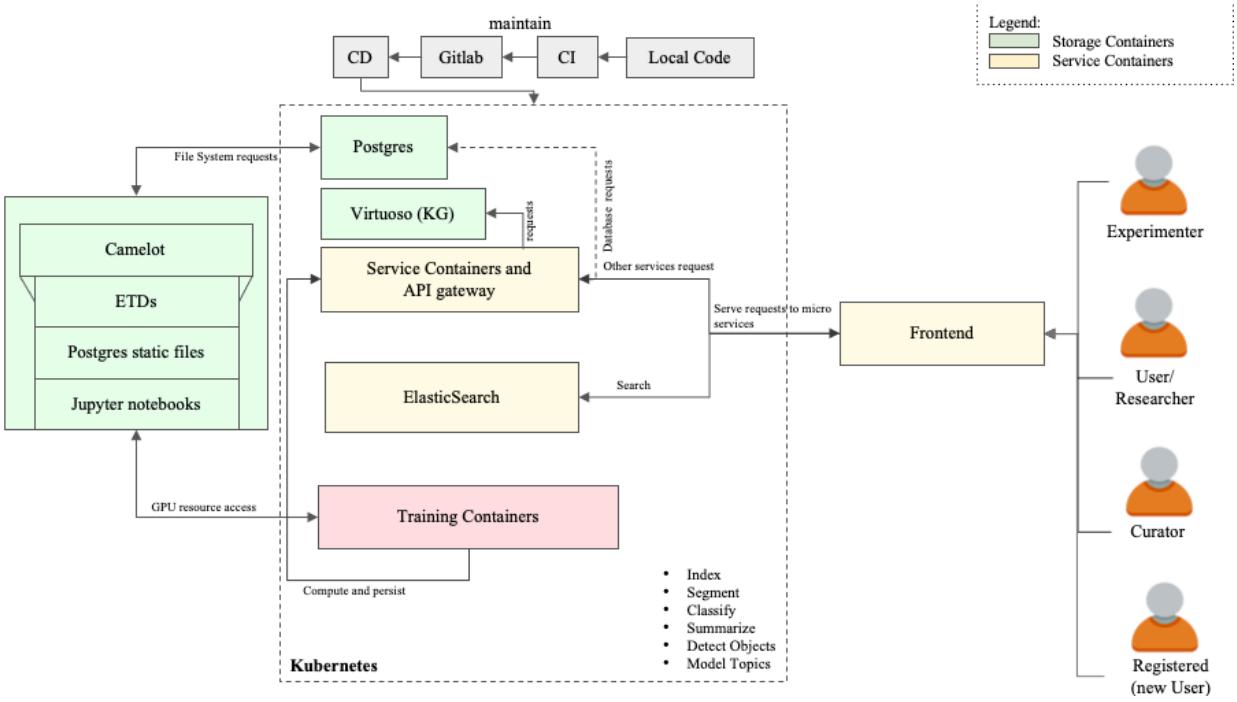


Figure 5.2: Overall system and process flow

Upon examining the comprehensive overview of the systems, we discerned shared high-level objectives across the teams. Consequently, we pinpointed four distinct focus areas:

1. Docker and Container: Each team's service is encapsulated within a container [24], ensuring dependency resolution and facilitating seamless deployment to the cluster. The class repository at code.vt.edu [32] maintained by Virginia Tech's GitLab instance hosts the container images corresponding to each team's service.

2. Continuous Integration and Continuous Delivery (CI/CD): Teams utilized GitLab as their primary Version Control System (VCS) [31] to maintain, test, and deploy their respective services. We integrate supplementary CD tools to enhance our deployment processes further, enabling streamlined application code deployment to the production environment.

3. File System: All Electronic Theses and Dissertations (ETDs) are stored in the designated file system called Camelot, which is hosted on camelot.cs.vt.edu. The Computer Science department at Virginia Tech manages this file system. Our team have access to this resource through designated internal network IPs, and access is facilitated by provided credentials, ensuring compliance with access controls.

4. Kubernetes and Cloud Infrastructure: We've delineated four distinct container categories: storage (specifically, Postgres), service, and training.

    - Storage Containers: These specific containers are dedicated to provisioning the RDBMS services. Their primary role is to house metadata and the results of trained models.
    - Service Containers: Predominantly, these containers cater to business logic, delivering it through web payloads, exemplified by formats like JavaScript Object Notation (JSON) [33] and Extensible Markup Language (XML). [34]. Beyond that, they facilitate the presentation of front-end web pages to the end-users. A significant part of their function also encompasses rendering search responses from engines, notably Elasticsearch [30], in addition to processing and indexing data pertinent to recommendations and the construction of knowledge graphs.
    - Training Containers: These containers are explicitly engineered to undertake the training of Machine Learning models, utilizing the Electronic Thesis and Dissertation (ETD) data.

5.1.3 System Components Overview

Frontend Design: The system's frontend is built by the user interface team using React.js, a component-based architecture that allows for efficient and flexible development of interactive UIs.

Backend and API gateway: The backend is developed using Flask, a lightweight and flexible Python web framework. Flask serves as the backbone, handling HTTP requests and managing database interactions. We use an Object-Relational Mapping (ORM) tool, which is used for database operations, abstracting complex SQL queries into Python code. **Microservices**:

- **Database and File System**: The database is an SQL Postgres database running on a container, in which the Postgres database is accessed via the API gateway. The file system is also mounted on the Camelot server and integrated into the containers on the cluster as well.
- **Other Teams Services**: Each team service is a modular unit deployed as a container as a service, of which the API gateway serves as the focal point for interacting with the various other services.

5.1.4 API Design

The system adopts RESTful principles for the API endpoints, facilitating clear and standardized communication for the frontend, backend and other team services. The REST architecture enables stateless interaction, scalability, and ease of integration. Key characteristics of our API endpoints are:

- Endpoint Naming: Each endpoint is named according to the resource it handles (e.g.,

/auth/login for authentication, /store-object for object storage). This naming convention offers clarity and predictability.

- HTTP Methods: Appropriate HTTP methods are used to signify the action performed (GET for data retrieval, POST for data creation, PUT for data updating, DELETE for data deletion).

- Status Codes: HTTP response codes are used to represent the outcome of API requests, aiding in error handling and debugging (e.g., 200 for success, 401 for unauthorized access).

API Development

Given the large number of APIs, they are grouped based on their functional areas for better clarity.

Authentication APIs: These are the user authentication APIs for access into the system. The API includes endpoints like /auth/login and /auth/register. The API endpoints are listed in Table 5.1.

API	Description	Request Method	Status
auth/register	API to create a new user and store user details in db	POST	Implemented
auth/login	API for login authentication	POST	Implemented

Table 5.1: Implemented Authentication API Endpoints

User Management APIs: Endpoints for user profile management and role assignment (e.g., /user/profile, /admin/users). The API endpoints are listed in Table 5.2.

API	Description	Request Method	Status
/user/<update>	API to update a user profile by user ID	PUT	Implemented
/user/<profile>	API to fetch a user profile by user ID from db	GET	Implemented
/user/admin	Get all users currently registered on the system	GET	Implemented
/user/update-user-role	Update current user role	PUT	Implemented

Table 5.2: List of Implemented User Management API Endpoints

ETD Management APIs: APIs related to the retrieval of ETDs, processing metadata and file storage (e.g., /etd-metadata, /store-xml, /save-page-image). The API endpoints are listed in Table 5.3.

API	Description	Request Method	Status
/services/etds/<int:etd-id>/pdf	To get the ETD document stored in the file system and the ETD ID corresponding to it. (ETD table)	GET	Implemented
/services/etds/<int:etd-id>/	API endpoint to Retrieve ETD, checks for ETD ID in the file system directory, cross-checks with database and returns ETD information and path.	GET	Implemented
/services/etds	API endpoint to retrieve all ETD info	GET	Implemented
/team3/etd-metadata	POST to ETD-metadata table	POST	Implemented
/common/etd-classes	API endpoint retrieve a list of all ETD classifications.	GET	Implemented
/common/etd-classes	Add to ETD-classes to save the results	POST	Implemented
/common/etd-classes/<int:etd-id>	Retrieve a specific ETD classification by ETD ID.	GET	Implemented
/team3/etd-metadata	POST to ETD-metadata table	POST	Implemented

Table 5.3: List of API Endpoints for ETD data management

Other API Endpoints: Endpoints used by the Object detection, Topic modelling team, large language models team, ElasticSearch team, and Knowledge Graph team. Other related data manipulation endpoints are listed in Table 6.1.

5.2 Database

The database image in Figure 5.3 represents version 1 of the PostgreSQL database, which contains various tables and their associated schema. In the 'public' schema, a number of tables have been constructed, each with specific attributes and constraints.

Figure 5.3: Version 1 of the PostgreSQL database

The 'class' table holds data pertaining to classifications, while 'etd' is a table for electronic theses and dissertations, with related metadata kept in 'etd_metadata'. The 'object' table handles objects with a local path and an associated ETD_ID. Tables like "object_classification" and "object_metadata" are used to hold different kinds of object-related metadata. The 'summarisation' table contains the summaries of objects. The database also includes sequences (such as "class_id_seq," "etd_id_se," and others) that are used to provide distinctive IDs for records in their respective tables. Foreign key restrictions that create

connections between tables are also present in the database. ‘etd_metadata_etds’, for example, creates a foreign key from ‘etd_metadata’ to ‘etd’, connecting metadata entries to the appropriate electronic theses or dissertations. Similar links are built for object classifications, object metadata, summarizations, and topic sets.

Version 2, as shown in Figure 5.4, was carefully created, and we established a comprehensive database schema.

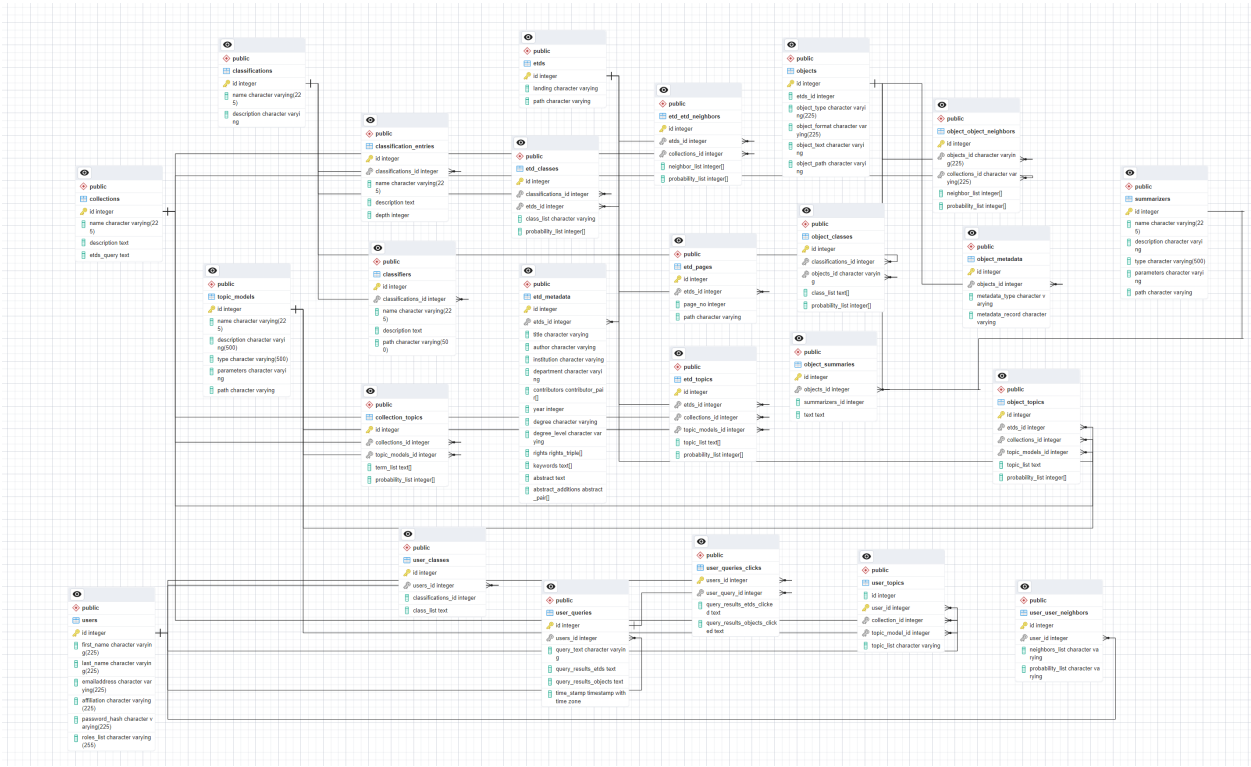


Figure 5.4: Version 2 of the PostgreSQL database

TABLE NAME	DESCRIPTION
Classifications	Classification schemes used to categorize ETDs and Objects, e.g., ProQuest
Classification_entries	Entries in a Classification
Classifiers	Models that generate Classes for an ETD or Object relative to a Classification
Collections	Sets of documents, used for processing, experimentation, topic modeling, etc.
Collection_topics	Set of topics resulting from applying a Topic_model to a Collection
ETDs	Electronic Theses or Dissertations in our corpus
ETD_classes	Assignment of categories from a Classification of a Collection
ETD_metadata	Metadata for an ETD
ETD_pages	Page images of the pages of an ETD
ETD_topics	Topics resulting from use of a Topic_model applied to an ETD
ETD-ETD_neighbors	Set of nearest neighboring ETDs for a given ETD
Objects	Objects resulting from applying an object detection or generation process to an ETD
Object_classes	Set of classes assigned by a Classifier to an Object
Object_metadata	Metadata for an Object
Object_summaries	Summaries for an Object
Object_topics	Topics for an Object
Object-object_neighbors	Set of nearest neighboring Objects for a given Object
Summarizers	Models that generate Summaries for texts
Topic_models	Models that generate Topics for a Collection
Users	Persons using our system
User_classes	Entries in a Classification that are of interest to a particular User
User_queries	Queries issued by a User
User_queries_clicks	Click information of ETDs and Objects returned to a User for a Query
User_topics	Topics of interest to a User, for a given Collection, from Collection_topics
User-user_neighbors	Set of nearest neighboring Users for a given User

Figure 5.5: List of tables in Version 2 database

Figure 5.5 shows a list of 25 tables implemented in the database, each designed to perform a specific task, from storing Electronic Theses and Dissertations (ETDs) to managing users and their interactions with the system. We chose the primary keys for each table to assure record uniqueness and set up relationships across tables using foreign keys to maintain referential integrity. We selected the data types and constraints to reflect the recorded information accurately. It includes tables like 'Topic-models' and 'Summarizers' to handle topic models and summarization methods and 'Objects', 'Object-object-neighbors', and 'Object-topics' to manage relationships and associations for object-related data. Additionally, tables like 'Object-summaries' and 'Object-metadata' store summarization results and metadata,

while ‘Object-classes’ handles class associations. This was created mainly because it is better suited for our application due to its granular data management, specific data handling capabilities, and support for more specialized functionality. It provides a structured approach for managing topics, summarizations, classifications, and user-related data, indicating a focus on detailed metadata management and summarization processes.

The ETDs table shown in Figure 5.6 and ETD_metadata table shown in Figure 5.7 were loaded with a dataset comprising of 500K entries from a file provided called etd_500k.csv, using a script, the details of which are elaborated further in the Developer Manual.

id [PK] integer	landing character varying	path character varying
10169	https://escholarship.org/uc/item/1bc2m5q8	001/0169
10170	https://escholarship.org/uc/item/1gd702sc	001/0170
10171	https://escholarship.org/uc/item/0pw2b0fc	001/0171
10172	https://escholarship.org/uc/item/0r4720nz	001/0172
10173	https://escholarship.org/uc/item/0ks7c3	001/0173
10174	https://escholarship.org/uc/item/0wx3x8kw	001/0174

Figure 5.6: ETDs Table of database version 2

id [PK] integer	etds_id integer	title character varying	author character varying	institution character var	department character var	contributors contributor_p	year character var	degree character var	degree_level character var	rights rights_triple[]	keywords text[]	abstract text	abstract_add abstract_pair
0	1436	Plasma Diagnostics and Plasma-Surf…	Titus, Monica Joy	ucb		[null]	2010		[null]	[null]	[null]	[null]	[null]
1	1437	Declarative Systems	Condie, Tyson	ucb		[null]	2011		[null]	[null]	[null]	[null]	[null]
2	1438	Portrait of the Rugged Individualist: T…	Horberg, Elizabet…	ucb		[null]	2010		[null]	[null]	[null]	[null]	[null]
3	1439	Essays in Empirical Macroeconomics	Nelson Mondrago…	ucb		[null]	2015		[null]	[null]	[null]	[null]	[null]
4	1440	Control and Trajectory Generation of …	Swift, Timothy Alan	ucb		[null]	2011		[null]	[null]	[null]	[null]	[null]
5	1441	Errors as a Productive Context for Cla…	Leveille Buchana…	ucb		[null]	2016		[null]	[null]	[null]	[null]	[null]
6	1442	Interrogating the Role of Spatial Orga…	Nair, Pradeep	ucb		[null]	2010		[null]	[null]	[null]	[null]	[null]
7	1443	Hydrodynamic Exchange in Estuarine …	HSU, KEVIN KAI-…	ucb		[null]	2013		[null]	[null]	[null]	[null]	[null]
8	1444	An Environmental and Economic Trad…	Robinson, Stefanie	ucb		[null]	2013		[null]	[null]	[null]	[null]	[null]
9	1445	Essays on Markets and Institutions in…	GHANI, TAREK FO…	ucb		[null]	2015		[null]	[null]	[null]	[null]	[null]

Figure 5.7: ETD_METADATA Table of database version 2

5.2.1 Data Analysis and Visualization

An examination was performed on the etd_metadata table to evaluate data integrity by identifying instances of null or empty values and assessing the length of the data fields.

1. Assessment of Null and Empty Values: A thorough review of null and empty values across various columns within the ETD_metadata table was performed. This analysis identified instances of missing data, assisting in recognizing potential issues related to data completeness.

   The resulting bar chart in Figure 5.8 visually illustrates the count of null and empty values in each column of the ETD_metadata table, providing an overview of data completeness.

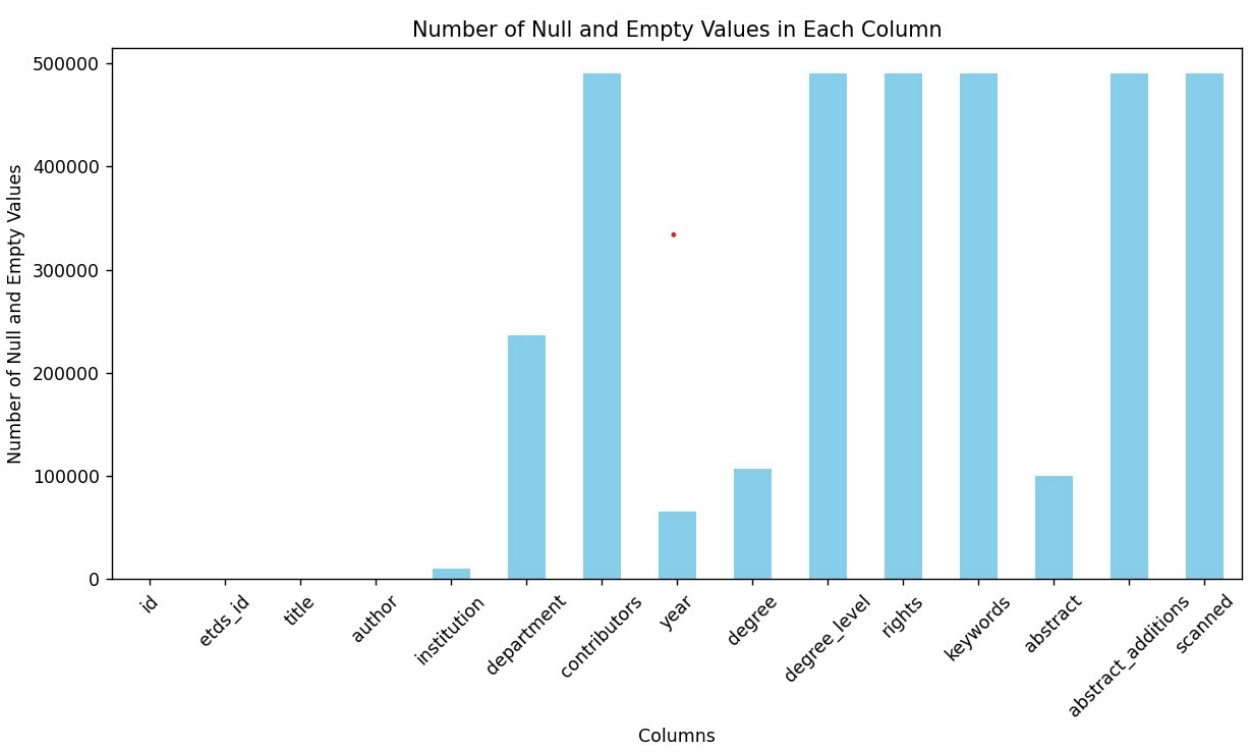


Figure 5.8: Histogram of counts of null/empty values in ETD_metadata columns

2. Text Length Distribution Analysis: Analysis of text length distributions was carried out on the 'title' and 'abstract' columns. The histogram in Figure 5.9 displaying the distribution of title lengths across records offers insights into the variability of title lengths within the ETD_metadata table. Similarly, the histogram demonstrating the distribution of abstract text lengths within specified ranges provides an understanding of the distribution of abstract content lengths.

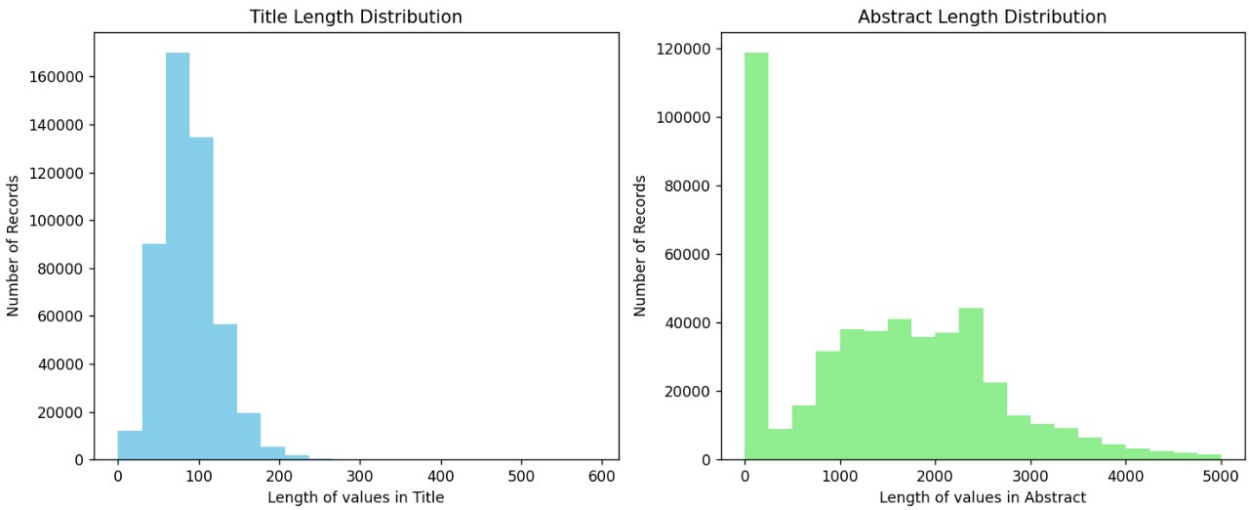


Figure 5.9: Histograms demonstrating the distribution of title and abstract text lengths

5.3 File System

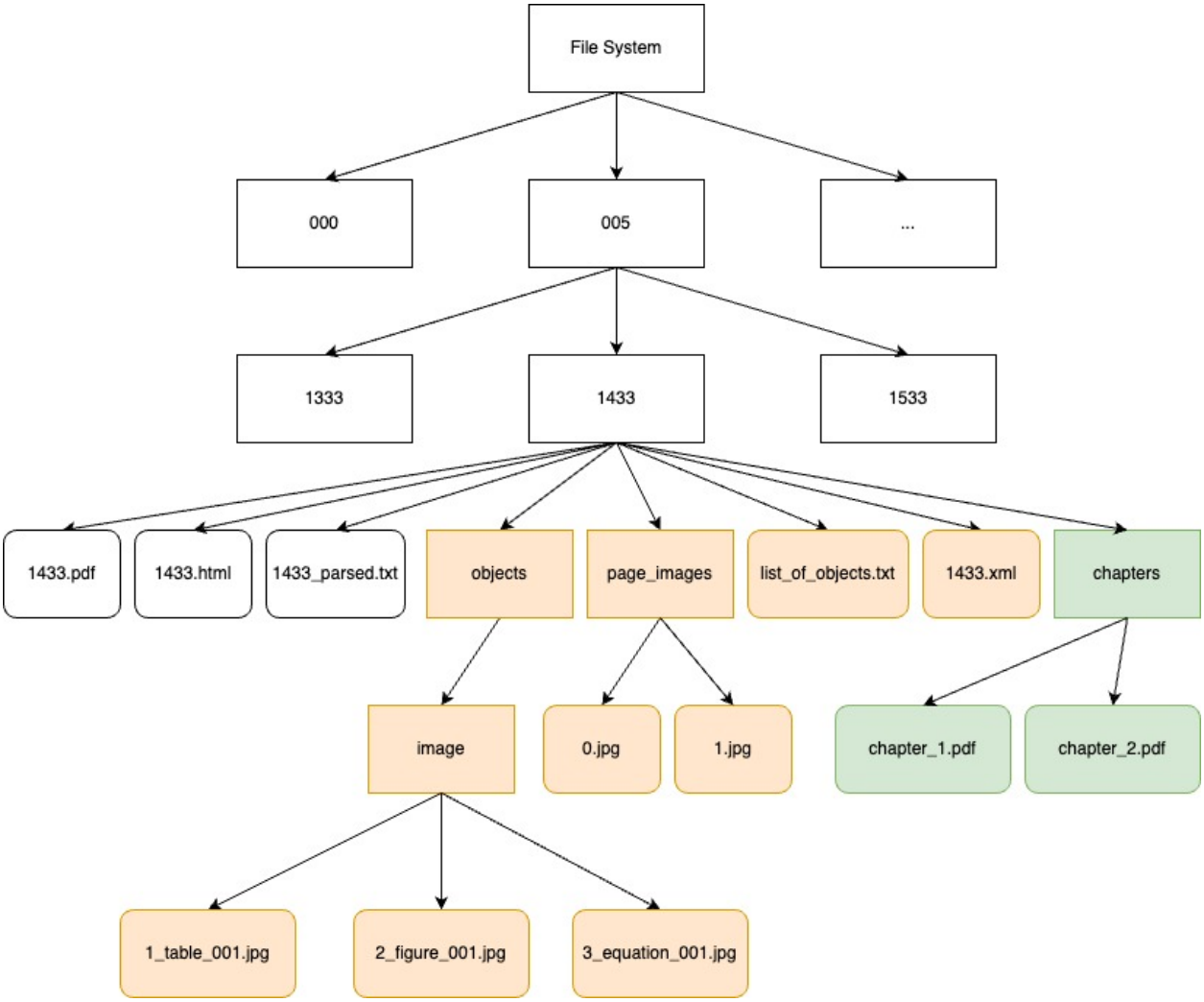


Figure 5.10: The current structure of the file system

Our system utilizes a directory structure similar to a tree for organizing the file system, specifically for managing Electronic Theses and Dissertations (ETDs). Each ETD is identified by a unique numerical code, with the initial numbers indicating the primary folder and a sequence of digits denoting the specific ETD. This structure is designed to efficiently arrange and maintain these files, encompassing elements like chapters, page images, and detected objects contributed by diverse teams.

Located on the Camelot server, within the "etdrepo" directory, this file system can be integrated seamlessly into containers. The structure begins with the main "etdrepo" directory containing numerous subdirectories, each marked by a unique three-digit code. These represent the first layer of the file system, and every subdirectory houses thousands of ETD-related folders. These folders comprise an assortment of digital files, including PDFs, XML, HTML, and plain text. The PDF files are the actual ETD documents, while XML files are outputs from Team 3's object detection workflow. HTML files are derived from these XML files and "parsed.txt" files. The "parsed.txt" files are older clean text files that were already present in the 500k file system. They were not generated or modified by any of the teams for this project and are available for certain ETDs.

We also designate specific directories for each type of object classified by Team 3, such as images, allowing these to be stored efficiently within the corresponding ETD folder. Our file system facilitates interactions for all teams involved, providing APIs for data management, as shown in Table 5.4. These APIs enable teams to input and extract data from both the file system and the associated database as per their project requirements, as detailed in Table 6.1. The layout and structure of this file system are depicted in Figure 5.10, providing a clear overview of its current setup.

API	Description	Request Method	Status
etds/<int:etd_id> /pdf	Get ETD IDs of the .pdf file	GET	implemented
etds/<int:etd_id>	Get ETD with ETD ID	GET	implemented
retrieve-page-image /< int:page_id>	Retrieve page image with pageID	GET	implemented
save-page-image	Save page image (page of .pdf)	POST	implemented
store-xml	Store the generated XML file	POST	implemented
store-object	Store the detected objects	POST	implemented

Table 5.4: List of APIs interacting with the file system

5.4 Inter-Team Integration

Defining the system architecture was pivotal for laying out a detailed implementation roadmap and also for comprehending how the teams synchronized their efforts and ensured smooth information flow. Here's how we strategized our support:

File System and Database Access: Teams 1, 3, and 4 are provided access to both the file system [28] and the database through APIs, so the teams can readily access data or ETDs from the file system or database. We have listed the APIs currently tested and implemented system-wide for various operations on retrieving data or processing data from the database or file system.

Main API Integration: We orchestrated the provisioning of APIs [29] to cater to a diverse range of services across different teams:

1. Knowledge Graph Team (Team-1): Facilitated access to object tables for the knowledge graph team.

2. ElasticSearch Service (Team-2): Served as the conduit for distributing user logs and for the ElasticSearch team to efficiently handle search queries [30].

3. Topic Modelling & Object Detection (Team-3): Enabled seamless access to ETDs and facilited the storage of metadata and object information within the database.

4. Large Language Model Team (Team-4): Provided data access to the LLM models.

5. Front-End Interaction (Team-6): Ensured a smooth and responsive user interface experience by driving the API's interaction [29] with the front end. Through this integrative approach, we're harmonized the functionalities of Teams 1, 2, 3, 4, and 6, fostering a collaborative and efficient ecosystem.

Continuous Integration and Deployment: About the workflows of each team, we have completed the supervision of the continuous integration and delivery process. This included sharing a sample script for the teams to utilize, the encapsulation of services in containers, and the coordination of deploying code that serves as a foundation for the work of other teams.

5.4.1 API authentication

API authentication is a critical component of overall system security, ensuring that every request is verified and the identity of the requesting system, user, or entity is validated before granting access to data or features. The authentication acts as the gatekeeper, ensuring that only fully authenticated requests are processed. The team has ensured best practices to ensure secure and efficient authorization measures.

- Security: Protecting sensitive data from unauthorized access.
- Data Integrity: Ensuring that only authenticated users can access data.

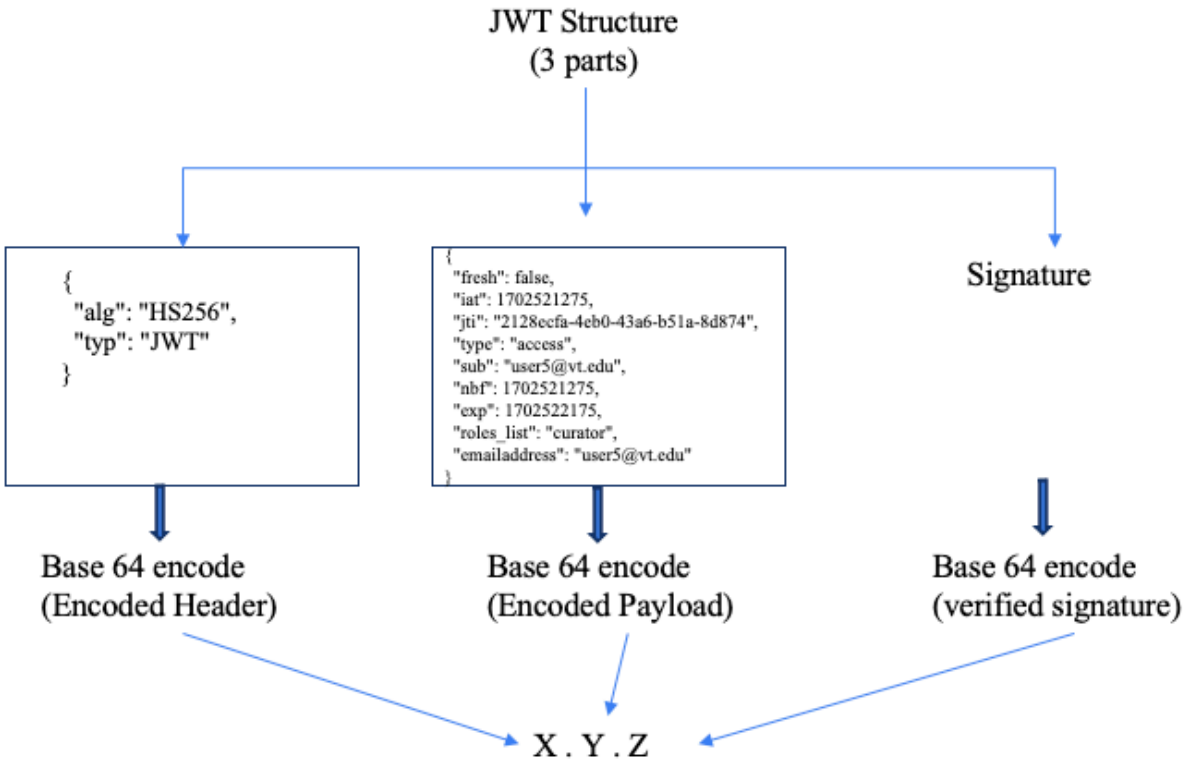


Figure 5.11: JWT three-part structure

JWT Authentication

The team has implemented a token-based authentication using JSON Web Tokens (JWTs) in the API gateway. JWTs are an open standard defining a compact and self-contained way to securely transmit information between parties as a JSON object [35]. The JWT structure contains three parts, all encoded in base64: the header, the payload, and a verified signature stored in the user's browser's local storage. The JWT structure is illustrated in Figure 5.11.

Workflow:

1. The user provides their credentials (username and password) in a request to an Auth system [36] built into the API gateway.

2. The Auth system [36] authenticates users and generates a JSON Web Token (JWT) containing user details, including roles and permissions that determine access to specific endpoints and features.

3. The JWT token is subsequently sent back to the client, where it is typically stored in either local or session storage as described in Figure 5.12.

4. For subsequent requests, the token is included in the headers. A payload containing a signature must be validated with a secret configured on the Endeavour cluster, allowing the user in that session to access protected routes and resources.

Advantages of JWT:

- Compact: The tokens can be sent through URL, POST parameter, or inside HTTP header.

- Self-contained: The payload contains all the required information about the session and user, avoiding the need to query the database more than once.

Secure Handling of Passwords:

The team has adhered to best practices for securing all passwords and secrets by utilizing environment variables while user credentials are stored securely:

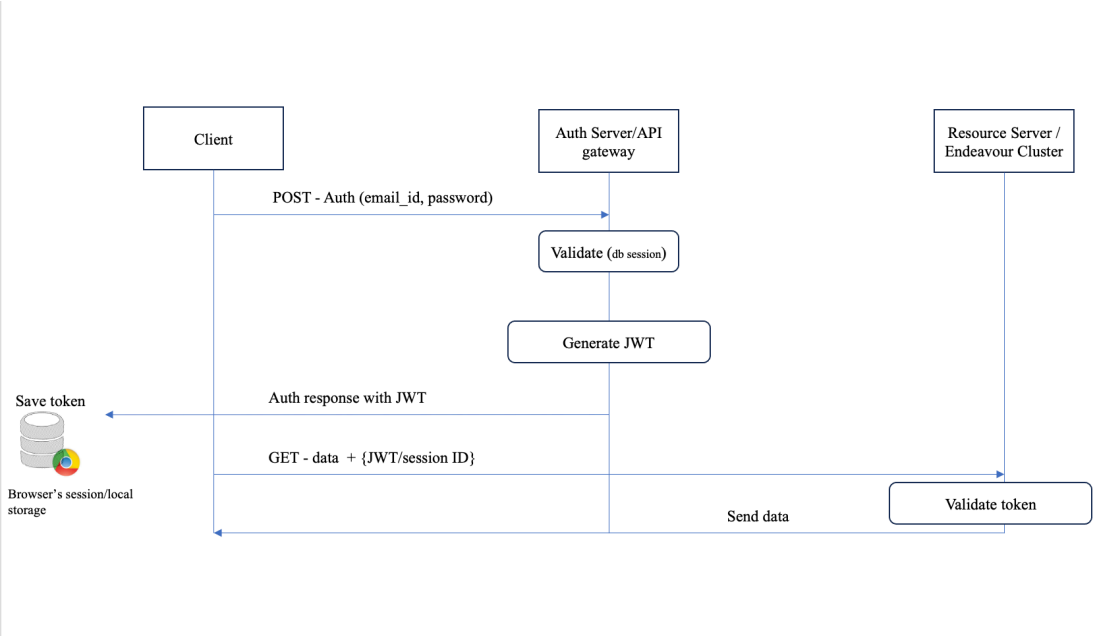


Figure 5.12: Authentication process flow

- **Hashing**: We use a hashing library provided by Werkzeug Utils [37] (a Flask library) to encrypt passwords. Hashing turns a password into a string of characters that's not reversible, ensuring original passwords can't be deciphered.

5.4.2 Securing Endpoints and Authenticating All APIs

Securing API endpoints is crucial for protecting sensitive data and ensuring reliable service operations [38]. We have considered two robust approaches for API security: JSON Web Tokens (JWTs) and API keys. While both methods offer strong protection, their applications and implications differ considerably. As described above, JWTs were implemented for user authentication.

Choosing Between JWT and API Keys for Securing Other Endpoints

In deciding between JWTs and API keys, we considered the specific needs of the application:

Use JWTs for:

- The current token generation process is cumbersome during testing, as each test requires the creation of new tokens. While this process will be automated in production, it remains complex during testing.
- The primary use case for JWTs is in scenarios that demand fine-grained access control and where the payload varies across users. In this regard, we have effectively utilized JWTs for user authentication, catering to the different personas (User, Experimenter, and Curator) present within the system. However, implementing JWTs for every API endpoint would be an excessive measure.

Use API Keys for:

- Ease of use for the respective teams throughout all project phases, including requirement gathering, blueprint design, and testing.
- The API keys effectively fulfil the requirement of endpoint security, verifying that requests originate from authorized sources (other services implemented by the other teams).
- API keys have demonstrated their advantage in terms of implementation speed and simplicity in instances where other teams require access to various API endpoints or modifications to these endpoints.

API Keys Overview

API keys are simple, long-lived tokens used to authenticate requests to an API [38]. They are easy to implement and are typically passed in the HTTP header or as a query parameter; therefore, all API endpoints are secured with this mechanism, and for user authentication, we utilize JWTs.

5.4.3 Role-Based Access Control (RBAC)

RBAC is a method of regulating access to a system or network resources based on the roles of individual users within an enterprise. In this project, RBAC was crucial to ensuring that users could only access the information and perform tasks relevant to their role, thus enhancing security and operational efficiency.

- **Roles Defined**: The system defines several user roles, including Curator, Experimenter, Registered, and User. Each role has specific permissions and access levels.

- **Role Assignment**: Users are assigned a default role when registering on the system. An administrator can later upgrade or change this role, depending on the user's needs and verification status.

- **Access Control**: The system checks a user's role by decoding the JSON web token to get the user's role before granting access to different parts of the application. Figure 5.13 shows the payload when the JWT is decoded to get the user role. For example, only Curators have access to user management, Curator screens, and all other system features, while Experimenters have access to the set of experimenter screens, such as for advanced queries.

Implementing Role-Based Access Control (RBAC) in the application involved a harmonious blend of technologies. A Flask controller orchestrates the logic on the backend, ensuring secure and efficient processing of user roles and permissions. SQLAlchemy citeSQLAlchemy seamlessly handles database interactions, ensuring data integrity and accessibility. Additionally, JSON Web Tokens (JWTs) are pivotal in securely transmitting user-role information between the client and the server, safeguarding sensitive data during transmission. On the front end, React takes the reins in managing access controls, dynamically tailoring the user interface based on the user's role as authenticated by JWTs. This synergistic combination of technologies provides a robust and scalable framework for implementing RBAC across our application, ensuring that users only have access to the resources and functionalities they are authorized to use.

Advantages of Implementing RBAC

- **Security**: By restricting access based on roles, the system minimizes the risk of unauthorized access to sensitive data.

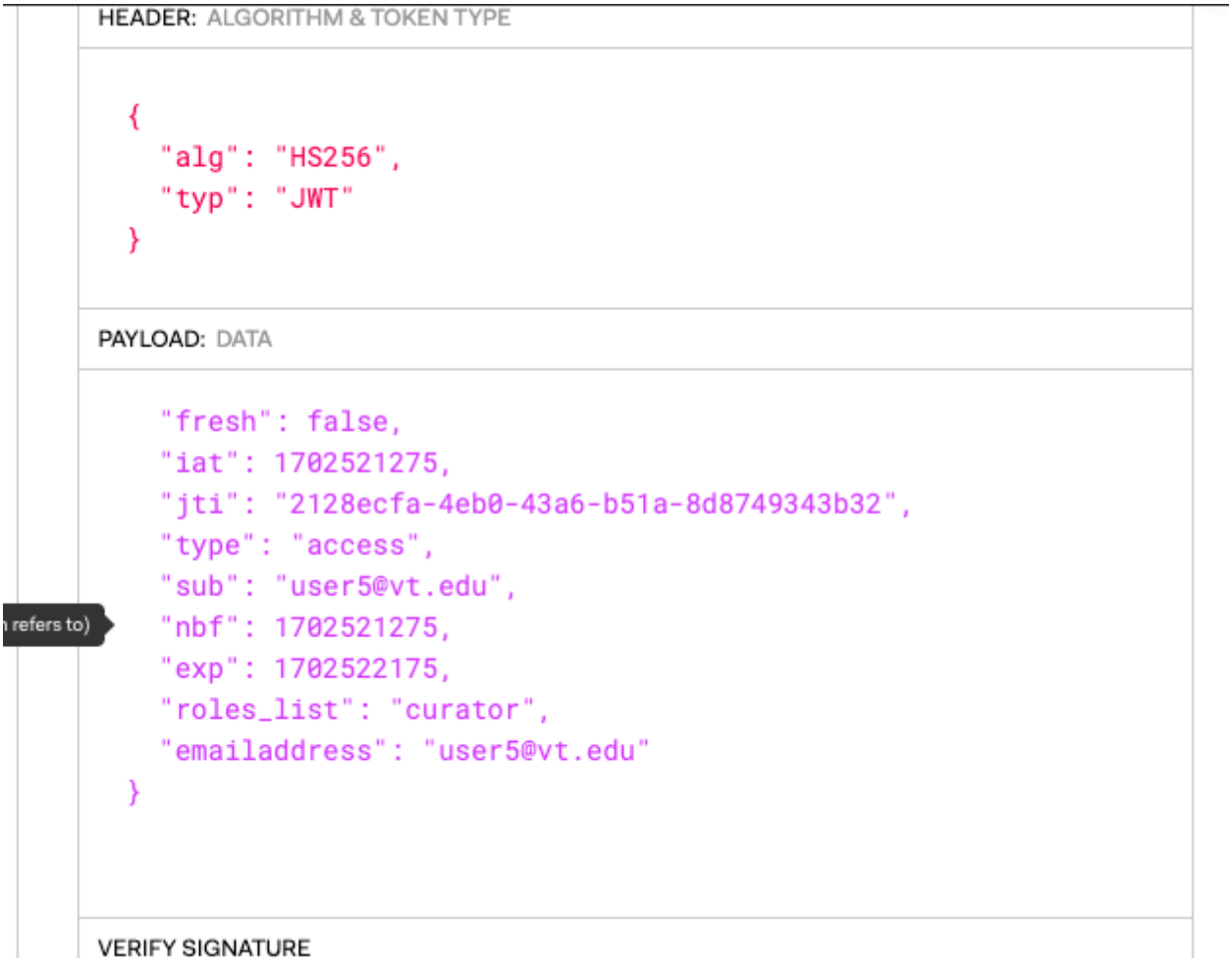


Figure 5.13: Payload of decoded JWT with user role data

- **Scalability**: As the system grows, new roles with specific permissions can be easily added to meet evolving requirements.

- **User Experience**: Users interact only with the functionalities relevant to their role, simplifying the interface.

5.5 Continuous Integration and Continuous Deployment

Since we use GitLab as our primary pipeline runner, pipelines in CI/CDs are nothing more than a piece of hardware and software that runs the provided code. GitLab's Runner [7] is open-source software that is fully featured and can create, test, and deploy, among other things. We need a virtual machine running the GitLab Runner with Docker as its executor in order to set up CI/CD for each team. The GitLab Group Runner on a Virtual Machine can be installed using the instructions in the Developer Manual in Section 6.4. We established a group runner under CS5604-F2023 [32], as each team must be accessible to the GitLab runner. Each team must maintain their code base under the group CS5604-F2023 and set up a gitlab-ci.yml file in the code to run the pipeline on the group runner.

We have built a simple pipeline script; see Figure 5.15. This script tests the connection between the GitLab repository and the GitLab Group Runner. See Figure 5.14 that shows that our Group runner is online.

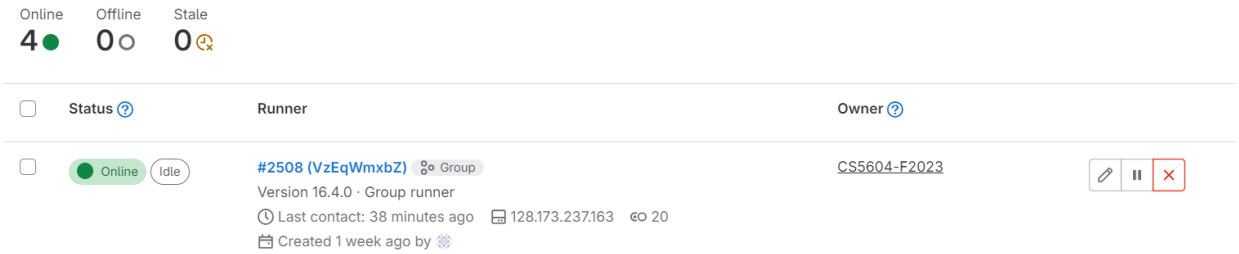


Figure 5.14: Status of Group Runner in GitLab

.gitlab-ci.yml

```yaml
stages:
  - test
  - build
  - deploy

test-job:
  stage: test
  script:
    - echo "Running tests..."
    # Add your test commands here.
    - echo "Tests completed successfully"

build-job:
  stage: build
  image: docker:stable
  rules:
    - if: $CI_COMMIT_BRANCH != 'main'
      variables:
        SHOULD_UPDATE_REGISTRY: "false"
    - if: $CI_COMMIT_BRANCH == 'main'
      variables:
        SHOULD_UPDATE_REGISTRY: "true"
  script:
    - echo "Building Docker image..."
    - docker build -f examples/Dockerfile -t code.vt.edu:5005/cs5604-f2023/team5-int/gitops/docker_image_test1:"$CI_COMMIT_SHORT_SHA" .
    - echo "Build complete"
  after_script:
    - >
      if [ $SHOULD_UPDATE_REGISTRY == "true" ]; then
        docker login -u "$DOCKER_USER" -p "$DOCKER_TOKEN" code.vt.edu:5005
        echo "Login Complete."
        docker push code.vt.edu:5005/cs5604-f2023/team5-int/gitops/docker_image_test1:"$CI_COMMIT_SHORT_SHA"
        docker logout
      else
        echo 'Registry does not need to be updated'
      fi
    - docker image prune -a -f


deploy-job:
  stage: deploy
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  rules:
    - if: $CI_COMMIT_BRANCH == 'main'
  script:
    - echo "Setting the image..."
    - kubectl -n etd set image deployment/gitops gitops=code.vt.edu:5005/cs5604-f2023/team5-int/gitops/docker_image_test1:"$CI_COMMIT_SHORT_SHA"
```

Figure 5.15: Sample .gitlab-ci.yml file containing test, build and deployment stage

Pipelines of the highest complexity can be built as YAML is infinitely scriptable.

Once we commit the YAML file, GitLab automatically runs the CI-CD pipeline. Navigate to Build(Left Panel) -> Pipelines to view your pipeline. Figure 5.16 shows the different stages

executed within the pipeline. If we hover over the stages, the UI shows the status of each stage within the pipeline.

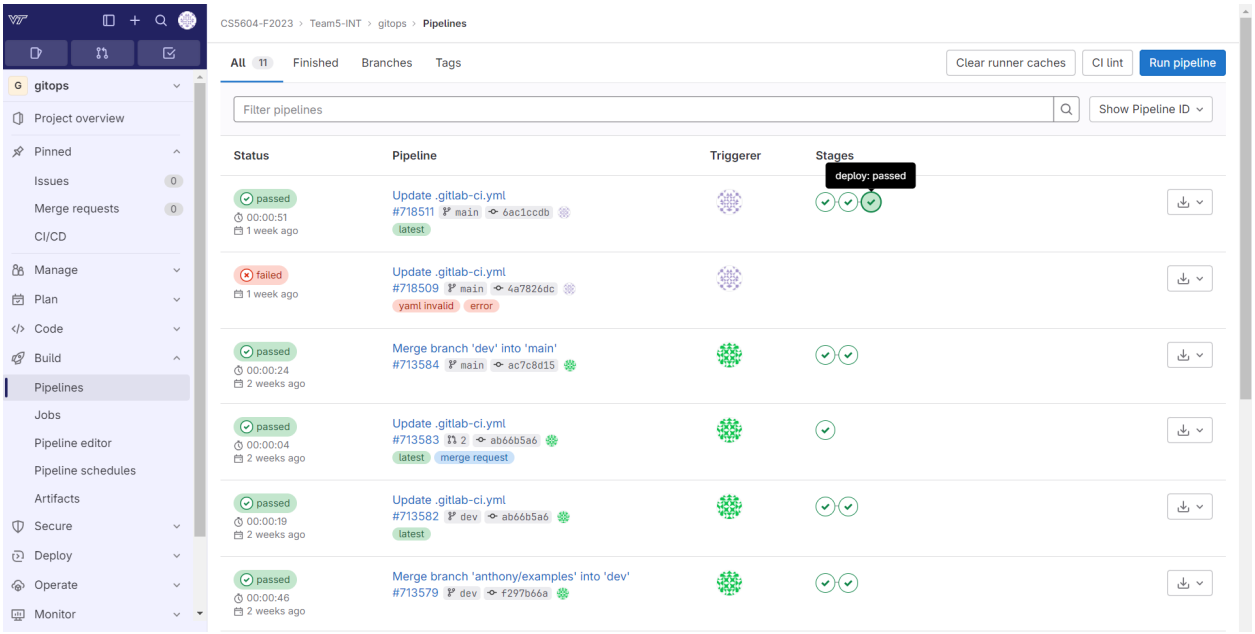


Figure 5.16: Status of the sample pipeline in GitLab

5.6 Message Queue System

Our messaging queue system is built on Kafka and Zookeeper. Zookeeper functions as a distributed storage system, wherein all Kafka configurations and messages are stored on disk. Kafka interacts with Zookeeper to manage messages across the distributed system. We only deployed one Kafka and one Zookeeper in our cluster.

Our architecture employs a single partition for each topic, simplifying the overall design. However, in scenarios of high message throughput, we have transitioned to a multi-partition structure supported by multiple consumers. Transitioning to a multi-partition design offers the initial step in augmenting our system's throughput capacity, should the increased partitions prove insufficient for our traffic demands.

In summary, each team can effectively utilize a message queue to facilitate communication with various microservices. Additionally, messages can be employed to achieve the functionality of a distributed architecture through remote procedure calls (RPC). Since message queue systems like Kafka allow for the runtime definition of topics, it becomes straightforward for each group to design their own messages and transmit them to different services.

5.7 Milestones

1. Perform manual updates for the service image.

2. Implement the deployment of the API gateway service.

3. Establish the GitLab-Runner for all team members within the CS5604-F2023 group.

4. Resolve container network issues on CI/CD VM by adjusting the network interface MTU setting.

5. Conduct comprehensive discussions on the database schema and initial design and review the existing schema.

6. Re-deploy and re-config Zookeeper to enable Kafka to establish a connection.

7. Deploy kafkacat and kafka-ui to facilitate troubleshooting for Kafka.

8. Leverage Kubernetes secrets to establish a static token for the Jupyter service in our cluster, ensuring it is accessible through this fixed token.

9. Test the connection between the GitLab group runner and the GitLab code repository using a sample code-base.

10. Create a group token in GitLab that all teams can use to build and deploy Docker images.
11. Create a sample CI/CD guide and share with all teams.
12. Establishing a new database schema, populating it with tables with data, and preparing for migration to the latest version of the database.
13. Connect API gateway to the new database and test connection.
14. Integrate API gateway with front-end application.
15. Implement priority APIs for LLM, object detection, and topic modeling groups.
16. Complete API documentation for all teams.
17. Add the path information of ETDs in the file system to the new database.
18. Create APIs for accessing the file system, such as storing XML, detected objects, page images, and retrieving ETDs and page images for Team 3.
19. Implemented all User personal status-related codes on the system.

5.8 Timeline

Table 5.5 illustrates the project timeline, featuring task descriptions and status information.

Table 5.5: Project Timeline

Sl.no	Description	Week	Status
1	Understanding knowledge gaps and learning through different resources to address these gaps	1	Done
2	Meeting with SME/TA to understand requirements. Learning about migration from old to new cluster	2	Done
3	Meeting with SME to get a better understanding of previous work done	3	Done
4	Working on migrating to a new cluster. Migrating secrets for image-pulling. Migrating deployments. Updating dependencies between each deployment after migrating from the old cluster	3	Done
5	Develop a sample API server	4	Done
6	Set up CI/CD pipeline and testing	5	Done
7	Deploy authentication service	8	Done
8	Update database schema	8	Done
9	Devise a proposal of table requirements for each team and subsequent discussion	9	Done
	Utilize Swagger doc as an open standard for documenting and testing APIs	16	Future work
11	Write a guide CI-CD yaml file	9	Done
12	Assist all teams with setting up their CI-CD pipelines	9	Future work
13	Scaling Kafka and add persistent storage to Zookeeper	9	Done
14	Update ETD and ETD_metadata tables with 500k data	11	Done
15	Add row for the 200 ETD collection in the collections table	12	Done
16	Connect API gateway to new database	13	Done
17	Fix front-end nginx server issue and deploy to the Kubernetes cluster	13	Done
18	Complete API list documentation	13	Done
19	Create path Add path information to new database schema	13	Done
20	Create new path for 200+ segmented ETDs; make an update to database schema and database tables	13	Done
21	Create API for file system access	14	Done
22	Generate a TSV with (ETD ID, Chapter ID) for team1	14	Future work
23	Provide a list of chapter IDs to teams 2 and 4 for indexing and processing	14	Done
24	Include a new persona in the database called “registered” and limit access to the system	16	Done
25	Write a test routine for all implemented APIs	15	Future work

Chapter 6

User Manual and Developer Manual

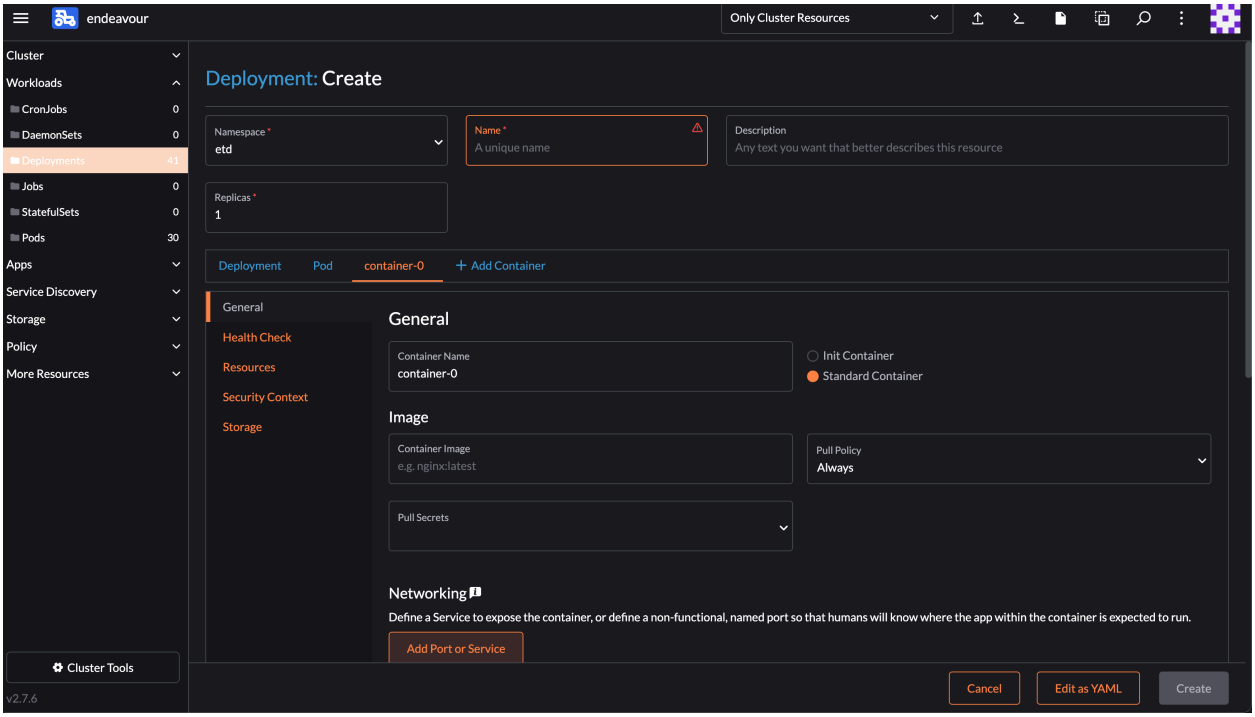
6.1 Rancher

All we need to know about how to manipulate Rancher is deployment, ingress, service, and node. We will elaborate on them with Rancher as an example.

6.1.1 Deployment

Deployment is a fundamental method for deploying pods within Kubernetes. In Figure 6.1, you can see a straightforward approach to creating a new pod on Kubernetes. By specifying the image name and providing certain configurations, Rancher assists us in generating the YAML file required for the deployment.

Figure 6.1: Rancher's deployment

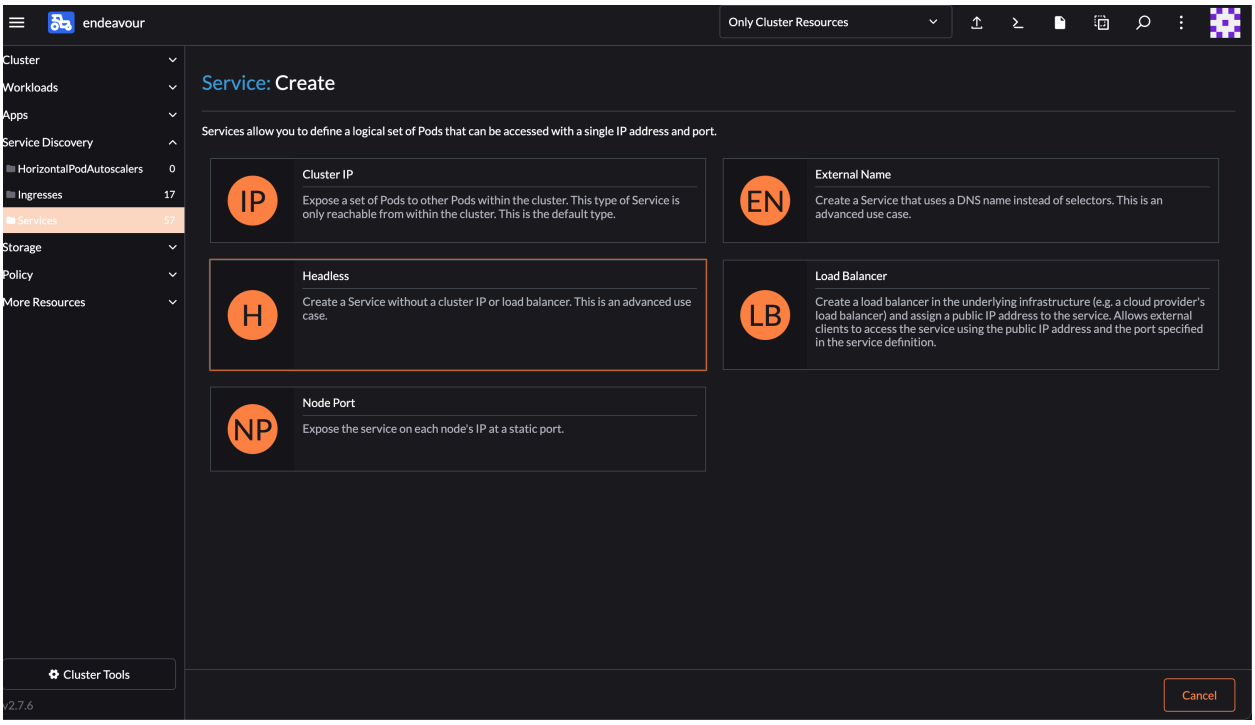


6.1.2 Service

In Kubernetes, a "Service" is a resource kind that defines how a group of pods can be accessed. Kubernetes provides several service types in Figure 6.2, including ClusterIP, NodePort, LoadBalancer, ExternalName, and Headless. ClusterIP is the default service type, exposing services with a cluster-internal IP address. NodePort allows external access by assigning a static port to each node's IP address. LoadBalancer uses a cloud provider's load balancer to distribute external traffic. ExternalName maps services to DNS names for connecting to external resources. Headless services return DNS records for all associated pods, suitable for stateful applications. These services enable developers to control how services are exposed both internally and externally, catering to various networking and access

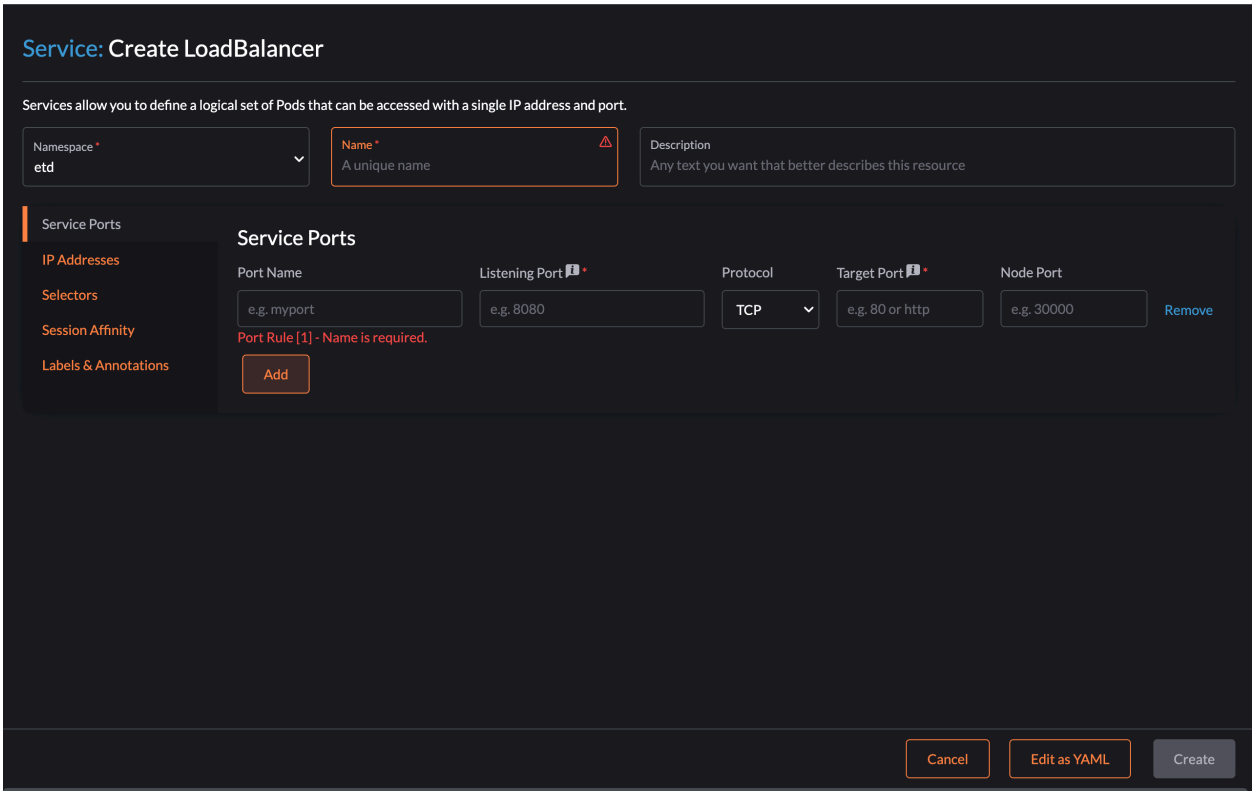
requirements in a Kubernetes cluster.

Figure 6.2: Rancher's service



In Kubernetes, the Service LoadBalancer, shown in Figure 6.3, automatically provisions a cloud provider's load balancer, allowing you to expose your application to the internet or external network. It distributes incoming traffic across pods for scalability and high availability. You only need to specify selectors or labels in your deployment for the service to identify targets. However, the details of LoadBalancer services can vary based on your cloud provider's Kubernetes integration.

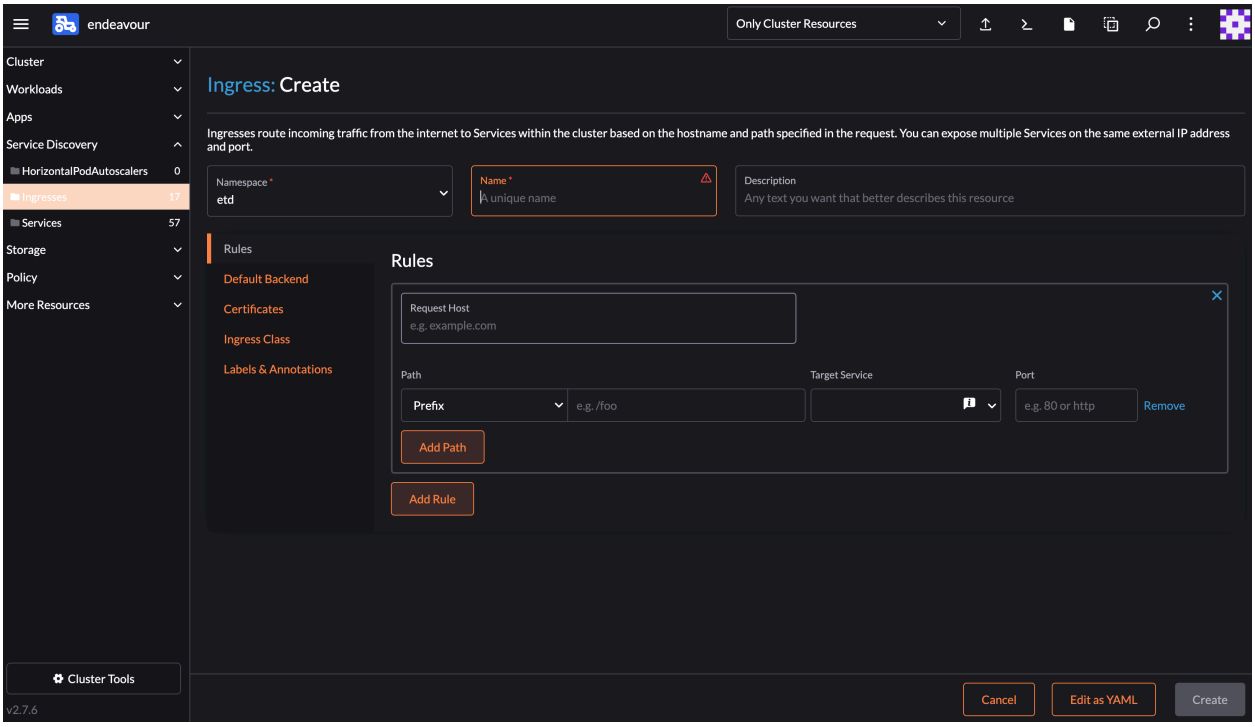
Figure 6.3: Rancher's loadbalancer



6.1.3 Ingress

Ingress provides a means to allow external users to access internal services. In Figure 6.4, Rancher can be directly utilized to generate Ingress YAML configurations. You simply need to specify the desired service and port mapping. You can connect to the internal services with the provided URL and port.

Figure 6.4: Rancher's ingress

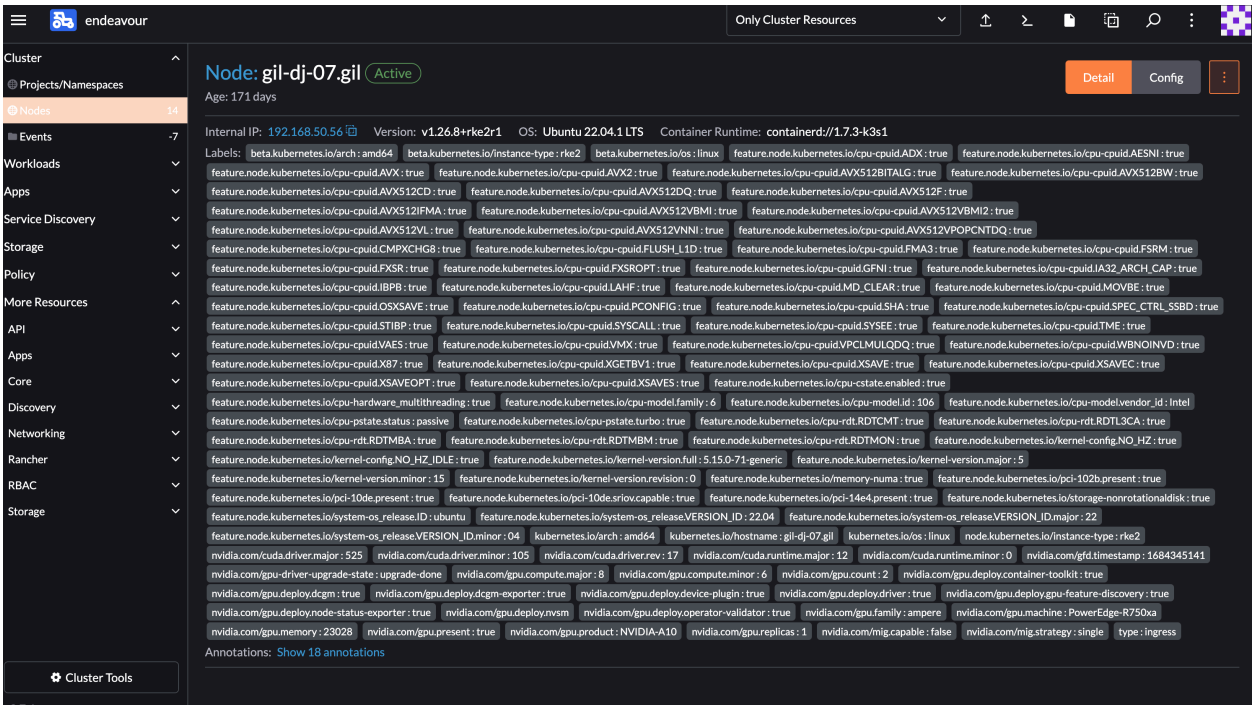


6.1.4 Node

In Kubernetes, a "node" represents an individual worker machine within a cluster, responsible for executing containers and providing computational resources for applications. Nodes can be adorned with key-value pairs known as "labels," facilitating the categorization of nodes based on diverse attributes. These labels serve the purpose of organization and node selection for specific tasks, such as deploying particular workloads or applying specific configurations. For instance, nodes can be labelled with descriptors like "environment=production," "memory=bigmemory," or "nvidia.com/gpu.present=true." Labels find utility in various Kubernetes configurations, including node selectors in pod specifications, enabling the precise allocation of pods to nodes with matching labels. In our scenario, labels assist in locating

nodes tagged with “owner:fox,” a requirement for services necessitating GPU resources, as demonstrated in Figure 6.5.

Figure 6.5: Rancher’s nodes



6.2 kubectl

Before using Kubectl, obtaining the configuration file from Rancher is essential. Figure 6.6 illustrates that the configuration file can be easily downloaded with a single click.

Figure 6.6: Rancher’s config

Once downloaded, you can configure Kubectl to use this file, typically achieved by setting

the KUBECONFIG environment variable to the file's path or using the –kubeconfig flag when running Kubectl commands. With Kubectl configured, you gain full control over your Kubernetes cluster. For instance, you can use commands like 'Kubectl –kubeconfig /Download/endeavour.yaml get pods –namespace=etd' to interact with your cluster, listing pods in the specified namespace as an example.

6.3 Migration Steps

The migration of all deployments from the Discovery cluster to the Endeavour cluster has been successfully completed. The initial step involved retrieving the YAML file from the Discovery cluster.

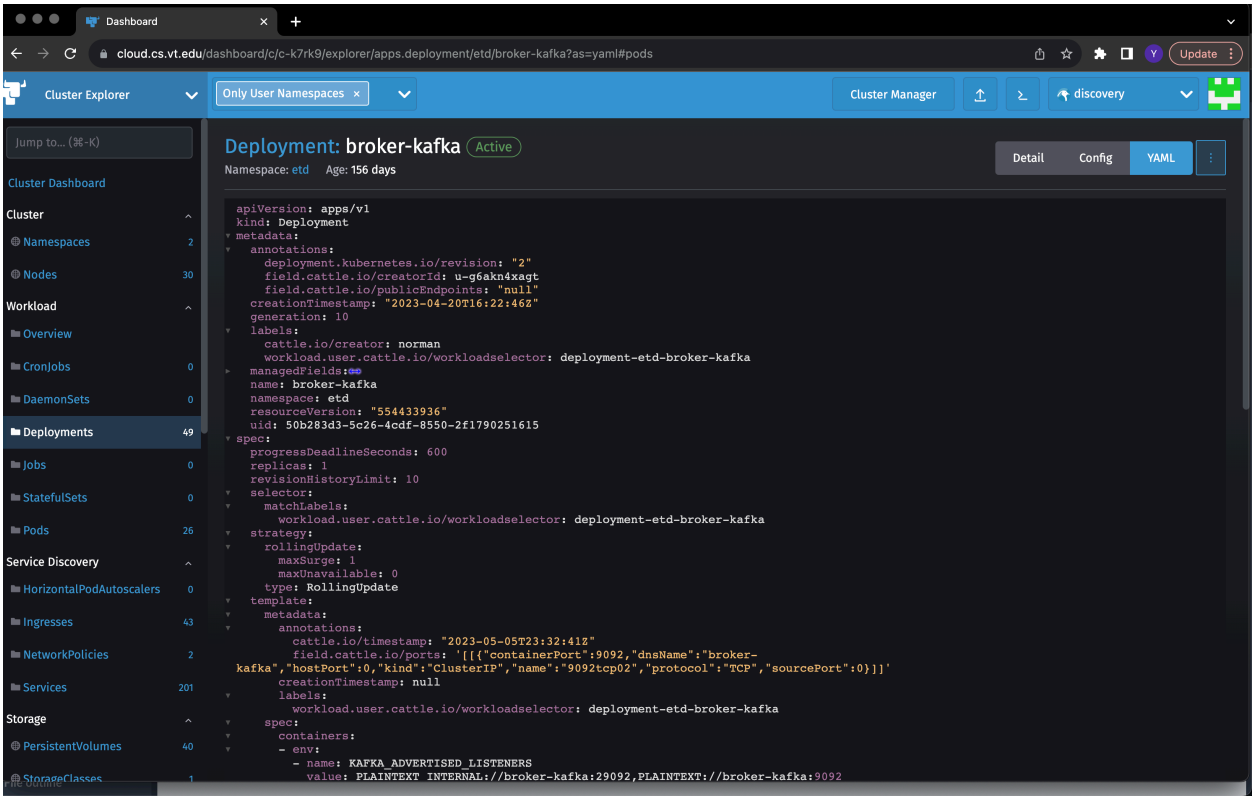


Figure 6.7: Kafka deployment yaml

In Figure 6.7, you can see the YAML file on the Discovery cluster on the Rancher website containing Kubernetes's configuration details. It's important to note that we should omit the version information from the YAML file and, if needed, update details such as ports or tokens.

The next step involves uploading the updated YAML file to the Endeavour cluster. We have two options for deployment: we can either use Rancher for uploading and deploying or employ 'kubectl' with a command like 'kubectl apply -f kafka.yaml' to deploy Kafka. After deployment, we should verify the status and inspect the logs to troubleshoot any potential issues.

The last crucial step is the migration of secrets. Failing to migrate secrets can lead to image-pulling failures or container access issues that depend on secret information. Therefore, we must download all the secrets from the Discovery cluster and then upload them to the Endeavour cluster.

6.4 GitLab CI/CD

A DevOps pipeline must be established in order to set up Continuous Integration and Continuous Deployment. This pipeline takes a piece of code, does some operations on it, such as building or testing, and then deploys it elsewhere. This is the main function of a pipeline. Additionally, pipelines may get more sophisticated. We will also use GitLab's own DevOps, also known as the GitLab Runner, since it serves as our primary repository for storing all of the code. Below are some instructions for setting up the GitLab Runner and configuring it to use the Docker executor.

As was previously indicated, Docker must be installed on the virtual machine from CS

containers.cs.vt.edu. Since the command line would be the only interface, installing the Docker Engine is sufficient. The following procedures must be followed in order to install Docker on Ubuntu; hence, they are OS-dependent.

1. Go to https://docs.docker.com/engine/install/ and choose your distribution. Follow the steps provided to install Docker.

2. Verify the MTU on the virtual machine. If the VM's MTU is less than 1500, adjust the MTU for Docker accordingly.

   ```
   echo '{ "mtu": 1450 }' >> /etc/docker/daemon.json
   ```

3. Once installed, initiate the Docker service by executing the command

   `sudo service docker start`.

4. Verify the successful installation and startup of Docker by running a test image, for example,

   `sudo docker run hello-world`.

Now that Docker Engine is installed, you can install GitLab Runner. It is recommended to utilize the Linux Distribution to utilize the local Docker Engine without the necessity of installing specific programming language libraries.

1. Go to https://docs.gitlab.com/runner/install/linux-repository.html and install the runner specific to your distribution.

2. Confirm the installation status by executing

   `sudo gitlab-runner status`.

3. Next, you need to enable shared group runner for the project; navigate to Settings (Left Menu) -> CI/CD -> Runners> Enable Shared Runners for this group; please refer to Figure 6.8.

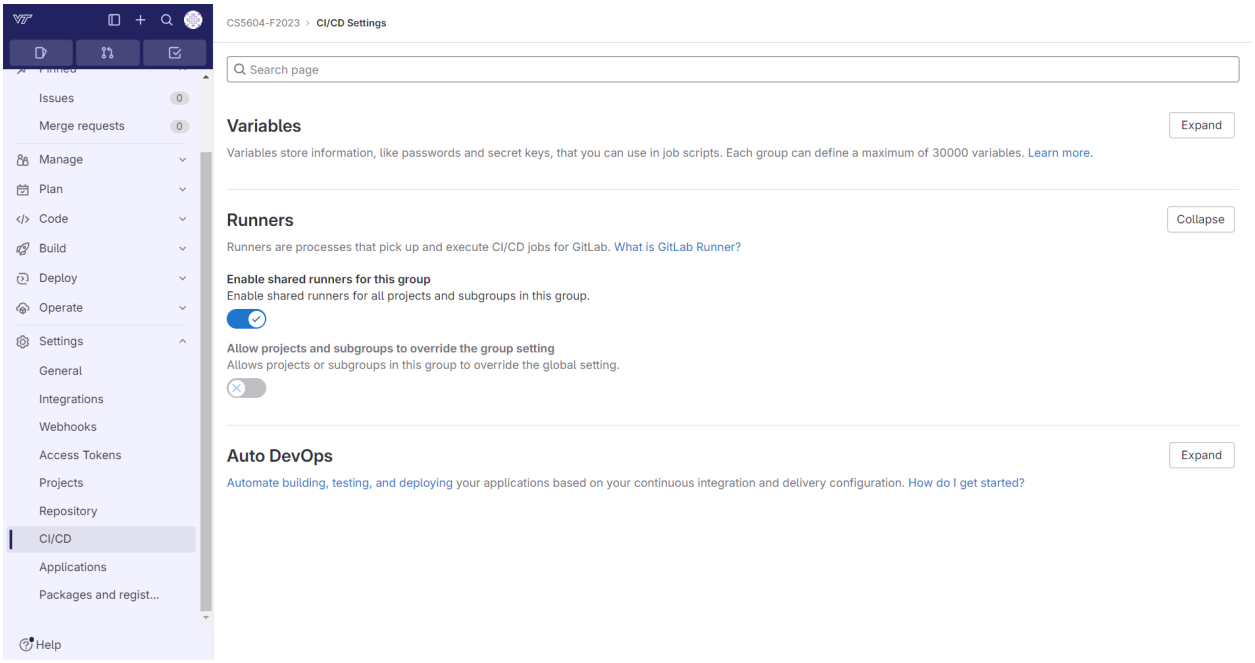


Figure 6.8: Enabling Shared Group Runner in GitLab

4. Next, you need to Register the Runner. In your Repository (we are utilizing a Repository in `https://code.vt.edu`), navigate to Build (Left Menu) -> Runner -> New Group Runner.

5. Fill in all the required fields to set up the group runner, and click on “create runner”.

6. Note the URL and Registration token.

7. Download the config file from the Endeavour cluster and move it to the config folder in the VM. This file will be later used to configure our group runner in the config.toml file.

8. Now run the following command.

```
sudo gitlab-runner register
```

9. Enter all the details as asked and select Docker as the executor and docker:dind as the default image.

10. In the config.toml file add concurrency as 6, so six teams can run their jobs with the same group runner. Also, set privileged to true, and under volumes, the Docker socket, as in the host Docker socket, should be mounted to the Containers.

```
concurrency = 6

privileged = true

volumes = ["/var/run/docker.sock:/var/run/docker.sock",

"/cache", "/home/aaron/2023/config:/.kube/config"]
```

11. Test the connection between the group runner and the GitLab code-base, by building a pipeline as shown in Section 5.5.

12. To support teams in establishing CI-CD pipelines for individual projects, we have developed a sample guide [10]. This guide demonstrates the process of setting up test, build, and deploy stages for a project repository.

Additionally, we encountered an MTU issue while setting up a GitLab runner in the VM. To resolve this, we adjusted the network interface's MTU settings. The problem was that packets exceeding 1450 bytes in length could not be delivered. This occurred because the host used an MTU of 1450, while Docker used an MTU of 1500. Consequently, Docker would send out 1500-byte packets, which the host would then drop. To address this issue, we configured Docker's MTU to match the host's MTU of 1450.

6.5 Database Migration

```
-- Name: Classifications; Type: Table; Schema: Public; Owner: docker

CREATE TABLE public.Classifications (
    id integer NOT NULL,
    name character varying(225) NOT NULL,
    Description character varying NOT NULL);


ALTER TABLE public.Classifications OWNER TO docker;


-- Name: Classification_entries; Type: Table; Schema: Public; Owner: docker

CREATE TABLE public.Classification_entries (
    id integer NOT NULL,
    classifications_id integer NOT NULL,
    name character varying(225) NOT NULL,
    description text NOT NULL,
    depth integer NOT NULL);


ALTER TABLE public.Classification_entries OWNER TO docker;

-- Name: classifiers; Type: Table; Schema: Public; Owner: docker

CREATE TABLE public.classifiers (
    id integer NOT NULL,
    classifications_id integer NOT NULL,
    name character varying(225) NOT NULL,
    description text NOT NULL,
    path character varying(500) NOT NULL);

ALTER TABLE public.classifiers OWNER TO docker;


-- Name: collection_topics; Type: Table; Schema: Public; Owner: docker

CREATE TABLE public.collection_topics(
        id integer NOT NULL,
        collections_id integer NOT NULL,
        topic_models_id integer NOT NULL,
        term_list text[] NOT NULL,
        Probability_list integer[] NOT NULL
);
```

Figure 6.9: Queries to create table

```
ALTER TABLE public.object_classes
ADD CONSTRAINT Object_classes_Classifications_fk
FOREIGN KEY (Classifications_ID) REFERENCES public.public.Classifications (ID);


-- Name: Object_classes Objects; Type: FK CONSTRAINT; Schema: public; Owner: docker

ALTER TABLE public.Object_classes
ADD CONSTRAINT Object_classes_Objects_fk
FOREIGN KEY (Objects_ID) REFERENCES public.Objects (ID);



-- Name: Object_summaries_Objects_fk; Type: FK CONSTRAINT; Schema: public; Owner: docker

ALTER TABLE public.Object_summaries
ADD CONSTRAINT Object_summaries_Objects_fk
FOREIGN KEY (Objects_ID) REFERENCES public.Objects (ID)



-- Name: Object_topics_ETDs_fk; Type: FK CONSTRAINT; Schema: public; Owner: docker

ALTER TABLE public.Object_topics
ADD CONSTRAINT Object_topics_ETDs_fk
FOREIGN KEY (ETDs_ID) REFERENCES public.ETDs (ID);


-- Name: Object_object_neighbors_fk; Type: FK CONSTRAINT; Schema: public; Owner: docker

ALTER TABLE public.Object_object_neighbors
ADD CONSTRAINT Object_object_neighbors_objects_fk
FOREIGN KEY (Objects_ID) REFERENCES public.Objects (ID)



-- Name: User_classes_Users_fk; Type: FK CONSTRAINT; Schema: public; Owner: docker

ALTER TABLE public.User_classes
ADD CONSTRAINT "User_classes_Users_fk"
FOREIGN KEY (Users_ID) REFERENCES public.users (ID);


```

Figure 6.10: Queries to create foreign keys

```python
import psycopg2
from psycopg2.extras import execute_values
import csv

db_params = {
    'host': '[redacted]',
    'port': [redacted],
    'dbname': '[redacted]',
    'user': '[redacted]',
    'password': [redacted]
}

csv_file_path = 'C:/Downloads/etds_500k.csv'
table_name = 'public.etd_metadata'

column_mapping = {
    'idsl': 'id',
    'id': 'etds_id',
    'title':'title',
    'author':'author',
    'university':'institution',
    'department':'department',
    'year':'year',
    'degree':'degree'
}

conn = psycopg2.connect(**db_params)
cur = conn.cursor()

data_to_insert = []

with open(csv_file_path, 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file)

    for row in csv_reader:
        transformed_row = {column_mapping[csv_column]: row[csv_column] for csv_column in column_mapping}
        data_to_insert.append(transformed_row)

insert_sql = f"""
    INSERT INTO {table_name} (id,etds_id,title,author,institution,department,year,degree)
    VALUES %s
"""

values = [tuple(row.values()) for row in data_to_insert]
execute_values(cur, insert_sql, values, page_size=1000)


conn.commit()
cur.close()
conn.close()
```

Figure 6.11: Script to import 500k data

Figure 6.9 displays a few of the queries necessary to create tables. On the other hand, Figure 6.10 presents queries that help establish relationships between these tables.

We developed a Python script to populate the database. Below is a description of the script and its processing.

The script shown in Figure 6.11 initiates a connection to the PostgreSQL database by specifying parameters such as host, port, dbname, user, and password. It then specifies the CSV file path (csv_file_path) and the target table name (table_name). Additionally, a column_mapping is created to align CSV columns with the corresponding ones in the database table.

Upon reading the CSV file, the script transforms each row based on the predefined column_mapping. It collects the changed data into a list called 'data_to_insert.' The script optimizes the data insertion process using a function called psycopg2.extras.execute_values, making the insertion of a large amount of data more efficient by handling it in batches with a specified size limit.

6.6 Kafka

We employ Python to interact with Kafka, where the designated topic is "testing_topic". Post message production, the topic and its messages become observable through a Kafka monitoring tool. For this purpose, we utilize a Kafka UI, as referenced in [39], to monitor the Kafka system's status.

The example code in Listing 6.1 demonstrates the process of interacting with Kafka using Python. It initializes a class IR_Streaming to manage Kafka consumer and producer functionalities. The script sets up the Kafka environment, including the configuration of the broker server and group ID. It also includes functions for producing and consuming messages to and from a Kafka topic, in this case, testing_topic. The produce method sends messages to the topic, while the consume_registration and consumer_run methods set up

the consumer to receive and process these messages. Error handling and message callbacks are incorporated to ensure reliable message delivery and processing. The script showcases handling Kafka messages using Python effectively, demonstrating key concepts in Kafka's message production and consumption.

Listing 6.1: Python example for Kafka testing

```
\begin{lstlisting}
from argparse import ArgumentParser, FileType
from configparser import ConfigParser
from confluent_kafka import Producer, Consumer, OFFSET_BEGINNING, Producer,
    KafkaException
import os
from threading import Thread
import asyncio
import pickle
import time

kafka_config_file = os.environ.get(captionpos'captionposKAFKA_CONFIGcaptionpos',
    captionpos'captionpos./captionposgetting_startedcaptionpos.captionposini
    captionpos')

class IR_Streaming():
     def __init__(self, config, loop=None):
          self._loop = loop or asyncio.get_event_loop()
          self.config = config
          self.topics = set()
          self.producer = None
          self.comsumer = None
          self._cancelled = False
          self._poll_thread = None

```

```
23      def enable_threed_loop(self):
24          if self._poll_thread is None:
25              self._poll_thread = Thread(target=self._poll_loop)
26              self._poll_thread.start()
27
28      def _poll_loop(self):
29          while not self._cancelled:
30              self.producer.poll(0.1)
31              self.producer.flush()
32
33      @staticmethod
34      def default_ack(err, msg):
35          if err:
36              print(captionpos'captionposERRORcaptionpos:captionpos captionposMessage
                    captionpos captionposfailedcaptionpos captionposdeliverycaptionpos:
                    captionpos captionpos{}captionpos'.format(err))
37          else:
38              print(captionpos"captionposProducedcaptionpos captionposeventcaptionpos
                    captionpostocaptionpos captionpostopiccaptionpos captionpos{
                    captionpostopiccaptionpos}:captionpos captionposkeycaptionpos
                    captionpos=captionpos captionpos{captionposkeycaptionpos:12}
                    captionpos captionposvaluecaptionpos captionpos=captionpos captionpos{
                    captionposvaluecaptionpos:12}captionpos".format(
39                   topic=msg.topic(), key=msg.key().decode(captionpos'captionposutf
                        captionpos-8captionpos'), value=msg.value().decode(captionpos'
                        captionposutfcaptionpos-8captionpos')))
40
41      def produce(self, topic, data, key=None, callback=None):
42          if self.producer is None:
43              self.producer = Producer(self.config)
44
```

```
45          if callback is None:
46              callback = self.default_ack
47
48          self.enable_threed_loop()
49          result = self._loop.create_future()
50
51          def ack(err, msg):
52              if callback == self.default_ack:
53                  return self._loop.call_soon_threadsafe(
54                      callback, err, msg)
55              if err:
56                  self._loop.call_soon_threadsafe(
57                      result.set_exception, KafkaException(err))
58              else:
59                  self._loop.call_soon_threadsafe(
60                      result.set_result, msg)
61              if callback:
62                  self._loop.call_soon_threadsafe(
63                      callback, err, msg)
64
65          self.producer.produce(topic, data, key, on_delivery=ack)
66          return result
67
68      def register_tpoic(self, topic):
69         self.topics.add(topic)
70
71      def comsume_registrition(self, handler, recover=False):
72          if self.comsumer is None:
73              self.consumer = Consumer(config)
74
75          def reset_offset(consumer, partitions):
```

```
76              if recover:
77                  # recover from error or crushing, we can use offset_beginning
                        to get undone message
78                  for p in partitions:
79                      p.offset = OFFSET_BEGINNING
80                  consumer.assign(partitions)
81
82          self.consumer.subscribe(list(self.topics), on_assign=reset_offset)
83          self.handler = handler
84          print(fcaptionpos"captionposregisteredcaptionpos captionpos{captionposlist
                captionpos(captionposselfcaptionpos.captionpostopicscaptionpos)captionpos}
                captionpos")
85
86      def __comsumer_run__(self):
87          count = 0
88          try:
89              while True:
90                  msg = self.consumer.poll(1.0)
91                  if msg is None:
92                      if count == 0:
93                          print(captionpos"captionposWaitingcaptionpos...captionpos")
94                      count += 1
95                      count = count if count != 1000 else 0
96                      continue
97                  if msg.error():
98                      print(fcaptionpos"captionposERRORcaptionpos:captionpos
                            captionpos{captionposmsgcaptionpos.captionposerror
                            captionpos()captionpos}captionpos")
99                  elif self.handler:
100                     self.handler(msg)
101                     self.consumer.commit(msg)
```

```
102             finally:
103                 self.consumer.close()
104 
105         def comsumer_run(self):
106             Thread(target=self.__comsumer_run__).start()
107 
108 if __name__ == captionpos'captionpos__main__captionpos':
109         # Parse the command line.
110         parser = ArgumentParser()
111         parser.add_argument(captionpos'captionposconfig_filecaptionpos', type=FileType
                (captionpos'captionposrcaptionpos'))
112         parser.add_argument(captionpos'captionpos--captionposresetcaptionpos', action=
                captionpos'captionposstore_truecaptionpos')
113         args = parser.parse_args()
114 
115         # Parse the configuration
116         config_parser = ConfigParser()
117         try:
118             config_parser.read_file(kafka_config_file or args.config_file)
119             config = dict(config_parser[captionpos'captionposdefaultcaptionpos'])
120         except Exception as e:
121             print(fcaptionpos"captionposERRORcaptionpos captionpos{captionposecaptionpos}
                    captionpos")
122             print(captionpos"captionposusecaptionpos captionposdefaultcaptionpos
                    captionpossettingcaptionpos")
123             config = {
124 captionpos                          captionpos'captionposbootstrapcaptionpos.
                            captionposserverscaptionpos': captionpos'captionposbroker
                            captionpos-captionposkafkacaptionpos:9092captionpos',
125 captionpos                          captionpos'captionposgroupcaptionpos.captionposid
                            captionpos': captionpos'captionposteam5captionpos',
```

```
captionpos                          captionpos'captionposautocaptionpos.captionposoffset
                            captionpos.captionposresetcaptionpos': captionpos'
                            captionposearliestcaptionpos'
                    }

    streaming = IR_Streaming(config)
    topic = captionpos"captionpostesting_topiccaptionpos"
    key = captionpos"captionposteam5captionpos-1captionpos"
    crushing_last_time = False

    def producer_callback(err, msg):
        if err:
            print(captionpos'captionposERRORcaptionpos:captionpos captionposMessage
                captionpos captionposfailedcaptionpos captionposdeliverycaptionpos:
                captionpos captionpos{}captionpos'.format(err))
        else:
            print(captionpos"captionposProducedcaptionpos captionposeventcaptionpos
                captionpostocaptionpos captionpostopiccaptionpos captionpos{
                captionpostopiccaptionpos}:captionpos captionposkeycaptionpos
                captionpos=captionpos captionpos{captionposkeycaptionpos:12}
                captionpos captionposvaluecaptionpos captionpos=captionpos captionpos{
                captionposvaluecaptionpos}captionpos".format(
                 topic=msg.topic(), key=msg.key().decode(captionpos'captionposutf
                    captionpos-8captionpos'), value=pickle.loads(msg.value())))

    def message_handler(msg):
        print(captionpos"captionposGotcaptionpos captionposmessagecaptionpos
            captionposwithcaptionpos captionpostopiccaptionpos captionpos{
            captionpostopiccaptionpos}:captionpos captionposkeycaptionpos captionpos=
            captionpos captionpos{captionposkeycaptionpos:12}captionpos
            captionposvaluecaptionpos captionpos=captionpos captionpos{captionposvalue
```

```
            captionpos}captionpos".format(
143                 topic=msg.topic(), key=msg.key().decode(captionpos'captionposutf
                        captionpos-8captionpos'), value=pickle.loads(msg.value())))
144         if msg.topic() == topic and msg.key().decode(captionpos'captionposutf
            captionpos-8captionpos') == key:
145             data = pickle.loads(msg.value())
146             if type(data) is dict:
147                 print(data)
148                 return
149         print(fcaptionpos"captionposERRORcaptionpos captionpos{captionposmsg
            captionpos}captionpos")
150
151     streaming.register_tpoic(topic)
152     streaming.comsume_registrition(message_handler, recover=True if
        crushing_last_time else False)
153     streaming.comsumer_run()
154
155     for i in range(3):
156         streaming.produce(topic, pickle.dumps(obj = {
157 captionpos              captionpos"captionposhellocaptionpos": i,
158 captionpos              captionpos"captionposokcaptionpos": i+1
159         }, protocol=4), key=key, callback=producer_callback)
160
161     time.sleep(10)
```

We utilize Python's pickle module to encode objects into binary data, which are then transmitted within the message. Upon receiving the message, the consumer decodes this data back into its original form. In our implementation, the decoded data is specifically converted into a dictionary.

6.7 API Gateway

The curated technology stack listed below empowers the delivery of the robust, scalable, and secure API gateway, ensuring seamless integration and data exchange across the entire project's landscape.

- Backend: Flask (Python framework for creating RESTful APIs)
- ORM: SQLAlchemy [40] (for database interactions)
- Authentication:
  - JWT (JSON Web Tokens for secure user authentication)
  - API keys

API gateway structure

Figure 6.12 illustrates the implementation of the API gateway and the organization of APIs within the project. It visually represents the overall architecture and the interactions between the various components.

JWT Utils

For user authentication and authorization in our API, JWT was implemented. JSON Web Tokens are an open standard that defines how to transmit information securely between parties as a JSON object. The information can be verified from the Endeavour cluster and trusted because it is digitally signed.

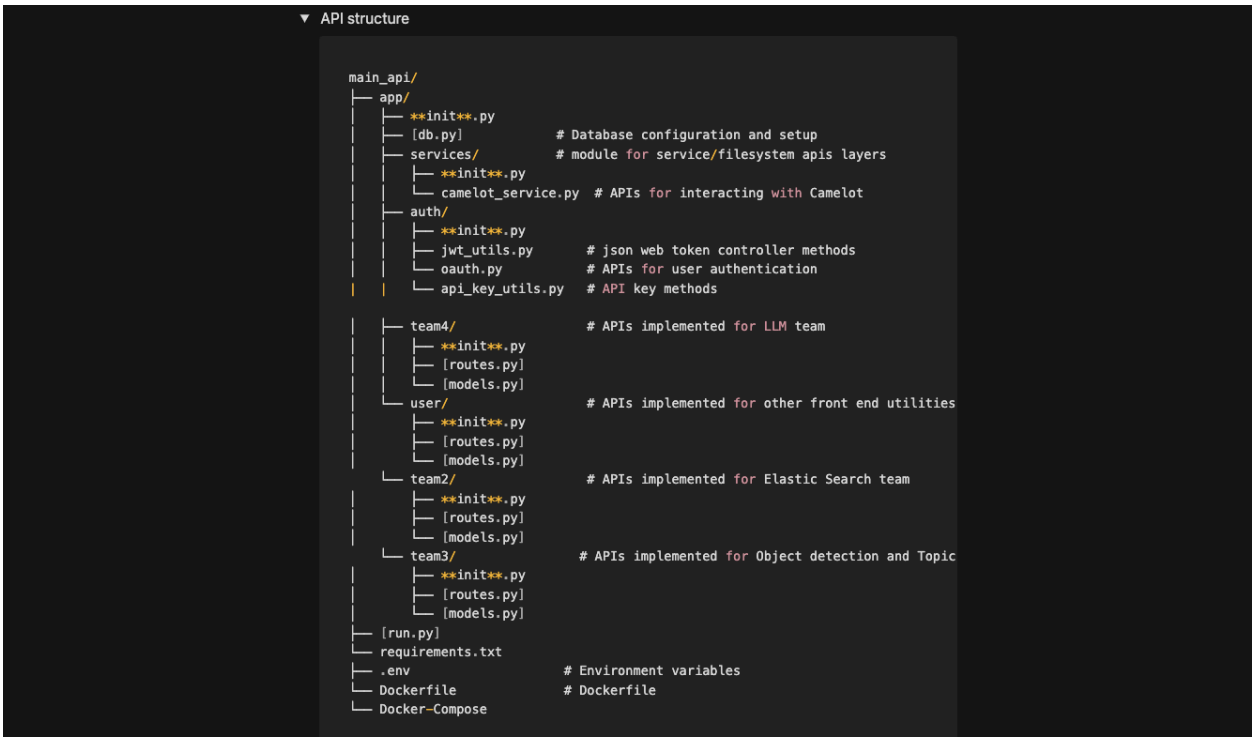


Figure 6.12: API gateway application structure

- JWT Utils Controller: The JWT Utils Controller generated and validated JWT tokens; see Figure 6.13 for the application code.
- JWT Secret Key: The security of JWT relies heavily on the secret key used to sign the tokens. In this project's implementation, the JWT secret key is stored securely in a Kubernetes (K8s) Endeavour cluster.

RESTful Principles:

In the development of the APIs, we adhered to some key RESTful principles to ensure the system was efficient, scalable, and easy to maintain:

- Client-Server Architecture: The client and the server operate independently, allowing each to evolve separately.

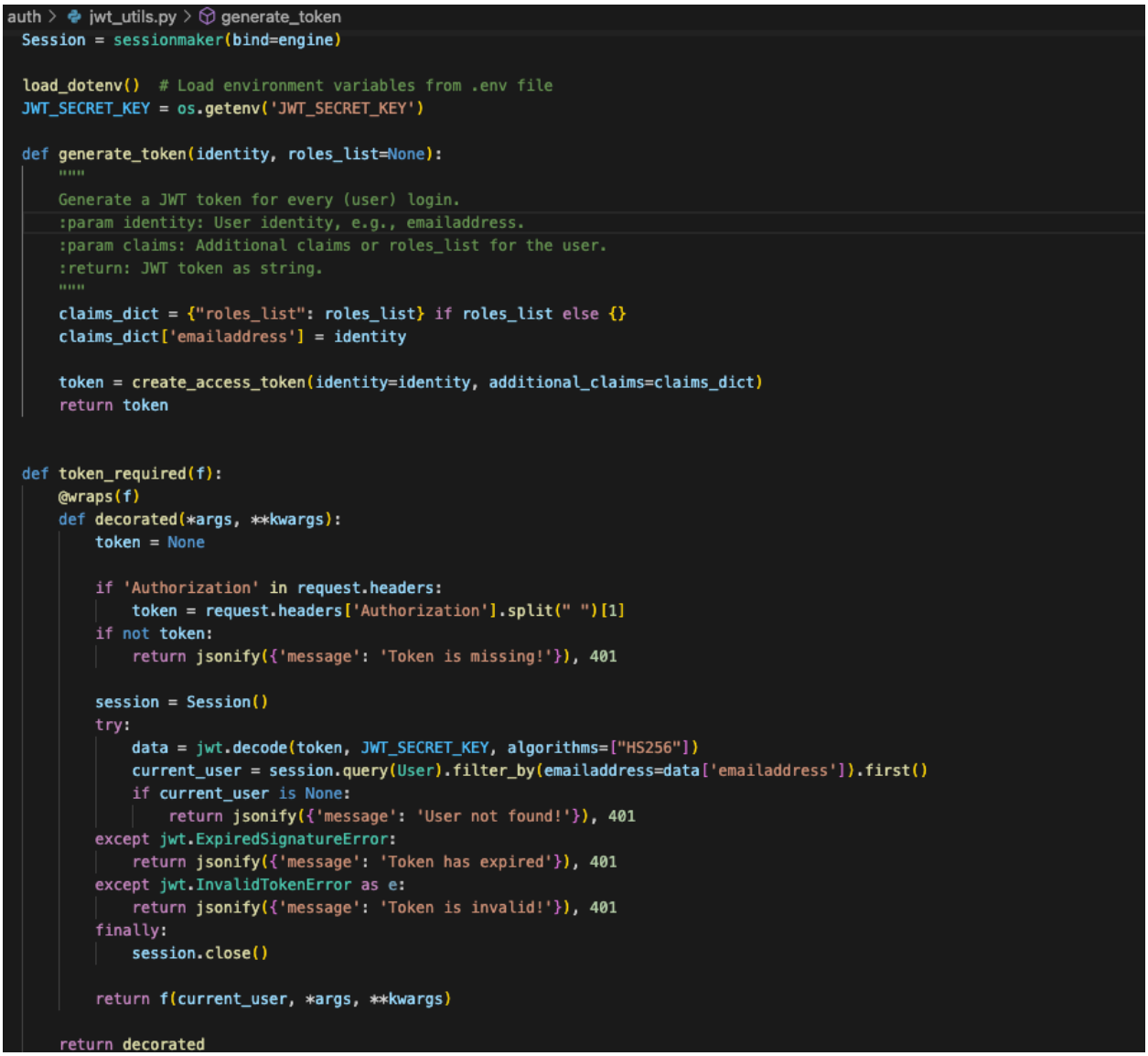


Figure 6.13: JSON web tokens Utils Controller

- Uniform Interface: A consistent and standardized approach to how the client interacts with the application. This includes using standard HTTP methods (GET, POST, PUT) and URLs to manipulate resources.

Version Control with GitLab

- Step 1: Clone the Repository
  - SSH or HTTPS: Decide whether to use SSH or HTTPS for cloning. SSH is more secure but requires setting up SSH keys. HTTPS is easier but less secure.
  - Clone Command: Use git clone <repository-url> to clone the repository from GitLab to your local machine. Replace <repository-url> with the actual URL of your GitLab repository.
- Step 2: Run the Application
  - Navigate to Directory: Change into the directory of the cloned repository using cd <repository-name>.
  - Install Dependencies and Run: Depending on the application, you might need to install dependencies first. You might run 'pip install -r requirements.txt' for a Python application. Then, start the application as defined in the project's documentation (e.g., Flask run and Gunicorn app for production server).

Containerization with Docker

- Build the Docker Image:
  - Navigate to the directory containing the Dockerfile and run: 'docker build -t containername:latest'.

- Run the Docker Container:
  - After building the image, you can run the application in a Docker container: docker-compose up

Deployment on Kubernetes (Endeavour Cluster)

- Step 1: Create Kubernetes Configuration Files
  - Define Deployment Configuration: Write a Kubernetes deployment YAML file. This file should specify the Docker image to use (pointing to the image in your container registry), the number of replicas, ports, environment variables, and other deployment-specific settings.
  - Define Service Configuration: Create a service YAML file to expose your application. The service acts as a load balancer that distributes network traffic across all instances of your application.
- Step 2: Apply the Configuration and Monitor
  - Apply Configuration: Use 'kubectl apply -f <filename>.yaml' to apply your deployment and service configurations to the Kubernetes cluster. This command will create the necessary pods, services, and other resources defined in your YAML files.
  - Monitor Deployment: Monitor the deployment's status using 'kubectl get pods' to ensure that the pods are running correctly. Check logs and events for any issues or errors that might need addressing.

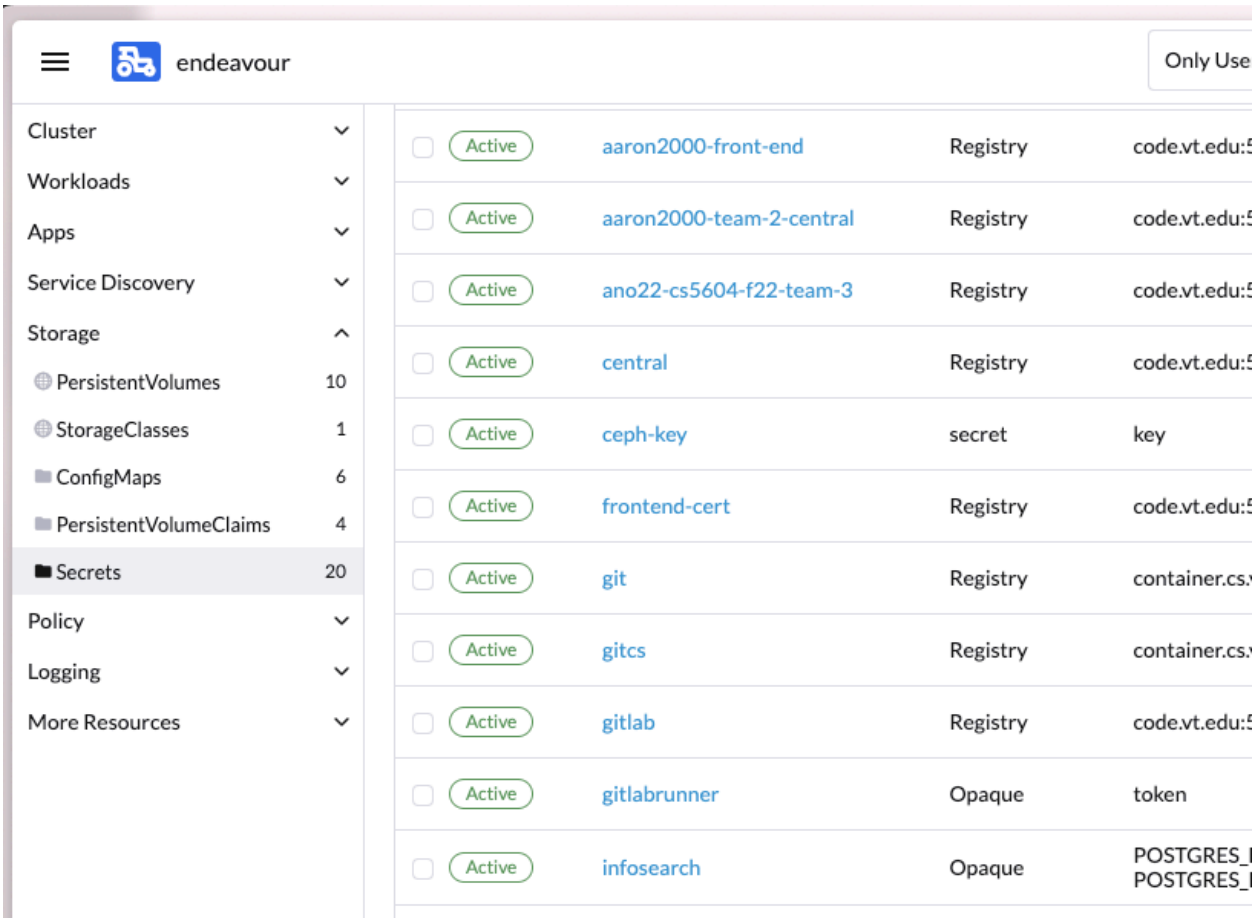


Figure 6.14: List of sensitive credentials stored on Endeavour

Security Best Practices

Sensitive Data Management: We used Kubernetes Secrets to store sensitive information such as database credentials, JWT secret keys, API keys, and Docker container keys, as shown in Figure 6.14. This approach ensures that such critical data is not hard-coded into the application or exposed in our Git repository.

API List

We facilitated the needs of other teams by providing APIs tailored to their specific requirements. See Table 6.1.

Team	Container role	Request Method	API	API Description	Entity (database/filesystem)
3	Object Detection	GET	/services/etds/<int:etd_id>/pdf	Get the ETD document stored in the file system and the ETD ID corresponding to it. (ETD table)	Database
3	Object Detection	POST	/services/save-page-image	Store page images of each ETD file's pages. (ETD_pages table)	Database & FS
3	Object Detection	GET	/services/retrieve-page-image/<int:page_id>	Retrieve page image with pageID (ETD_pages table)	Database & FS
3	Object Detection	POST	/services/store-object	Store the objects detected. The text-based objects would be stored in the database system, and for image-based objects, their path would be stored in the DB, and the actual images would be stored in the file system (Objects table). API returns a unique Object ID	Database & FS
3	Object Detection	POST	/services/store-xml	Store the path of the generated XML/JSON file in the database and the XML/JSON file in the file system.	Database
3	Object Detection	GET	/services/etds/<int:etd_id>	Retrieve ETD, checks for ETD ID in file system directory, cross-checks with database and returns etd information and path	Database & FS
3	Object Detection	GET	/services/etds/<int:etd_id>/pdf	Retrieve actual PDF ETD, checks for etd ID in the file system directory and cross-checks with database	Database & FS
3	Object Detection	GET	/services/etds/	Retrieve all ETDs info	Database & FS
3	Object Detection	POST	/team3/api/objects	POST to objects table	Database
3	Object Detection	POST	/team3/etd-metadata	POST to ETD-metadata table	Database
3	Topic Modelling	POST	/team3/api/topic-models	Add to Topic_models table, one row for a topic modeler	Database
3	Topic Modelling	POST	/team3/api/collection-topics	Add to Collection_topics table to store the set of topics from that modeler	Database
3	Topic Modelling	GET	/team3/api/etd-topics/<int:etd_id>	Retrieve Topic Vector for an ETD	Database
3	Topic Modelling	POST	/team3/api/etd-topics/	Store Topic Modeling Results for ETDs	Database

3	Topic Modelling	POST	/team3/api/object-topics	Add to Object_topics table to store the topic modeling results using the topic	Database
4	Classifications	POST	common/classifications	Add to Classifications table to add one row, for ProQuest	Database
4	Classifications	POST	/classification-entries	Add to Classification_entries table to one row for each of the 2nd level categories in the ProQuest set	Database
4	Object Detection	POST	common/objects	Add to Objects table to save the chapter texts for the 200 ETDs	Database
4	Classifications	POST	/object-metadata	Add to Object_metadata table to save some metadata about those chapters	Database
4	Summarizations	POST	/summarizers	Add to Summarizers table to add one row for that summarizer	Database
4	Summarizations	POST	common/object_summaries	Add to Object_summaries table to save the summarization results	Database
4	Summarizations	GET	common/object_summaries	Retrieve a list of all object summarizations.	Database
4	Summarizations	GET	common/object-summaries/{object_id}	Retrieve a specific object summarization by object ID.	Database
4	Classifications	POST	/classifiers	Add to Classifiers table, one row for classifiers	Database
4	Classifications	POST	common/etd_classes	Add to ETD_classes table to save the results	Database
4	Classifications	PUT	common/etd_classes	Update an existing ETD classification entry.	Database
4	Classifications	GET	common/etd_classes	Retrieve a list of all ETD classifications.	Database
4	Classifications	GET	/common/etd-classes/{etd_id}	Retrieve a specific ETD classification by ETD ID.	Database
4	Classifications	GET	/objects/{objectId}	Get from Object text data from the object table	Database
4	Classifications	POST	/common /object-classes	Add to Object_classes to save the classification results/ Create a new object classification entry.	Database
4	Retrieve from object classification	GET	/common/object-classes/{object_id}	Retrieve a specific object classification by object ID.	Database
4	Update existing object classification	PUT	/object-classes/{object_id}	Update an existing object classification entry.	Database

4	Retrieve a list of all object classifications.	GET	/object-classes	Retrieve a list of all object classifications.	Database
1	Back-end server	POST	/etd/new	Add new ETD to the database and upload the file to the file system	File system
1	Back-end server	POST	/topic-modeling/results	Save topic modeling results	Database
1	Back-end server	GET	/etd/v2/objects	Get ETD and its objects v2	Database
1	Back-end server	GET	/etd/{etdId}	Get ETD by ETD ID	Database
1	Back-end server	GET	v1/team-1/graphQL	Get data from various tables, complex queries and unions	Database
3	Topic modeling	GET	/etd/{etdId}/topic-vector	Collect dataset etd_id and return Topic vector	Database
3	Topic modeling	GET	/etd/{etdId}/related-documents	Collect dataset etd_id and return the related documents	Database
3	Topic modeling	GET	/etd/{etdId}/related-topics	Collect dataset etd_id and return the related Topics	Database
2	Search service backend	POST	/api/ v1/search	API to take in search query, search method and search by and return ETD metadata as search result	Database
2	Search service backend	GET	/api/ v1/documents/<etd_id>	API to fetch ETD metadata by its ID, used by front-end in the Document View	Database
2	Search service backend	GET	https://search.endeavour .cs.vt.edu/api/v1 /logs/<user_id>	API to fetch logs of a specific user by the user ID. Used by recommendation service	Database
2	Recommendation service Backend	POST	https://recommendation .endeavour.cs.vt.edu/api/ v1/recommendation/infer	Takes in user_id to infer recommendations for the user.	Database
2	Recommendation service Backend	POST	https://recommendation. endeavour.cs.vt.edu/api/v1 /recommendation/train	Takes in ETDs read of all user and ETD data to train a model for recommendation	Database
2	Search service backend	POST	https://search.endeavour. cs.vt.edu/api/v1/ experiment/create	Creates a search experiment with given parameters from front-end search experimenter page	Database
2	Search service backend	POST	https://search.endeavour. cs.vt.edu/api/v1/ experiment/delete	delete a search experiment by taking its name as input	Database
2	Search service backend	POST	https://search.endeavour. cs.vt.edu/api/v1/ experiment/search	Searches over a created experiment index by taking in query parameters from the search experimenter page	Database
2	Search service backend	POST	https://search.endeavour. cs.vt.edu/api/v1 /index	Indexes the ETD metadata into Elasticsearch, which is used by workflow automation.	Database
2	Search service backend	GET	https://search.endeavour. cs.vt.edu/api/v1 /chapters/<chapter_id>	Gets the respective chapter-by-chapter ID from Elasticsearch	Database
2	Search service backend	POST	https://search.endeavour. cs.vt.edu/api/v1/ chapters/search	Searches over all the chapters stored in Elasticsearch	Database

6	User management backend & Frontend	POST	/auth/register	API to create a new user and store user details in db	Database
6	User management backend & Frontend	GET	/users/<profile>	API to fetch a user profile by user id from db	Database
6	User management backend & Frontend	PUT	/users/<update>	API to update a user profile by user id	Database
6	User management backend & Frontend	PUT	/api/v1/users_roles/<update>	API to update a user role by user id to other status	Database
6	User management backend & Frontend	POST	https://team5-api.endeavour.cs.vt.edu/api/v1/users_topics/<post>	"User_topics" to save their topic information	Database
6	User management backend & Frontend	POST	https://team5-api.endeavour.cs.vt.edu/api/v1/users_classes/<post>	"User_classes" to save their classes information	Database
2	Search service backend	GET	/team2/etd-metadata/1437	To get ETD metadata from ETD_Metadata table	Database
2	Search service backend	GET	https://team5-api.endeavour.cs.vt.edu/api/team2/get_etds-topics	Topic modelling data - ETD_topics	Database
2	Search service backend	GET	/api/team2/etd_etd-neighbours	Get from ETD-ETD neighbors table	Database
2	Search service backend	POST	/api/team2/post-etd_etd-neighbours	Post to ETD-ETD neighbors table	Database
2	Search service backend	GET	/api/team2/get_object-metadata	Get Object Metadata - For Image captions - Object_Metadata table	Database
2	Search service backend	GET	/api/team2/get_object-summaries	Get summaries - Object_summaries table	Database
2	Search service backend	GET	/api/team2/get_collections	Collections table	Database
2	Search service backend	GET	/api/team2/get_object-to-object-neighbours	Get from Object-object neighbors	Database
2	Search service backend	POST	/api/team2/get_object-to-object-neighbours	Post to Object-object neighbors	Database
2	Search service backend	GET	/api/team2/get_user-modules	Get from User_classes, User_queries_clicks, User_topics,	Database
2	Search service backend	GET	/api/team2/get_user-user-neighbours	User-user neighbors	Database
2	Search service backend	POST	/api/team2/get_user-user-neighbours	User-user neighbors	Database

Table 6.1: List of APIs tailored as per each team's requirements

Bibliography

[1] Slack Technologies LLC. *Slack*. 2023. URL: https://slack.com/features. (accessed: 08.24.2023).

[2] Zoom Video Communications Inc. *Zoom*. 2023. URL: https://explore.zoom.us/en/about/. (accessed: 08.24.2023).

[3] Docker. *What is DevOps*. 2023. URL: https://aws.amazon.com/devops/what-is-devops/. (accessed: 08.30.2023).

[4] The Linux Foundation. *What is Ingress*. Dec. 2023. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/. (accessed: 08.22.2023).

[5] The Linux Foundation. *What is a Pod*. Dec. 2023. URL: https://kubernetes.io/docs/concepts/workloads/pods/. (accessed: 08.22.2023).

[6] RedHat. *What is CI/CD?* 2023. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd. (accessed: 10.18.2023).

[7] GitLab. *GitLab Runner*. 2023. URL: https://docs.gitlab.com/%20runner/. (accessed: 10.18.2023).

[8] Alexander Hicks et al. "Integration and Implementation (INT) CS 5604 F2020". In: Virginia Tech team term project submission, Blacksburg, VA 24061, Dec. 2020. URL: http://hdl.handle.net/10919/101544.

[9] Tigera Inc. *Network Policy Operator*. 2023. URL: https://docs.tigera.io/calico/latest/about/. (accessed: 12.03.2023).

[10] Vasundhara Gowrishankar. *CI-CD Guide*. 2023. URL: https://canvas.vt.edu/courses/176258/files/30978135?wrap=1. Virginia Tech CS5604 Fall 2023 internal class report (accessed: 11.10.2023).

[11] R. Martin et al. "Container as a Service: Understanding and Evaluating the Emerging Container-based Virtualization Paradigm". In: *IEEE Transactions on Cloud Computing* 5.2 (2017), pp. 361–374.

[12] A. Sharma and S. Khattar. "Container as a Service (CaaS): A Systematic Literature Review". In: *International Journal of Advanced Computer Science and Applications* 10.7 (2019), pp. 489–495.

[13] B. Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14.1 (2016), pp. 70–93.

[14] P. Arora and A. Chaudhary. "A Survey on Kubernetes Security: Vulnerabilities, Threats, and Countermeasures". In: *2018 3rd International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU)*. 2018, pp. 1–6.

[15] Zachary Tong and Clinton Gormley. *Elasticsearch: The Definitive Guide*. O'Reilly Media, 2015. URL: https://www.elastic.co/guide/en/elasticsearch/guide/current/index.html.

[16] Gaurav Modi et al. "Elasticsearch: A search engine for structured and unstructured data". In: *International Research Journal of Engineering and Technology (IRJET)* 5.2 (2018), pp. 2345–2350. URL: https://www.irjet.net/archives/V5/i2/IRJET-V5I2692.pdf.

[17] Real Python. *Jupyter Notebook: An Introduction*. 2023. URL: https://realpython.com/jupyter-notebook-introduction/. (accessed: 09.03.2023).

[18] Martin Fowler. "Continuous Integration". In: *martinfowler.com* (2018). URL: https://martinfowler.com/articles/continuousIntegration.html.

[19] GitLab Inc. *GitLab - The first single application for the entire DevOps lifecycle*. Accessed 2023. URL: https://about.gitlab.com/.

[20] Scott Chacon and Ben Straub. *Pro Git*. 2nd ed. Apress, 2021. ISBN: 978-1484250323.

[21] Anmol Shukla et al. "CS5604 Fall 2022 - Team 5 INT". In: Virginia Tech team term project submission, Blacksburg, VA 24061, Jan. 2023. URL: http://hdl.handle.net/10919/114078.

[22] Tyler Charboneau. *How to Use the Postgres Docker Official Image*. Oct. 2022. URL: https://www.docker.com/blog/how-to-use-the-postgres-docker-official-image/. (accessed: 10.01.2023).

[23] Virginia Tech Department of Computer Science. *Computer Science Cloud at VT*. 2023. URL: cloud.cs.vt.edu. (accessed: 08.22.2023).

[24] Docker. *Docker Hub Jupyter Notebook*. 2023. URL: //hub.docker.com/r/jupyter/scipy-notebook. (accessed: 08.31.2023).

[25] Google. *Container Registry documentation*. 2023. URL: https://cloud.google.com/container-registry/docs. (accessed: 08.25.2022).

[26] Michael Bose. *Containerization*. 2023. URL: https:%20//www.nakivo.com/blog/docker-vs-kubernetes/. (accessed: 09.23.2023).

[27] Rancher. *Rancher*. 2023. URL: https://rancher.com/. (accessed: 09.04.2023).

[28] Ceph authors and contributors. *Intro to CEPH*. 2016. URL: https://docs.ceph.com/en/latest/start/intro/. (accessed: 10.18.2023).

[29] Amazon Web Services Inc. *What is an API*. 2023. URL: https://aws.amazon.com/what-is/api/. (accessed: 08.29.2023).

[30] ElasticSearch. *What is ElasticSearch*. 2023. URL: https://aws.amazon.com/what-is/elasticsearch/. (accessed: 09.05.2023).

[31] GitLab Inc. *What is version control?* 2023. URL: https://about.gitlab.com/topics/version-control/. (accessed: 09.01.2023).

[32] CS5604Fall2023. *CS5604-F2023 Group Project Code*. 2023. URL: https://code.vt.edu/dashboard/groups. (accessed: 11.10.2023).

[33] Tim Bray. "The JavaScript Object Notation (JSON) Data Interchange Format". In: RFC8259 (2017), p. 16. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[34] Miguel Angel García and Gonzalo Camarillo. "Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists". In: RFC5364 (2008), p. 17. DOI: 10.17487/RFC5364. URL: https://www.rfc-editor.org/info/rfc5364.

[35] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[36] OpenID Foundation. *What is OpenID Connect*. 2023. URL: https://openid.net/developers/how-connect-works/. (accessed:2023).

[37] Tech Monger. *How to store passwords securely using Werkzeug*. Nov. 2017. URL: https://techmonger.github.io/4/secure-passwords-werkzeug/.

[38] SmartBear Software. *API Keys*. 2023. URL: https://swagger.io/docs/specification/authentication/api-keys/. (accessed: 12.03.2023).

[39] provectus. *Kafka monitor*. 2023. URL: https://github.com/provectus/kafka-ui. (accessed: 12.03.2023).

[40] SQLAlchemy. *What is SQLAlchemy*. Dec. 2023. URL: https://www.sqlalchemy.org/. (accessed: 08.03.2023).