Final Report
CS 5604: Information Storage and Retrieval

Integration team (INT):

Aaron Travasso, Anmol Shukla, Harish Babu Manogaran,
Pallavi Sisodiya, Yuze Li

Subject Matter Expert:

Dhanush Dinesh

January 8, 2023

Instructed by Professor Edward A. Fox

Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

The primary objective of the project is to build a state-of-the-art system to search and retrieve relevant information effectively from a large corpus of electronic theses and dissertations. The system is targeted towards documents such as academic textbooks, dissertations, and theses where the information available is enormous, compared to websites or blogs, which the conventional search engines are equipped to handle effectively. The entire work involved in developing the system has been divided into five areas such as data management (Team-1, Curator); search and retrieval (Team-2, User); object detection and topic analysis (Team-3, Objects & Topics); language models, classification, summarization, and segmentation (Team-4, Classification & Summarization); and lastly integration (Team-5, Integration). The teams and their operations are structured in a way to mirror an environment of a company working on new product development. The Integration (INT) team focuses on important aspects such as setting up work environments with all requirements for the teams, integrating the work done by the other four teams, and deploying suitable Docker containers for seamless operation (workflow), along with maintaining the cluster infrastructure. The INT team archives this distribution of code and containers on the Virginia Tech Docker Container Registry and deploys it on the Virginia Tech CS Cloud. The INT team also guides team evaluations of prospective container components and workflows. Additionally, the team implements continuous integration and continuous deployment to enable seamless integration, building, and testing of code as it is developed. Furthermore, the team works on setting up a workflow management system that employs Apache Airflow to automate creating, scheduling, and monitoring of workflows. We have created customized containers for each team based on their individual requirements. We have developed a workflow management system using Apache Airflow that creates and manages workflows to achieve the goals of each team such as indexing, object detection, segmentation, summarization, and classification. We have also implemented a Continuous Integration and Continuous Deployment (CI/CD) pipeline to automatically create, test, and deploy the updated image whenever a new push is made to a Git repository. Additionally, we extended our support to other teams in troubleshooting the issues they faced in deployment. Our current cluster statistics (i.e., Kubernetes Resource Definitions) are: 45 deployments, 40 ingresses, 39 pods, 180 services, and 13 secrets. Lastly, the INT team would like to express its gratitude to the work of the INT-2020 team and the predecessors who have done substantial work upon which we built. We would like to acknowledge here their significant contribution.

Contents

List of Figures

List of Tables

Chapter 1

Overview

Our entire system consists of the following components.

1. Developer Operations: We have created containerized environments for each team. Continuous Integration and Continuous Deployment have also been a major focus of our DevOps tasks. We have developed a pipeline that automatically deploys the latest image for any code that has been pushed to the master branch of the repository. It also has the provision of adding testing that can be done before deployment.

2. Workflow Management: The Workflow Management consists of 3 parts - the Registry UI (Section 6.1.3), the Registry Backend (Section 6.1.6), and the Airflow workflow automation system (Section 6.1). The Registry UI is aimed to allow the different teams to view, modify, or add more services, goals, and workflows to our Registry database. The Registry backend handles making these API calls and correctly registering this data in the database. The reasoner table, which holds the relationships between Goals and Services, is used to create a context-free grammar. This grammar, which contains production rules, is used to create workflows. The reasoner contains all the code to convert the reasoner table to a context-free grammar, and then from production rules, which are generated as a result of the context-free grammar, to actual workflows. The workflows are generated in the form of DAG files. The Airflow System brings the workflow to life by running the DAG file and spinning up all the required services mentioned within the file to accomplish the results of the workflow.

3. DevOps Support: Alongside setting up the above, the INT team members have created guidelines and documentation to support other teams. We have also connected each of our team members to other teams to set up their containerized environments along with any other setup requirements and troubleshooting they may require.

1.1 Collaboration and Communication

We have set up efficient collaboration and communication structures that involve all five people in our team. Our internal collaboration model is described in detail, in Section 6.3, which talks more about how we picked up a Human-Computer Interaction related monograph [4] that prescribes user-centered design techniques using Agile methods to define our project management strategy. With regards to communication, we use a Discord [24] server, called 'DevOps-Team' with dedicated channels to have all our conversations and provide any

context. We usually meet in person but have accommodations for people who may not be able to make it to the venue via Zoom [26]. We have dedicated liaisons for each team, details regarding which are communicated via our team page on Canvas. We use the class sessions as synchronous liaisoning sessions to communicate and collaborate with other teams. We have used these sessions to get requirements and also offer support to other teams. We use email, the class Discord channel, and our Canvas page for all asynchronous communication.

1.2 Problems Faced

We faced the following problems in the first and second phases of implementation. Their solutions are discussed in Section 1.3:

1. Understanding which style of project management we must use, user requirements, and gaining those requirements from the respective teams.

2. We were informed that the storage that was initially provisioned for us (Ceph [3]) is not going to be enough to store all the ETD data and will significantly fall short of it.

3. We were informed by Teams 3 and 4 that they were unable to install dependencies on the Docker containers that we had spun up for them and were facing storage issues. This happens because the disk space on the container is provisioned at the time of creation and when the dependencies are installed after the container has been spun up, this results in disk space issues.

4. Two of our original team members, who were liaisons to Team-3 and Team-4, stopped taking the course. Unfortunately, this happened after the team liaisons were allocated and some work had been scheduled and assigned.

5. Some of our team members did not have prior experience working with cloud technologies such as Kubernetes, Docker, etc.

6. We faced problems with deploying GitLab onto Kubernetes, as within a container it cannot gain root access to the filesystem, leading to various permission errors.

7. We faced problems with detecting GPU nodes of the CS cloud infrastructure from the clusters of Team-3 and Team-4, which are necessary for training and running machine learning models.

8. We faced issues with understanding and integrating the reasoner implementation of Team INT-2020 into our work due to a lack of sufficient documentation of the design choices they made.

9. Downtime of the Cloud Environment sometimes led to a pause in all team's work. Since this is managed by the CS Department, we had to communicate with them to keep it up and running.

1.3 Solutions Developed

We have proceeded to approach problems as described above and as instructed by Professor Edward Fox, by gaining user requirements, identifying their goals, and designing our solution around them. We have worked along with our subject matter expert Mr. Dhanush Dinesh, as well as the teaching assistant for the course, Mr. Hemayet Ahmed Chowdhury. They have helped us outline and provide guardrails for the process. They in turn put us in touch with Mr. Chris Arnold, who manages both Ceph and Camelot. We also build on the work done by the team before us, who took the course in Fall 2020 [22], as they have provided some directions to us in terms of the solution and architecture design. We worked to improve upon their solution. Specifically, in response to some of the problems mentioned above, we came up with the following solutions, numbered to match the list of problems from the previous section.

1. The solution to this problem is described in detail in Section 6.3. However, to mention it briefly, we decided to use Scrum as the chosen framework for project management. Scrum also details how to identify user requirements.

2. To get around the problem of limited storage on Ceph, we were provided with Camelot [43] as an alternative storage solution.

3. We solved the problem of low disk space on Docker containers by specifying Docker files that contained the published image with pre-installed dependencies so that at the time of image pulling, Docker will allocate enough space for the base binary as well as any dependencies.

4. We solved the problem of the reduced headcount by re-assigning one of the liaisons for Team-4 over to Team-3. We accounted for the loss of headcount during October by dividing up the work between Team-3 and Team-4 liaisons by putting these teams under the same umbrella, given the similarity in the teams' nature of work.

5. We got around the problem of missing knowledge by taking onboarding sessions and studying suggesting readings.

6. We overcame the problem with deploying GitLab onto Kubernetes by deploying GitLab on a dedicated Virtual Machine. Further, the parameters 'privileged' and 'volumes' in the GitLab runner were updated as below.

```
privileged = true
volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]
```

7. For the containers to detect the GPUs of CS cloud infrastructure which is under the label 'gpu', the nodeAffinity constraint is set in the Kubernetes YAML file to the label 'gpu' so that the deployments of Team-3 and Team-4 containers are tied to the nodes under the particular label.

8. To understand and integrate Team INT-2020's reasoner into our system we sought the help of Mr. Mohit Thazhath, from Team INT-2020, and Dr. Prashant Chandrasekar at UMW, to help us understand their design choices.

Chapter 2

Requirements

2.1 General Requirements

Regarding the whole ETD project, our team should liaise with other teams and containerize all team's code that runs on cloud.cs.vt.edu. We should make sure we have continuous integration / continuous deployment (CI/CD), with a test routine for each service, and a test routine for each frequently run workflow, so when changes are checked in, there will be automated service and workflow testing, leading to immediate deployment if tests are successful. This CI/CD approach should lead to a considerable speedup in our system building, testing, and experimentation. Following is a list of requirements/objectives.

1. Search will be performed on Electronic Theses and Documents.
2. To facilitate this search, classification and other methods such as object detection will be used.
3. Another workflow would be chapter or full-text summarization.
4. Provide file systems to other teams to store the actual files and a database (Postgres) to store all the metadata.
5. Provide appropriate levels of access to data.
6. Tie all these together with the help of web servers running in Containers, Databases, Reasoner Engine, and Machine Learning models aided by GPUs.
7. Finally, using CI/CD to automatically deploy code and use automated test cases with the help of pipelines.

2.2 User Personas

User personas are built as a way to understand our requirements as we serve teams 1, 2, 3, and 4. These user personas will be more abstract and less 'open' for Teams 3 and 4, as compared to Teams 1 and 2. Moreover, these personas are implicit, as in they do not reflect the exact nature of the user and their requirements. Following are some of the user personas that we have collected from other teams.

- **Team 1**
    1. As an engineer, I want to deploy my backend/frontend code and any storage to cloud servers.
    2. As an engineer, I want to automatically test and integrate my code with the one stored on the cloud.
    3. As a curator, I should be able to curate digital objects such as ETDs and metadata objects.
    4. As an engineer, I should be able to make use of a Knowledge Graph to support question answering and managing the relationships between metadata and digital objects.
- **Team 2**
    1. As an engineer, I want a cloud platform that helps me pre-process, index, and support the search of data.
    2. As an engineer, I want to deploy my front-end web application for searching, and to make it available to my end user.
    3. As an engineer, I should be able to set up the system in a way that it can answer questions.
- **Team 3 and Team 4**
    1. As an Experimenter, I want a platform to train a machine-learning model to perform object detection and topic modeling.
    2. As an Experimenter, I should be able to run experiments using these models and compare their results for optimal use.
    3. As an Experimenter, I want access to the training data for my model.
    4. As an Experimenter, I want to offload the results of my trained model to a persistent storage
    5. As an Experimenter, I should be able to perform chapter segmentation, text summarization, and classification of ETDs.
    6. As an Experimenter, I should be able to perform object detection and topic modeling on ETDs.

2.3 Workflow Automation Requirements

The workflow automation system is centered around executing various tasks regarding Electronic Theses and Dissertations using services provided by other teams. The task execution

will be abstracted into a Directed Acyclic Graph (DAG), fulfilled by a workflow engine called Apache Airflow. Following is a brief overview of Apache Airflow:

1. Developers should be able to add goals, workflows, and services metadata, and test their functionalities.

2. Experimenters and Developers should be able to see and execute the goals provided by other developers.

3. Workflows (expressed as Directed Acyclic Graphs) should be automatically generated seamlessly.

4. Containerization of the Airflow engine and PostgreSQL database (that holds relevant workflow tables) should be in place.

5. Appropriate data parsing should occur from the metadata tables to generate DAGs.

6. Use of a scalable indexing engine would be desirable to facilitate optimized querying of information.

7. The system should be straightforward to use.

8. A Curator should be able to add a new ETD, so that its digital objects along with the document itself should be added to the already-existing collection of ETDs.

2.4 Performance

While we do not expect this system to be able to serve millions of people, we do expect that our basic architecture is scalable and parallel enough to prevent major bottlenecks. We will make sure, even if there are hundreds of services or workflows, that query speed will not be largely affected. We have used an appropriate data tables design, and incorporated various optimizations, such as load balancing by Kubernetes and for indexing. These are just two of the key things implemented for better performance.

2.5 User Support

The Workflow System consists of 5 elements: Reasoner, Apache Airflow, Registry UI, Registry Backend server, and the PostgreSQL Database that stores all goals and services (and their linkage). To generate or run workflows, various APIs have been exposed. We have built a UI that makes use of Workflow APIs such as 'Add Workflow' and 'Edit Workflow'

for Experimenters to add new goals. This UI makes it easy to visualize and edit workflows and also to add new services.

Apart from this, we have also provided teams with an easy way to integrate this system into their frontend applications by the way of a code wrapper that is written in JavaScript, as is discussed more in Section 6.1.4.

Chapter 3

Literature Review

3.1 Cloud-based Service Provisioning

Cloud-based computation resources are offered in a vertical model, with each layer in the model operating at a different level of abstraction [17]. The layers in this model are:

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

We decided to choose CaaS as our level of abstraction. This involves spinning up the containers and their orchestration. We chose Docker and Kubernetes for this job, respectively (discussed more in Section 4).

3.1.1 Selecting the Level of Abstraction

We have chosen "Container as a Service" as our desired level of abstraction. The reasoning behind this is as follows. Our requirements, as discussed in Section 2, have us supporting other teams in the way of offering a platform for their applications. We must give them autonomy in choosing their technological stack, while at the same time minimizing the amount of effort across all teams. Unfortunately, IaaS and PaaS, despite their flexibility, usually entail more work on the part of the team in the form of initial set-up. For example, with IaaS, each team would get a compute node hosted on the cloud, but would then have to set up networking, storage, security, and other aspects associated with the compute node. On the other hand, using SaaS or FaaS would limit the flexibility in terms of choosing the technological stack which the teams might want to work with. For these reasons, we conclude that CaaS is situated in a sweet spot of the trade-off range between flexibility and effort.

3.1.2 Containers as a Service

Containers-as-a-Service (CaaS) is a cloud service that helps manage and deploy apps using container-based abstraction. CaaS can be deployed on-premises or in a cloud [41]. CaaS is especially useful to developers in building containerized apps that are more secure and also scalable. Users can buy only the resources they want (scheduling capabilities, load balancing, etc.), saving money and increasing efficiency. Containers create consistent environments to rapidly develop and deliver cloud-native applications that can run anywhere.

Compared to IaaS platforms that use virtual machines or bare metal hardware as fundamental resources, CaaS platforms package up applications and all their dependencies into containers more lightweight than virtual machines. For this reason, it is possible to host more containers on a single host than full-fledged virtual machines [5]. CaaS is a suitable platform for developers who want more control over container orchestration. Using CaaS allows developers to deploy applications on containers without worrying about the limitations on container orchestration provided by the typical Platform-as-a-Service (PaaS).

The basic resources of CaaS are containers, which are a popular deployment mechanism for cloud-native apps and microservices. CaaS also increases portability between environments, whether hybrid or multi-cloud.

Using containers has many benefits:

- **Efficiency**: Containers require fewer resources than virtual machines (VMs) since they don't need a separate operating system. One can run several containers on a single server. They require less bare-metal hardware, which means lower costs.
- **Portability**: Apps developed in containers have everything they need to run and can be deployed in multiple environments, including private and public clouds. Portability means flexibility because one can more easily move workloads between environments and providers.
- **Scalability**: Containers can scale horizontally, meaning a user can multiply identical containers within the same cluster to expand when needed.
- **Increased security**: Containers are isolated from each other, which means if one container is compromised, others won't be affected. A container allows incorporating specific security requirements and practices into golden images to maintain the security posture even in different cloud environments and different footprints.

3.1.3 Kubernetes

Kubernetes is regarded as one of the most trending container orchestration services [37] and is used by the CNCF (Cloud Native Computing Foundation) as their orchestration

technology [10]. Kubernetes can give the orchestration container management capabilities required to deploy containers at scale across multiple server hosts with multiple layers of security while managing the health of those containers over time. In Kubernetes, a node is a worker machine; it may be a virtual machine or a physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components of Kubernetes.

In this project, we use `cloud.cs.vt.edu` which offers us a Discovery cluster to build the ETD project. We offer each team several containers to build their services, and we use Kubernetes for orchestration.

3.2 Database Technologies

3.2.1 Criteria for Choice of Database Management System

The primary user of the database is Team-1, as they're responsible for the central persistence of the results of all other teams. As per their requirements, we required a database management system that is:

1. Relational in nature
2. ACID compliant [48]
3. Well-tested, mature, and with community support
4. Scalable (as we are going to be storing a considerable amount of ETD data)

Some of the candidates that were considered are PostgreSQL [47], MySQL [34], MS SQL [32], and Oracle DB [25], out of which we decided to go with PostgreSQL for the reasons given in Section 3.2.2.

3.2.2 PostgreSQL

PostgreSQL is a free and open-source object-relational database management system that is used as one of the chief databases across various web and mobile applications [36]. It is well known for its reliability, stability, and performance. The tool, along with providing support for a variety of advanced data types, also provides well-optimized performance on par with other DBMS tools like MS SQL and Oracle SQL.

There are multiple benefits to using PostgreSQL:

- **Open source**: PostgreSQL is a completely free and open source tool, which provides the freedom and flexibility to modify and implement as per the particular application requirements.
- **Reliable**: PostgreSQL is ACID (Atomicity, Consistency, Isolation, and Durability) compliant, thereby ensuring that all transactions are atomic and the data consistency is retained in case of failures by rolling back all the changes made. Further, due to massive contributions to its maintenance from various companies and individuals across the world, any bugs found are resolved immediately.
- **Scalable**: PostgreSQL scales well vertically and performs better as more computing resources are added.

In this project, we relieve the query workload of both end-users and developer-users by allowing them to access and manage the PostgreSQL database on a web browser, instead of writing tedious SQL statements.

3.3 Workflow Automation

We aim to achieve workflow automation to help other teams automate various tasks as services and workflows, to increase the overall productivity of the group.

3.3.1 Workflows as Directed Acyclic Graphs

A directed acyclic graph (DAG), is a finite directed graph with no directed cycles. That is, it consists of finitely many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex 'v' and follow a consistently-directed sequence of edges that eventually loops back to 'v' again. Equivalently, a DAG is a directed graph that has a topological ordering, a sequence of the vertices such that every edge is directed from earlier to later in the sequence [11]. We show an example in Figure 3.1, which is one of our workflow automation designs.

A workflow is a series of steps taken to achieve a goal from a set of inputs. Workflows usually have an end goal, for example, creating visualizations for sales numbers of the last day. Thus, workflows can be imagined as DAGs, where each node is an executable step. For example, to create your visualization from the past day's sales, the following steps may need to be performed:

1. Read data from a relational database.
2. Transform the data by clipping irrelevant columns or attributes.

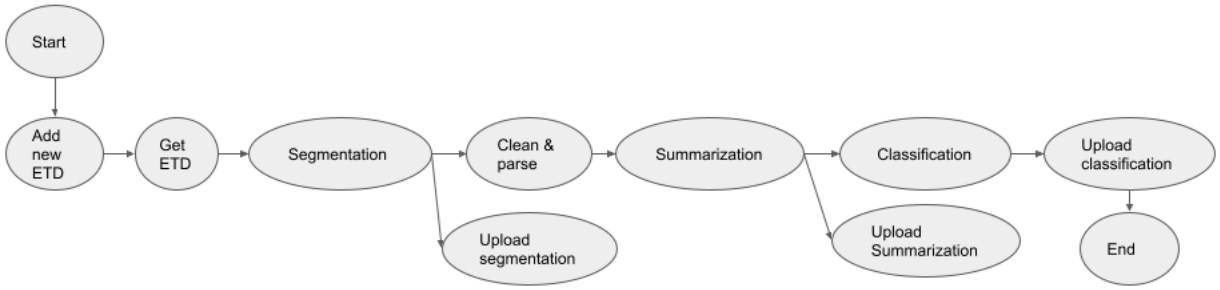

Figure 3.1: A DAG Sample

3. Load this data into a data warehouse tool.
4. Trigger the fetching of this data from the visualization tool.

Thus, if we were to represent this as a DAG, it would look as depicted in Figure 3.2.

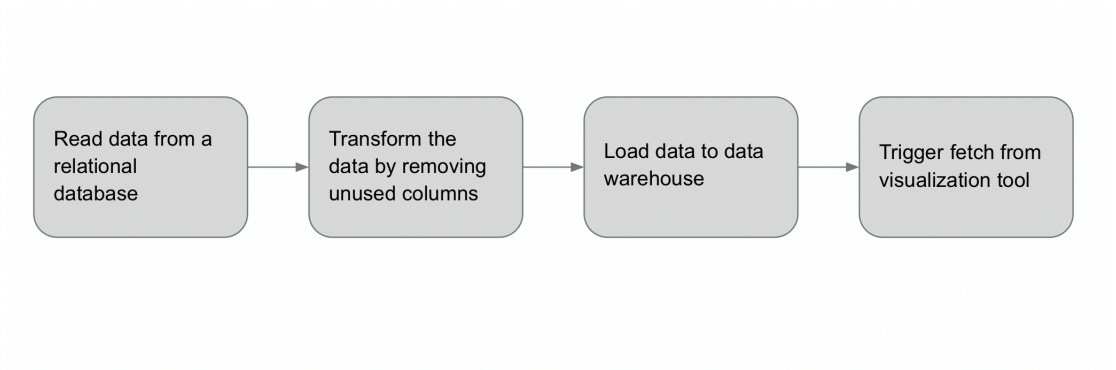

Figure 3.2: A workflow represented as a DAG

3.3.2 Apache Airflow as a Workflow Automation Tool

Apache Airflow [1] is a platform that lets you build and run workflows. A workflow is represented as a DAG, and contains individual pieces of work called Tasks, arranged with dependencies and data flows taken into account [2].

An Airflow installation generally consists of the following components:

1. A scheduler, which handles both triggering scheduled workflows, and submitting Tasks to the executor to run.
2. An executor, which handles running tasks. In the default Airflow installation, this runs everything inside the scheduler, but most production-suitable executors push task execution out to workers.

3. A web server, which presents a handy user interface to inspect, trigger, and debug the behavior of DAGs and tasks.

4. A folder of DAG files, read by the scheduler and executor (and any workers the executor has).

5. A metadata database, used by the scheduler, executor, and web server to store state.

In Airflow, an operator encapsulates the operation to be performed in each task in a DAG. Airflow has a wide range of built-in operators that can perform specific tasks, some of which are platform-specific. For example, a 'BashOperator' executes a command inside a Bash shell on the machine that is running Airflow. Additionally, it is possible to create customized operators. The dependencies of operators could be defined as follows: first_task » [second_task, third_task]. To execute a DAG, users can either operate through CLI or a DAG user interface.

In this project, to accomplish workflow automation, we use Airflow to execute DAGs generated by the Knowledge Graph and Reasoner. We explain this further in Section 5.3.

3.4 Continuous Integration and Continuous Deployment in GitLab

Continuous Integration and Continuous Deployment or 'CI/CD', is "a method to frequently deliver apps to customers by introducing automation into the stages of app development" [42], put simply. CI/CD is defined declaratively as a pipeline. This pipeline takes in code, performs some commands, and deploys it. Each of the 'Continuous' steps has certain stages

Figure 3.3: The General CI/CD Pipeline Workflow

as explained below. Refer to 3.3 for a Visual Representation.

1. Continuous Integration: Continuous Integration focuses on automating code integration by automatically testing the code, ensuring that it is bug-free, and integrating it with a Version Control System, typically hosted in the Cloud. This speeds up the code merge process as it no longer needs to be merged and built on a developer's machine and is instead built in the cloud.

2. Continuous Delivery: Continuous Delivery is the automatic delivery of code to various testing environments such as development and staging for quality assurance (QA) purposes.

3. Continuous Deployment: This is the final step in the DevOps process wherein the code, after testing and building, is pushed to the server in the form of build artifacts.

These stages need to be in the same order but a typical workflow may not see all the stages. For example, we can compare the concept of 'Merge Request' builds with a typical workflow. In it, the merge is tested with the help of automated testing before it can be merged into another branch.

The following are some of the advantages of CI/CD [46].

- Automated testing enables continuous delivery, which ensures software quality and security, and increases the profitability of code in production.
- CI/CD pipelines enable a much shorter time to market for new product features, creating happier customers and lowering strain on development.
- The great increase in overall speed of delivery enabled by CI/CD pipelines improves an organization's competitive edge.
- Automation frees team members to focus on what they do best, yielding the best end products.
- Organizations with a successful CI/CD pipeline can attract great talent. By moving away from traditional waterfall methods, engineers and developers are no longer bogged down with repetitive activities that are often highly dependent on the completion of other tasks.

GitLab has an inbuilt solution for CI/CD called GitLab Runner which runs and deploys the code. The GitLab Runner can be installed on our infrastructure which means that the runner should run on an isolated VM which is only dedicated to the runner [18]. This is due to performance issues.

The runner is open source and hence, the source code is publicly accessible at https://gitlab.com/gitlab-org/gitlab-runner/.

Pipeline code can be executed by various 'methods' that GitLab calls as executors. The following is the list of executors [19]:

- SSH
- Shell

- Parallels
- VirtualBox
- Docker
- Docker Machine (auto-scaling)
- Kubernetes
- Custom

For our implementation, we have used 'Docker' as our executor which means it provides a clean build environment without having to install any tools on the host machine.

3.5 Grafana

Grafana [31] is an open-source solution for running data analytics, pulling up metrics that make sense of the massive amount of data to monitor our apps with the help of customized dashboards. Grafana, being an open-source solution, also enables us to write plugins from scratch for integration with several different data sources. The tool helps us study, analyze, and monitor data over some time, technically called time series analytics. It helps us track the user behavior, application behavior, frequency of errors popping up in production or a pre-prod environment, type of errors popping up, and contextual scenarios – by providing relevant data.

Chapter 4

Components

4.1 Docker and Containers

For this project, we are leveraging the CS cloud Infrastructure [35]. Docker is an open-source platform that enables developers to build, deploy, run, update, and manage containers—standardized, executable components that combine application source code with the operating system (OS), libraries, and dependencies required to run that code in any environment (Figure 4.1). It benefits both system administrators and software developers, which is a crucial part of DevOps toolchains.

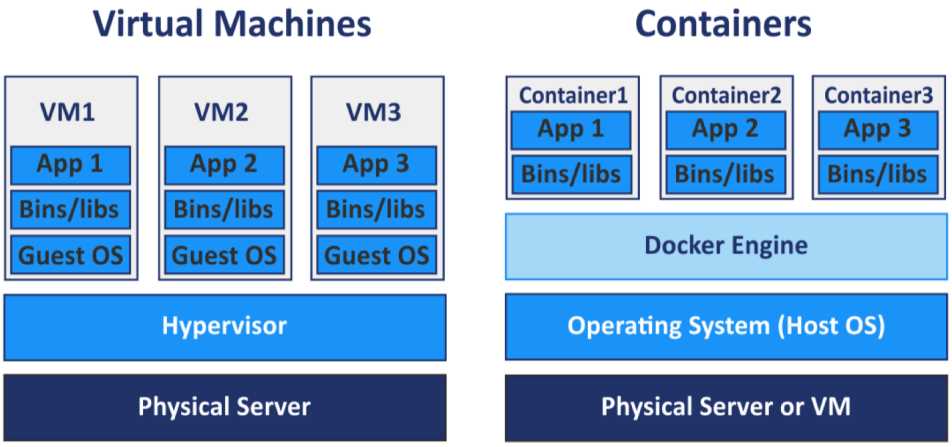

Figure 4.1: Container is on top of OS [6]

Containers are created from customized images that are stored in a container registry both privately and publicly such as Docker Hub, and the Google Container Registry [20]. Virginia Tech also has its container registry.

We will create, deploy and maintain containers based on the requirements of each team. To meet each team's specific requirements, we also require (and help) each team to write a Dockerfile for fast deployment. A Dockerfile is a text document that contains all of the commands a user could call on the command line to assemble an image [15]. In the VT CS cloud [13], we use a cluster called *Discovery* on which we deploy all containers.

4.2 Jupyter Notebooks

The Jupyter Notebook is an open-source web application that can be used to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Project Jupyter [39].

Jupyter Notebooks are a spin-off project from the IPython project, which used to have an IPython Notebook project itself. IPython is a growing project, with increasing language-agnostic components (Figure 4.2). The name, Jupyter, comes from the core supported

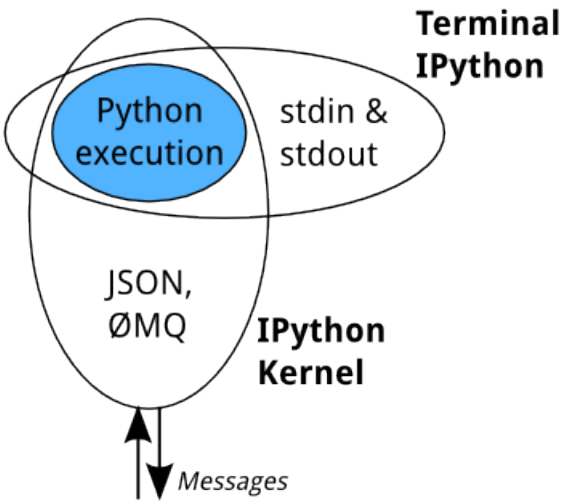

Figure 4.2: IPython Structure [28]

programming languages that it supports: Julia, Python, and R. Jupyter ships with the IPython kernel, which allows one to write programs in Python.

We offer each team containers with Jupyter Notebook embedded that allow them to access a remote server and modify files easily, instead of deploying and testing locally. We use a Docker image file from Docker Hub [14] to quickly spin up containers inside the CS cloud. Detailed steps will be shown in Section 7. Jupyter Notebooks are used as computational documents by the teams.

4.3 PostgreSQL

We use PostgreSQL (Postgres) [47] as our database for the services available as well as holding the configuration and data for Airflow. As is described in Section 3.2.2, PostgreSQL is an advanced, enterprise-class, and open-source relational database system that supports both

SQL (relational) and JSON (non-relational) querying. Postgres acts as the central database for ETD profiling. It holds the metadata for ETDs such as title, date issued, authors, types, and URLs uploaded by the curator team. The experimental teams are using APIs to fetch the metadata and analysis, segment, train models, and save profiled results back to Postgres. Teams will also create their databases to ensure that each service only has access to the appropriate database.

Same as Jupyter Notebook, Postgres is set up on a container using the official Docker image [16]. To access Postgres services, we forward the hostname and port of Postgres, which can be accessed by other Jupyter Notebook containers within the same cluster.

4.4 Ceph

Ceph is an open-source software-defined storage system that acts as an object, block, and file storage in one unified system. It is a reliable and scalable architecture that decouples data from physical storage hardware. This is done by introducing software abstraction layers which then help with fault management. It provides interfaces for multiple storage types, thus eliminating the need for multiple storage solutions. Hence, Ceph's best applications are for cloud, Openstack, Kubernetes, and other microservices and container-based workloads, to manage large data storage.

For this project, we are using Ceph as our storage solution for managing code repositories of the various teams. These repositories consist of programs that run the various services that are the crux of this project, hence their storage on Ceph is crucial for the efficient running of the system on the cloud.

4.5 Camelot

Camelot provides the functionality to manage multiple isolated collections of data within a single running Camelot. There are situations where you want to keep data isolated from each other, for example, as a contractor using Camelot to conduct research for several clients. For this, Camelot offers datasets where each dataset can be thought of as a separate Camelot data environment. Each dataset, as per their documentation [43], is an "isolated collection of data" within a single running Camelot.

For our project, Camelot is being used as the main storage system for the Electronic Theses and Dissertations.

4.6 Kubernetes

In this project, we use Kubernetes (K8s) which is embedded in `cloud.cs.vt.edu` to help us manage and orchestrate Docker containers. While we offer each team several containers to build their services, K8s comes to play its role: (1) whenever there is a need, to scale one service up/down, and (2) to monitor and manage the health of the containers (abstracted as pods in K8s).

4.7 Rancher

Rancher [40] is an open-source multi-cluster orchestration platform that addresses the operational and security challenges of managing multiple Kubernetes clusters across any infrastructure. It deploys Kubernetes clusters and also unites them with central authentication and access control. In simple terms, it helps manage Kubernetes clusters and provides a useful UI for monitoring and managing.

The major features that make Rancher incredibly useful are as follows.

1. **Authentication and Authorization** - Teams need user management to ensure that administrative access is restricted to those qualified. This makes sure that developers don't perform operations outside of their scope, for example deleting a worker or modifying a cluster.

2. **Security Compliance** - Kubernetes clusters in production need to ensure that the right form of security is provided. Rancher provides a unique feature that scans the deployed clusters and runs tests for assessing if they meet the benchmarks.

4.8 GitLab and CI/CD

As discussed in Section 3.4, GitLab is an open-source code repository and collaborative software development platform which provides free open and private repositories. It provides Continous Integration and Continuous Deployment capabilities which automate the build, integration, and verification of the code.

For this project, we are using GitLab as our primary Version Control System to ensure a faster, more reliable, and more collaborative software development process.

Chapter 5

Design

5.1 System Architecture

To design the architecture, we first proceeded with defining the overall architecture and inter-component relationships, to gain a holistic understanding of the system. We then concluded defining system boundaries and using domain-driven design, separating the concerns that we will facilitate as a team.

The whole system revolves around the idea of containers. Containers are executable units of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether it be on a desktop, traditional IT, or the cloud.

Each team writes code which is then packaged into containers, tagged with the git short commit, and then deployed onto the CS Cloud with the help of CI/CD.

Figure 5.1 shows the overall system with each team's responsibilities. We define them here briefly.

1. Team-1 needs to be able to expose access and operations on data stored in persistent storage layers, namely the Camelot and Ceph filesystem [3], and a Relational DataBase Management System (RDBMS) like PostgreSQL [21], via an Application Programming Interface (API) [23].

2. Team-2 needs to be able to query multiple data sources, process the data, and expose those search results through a search engine like Elastisearch [44].

3. Team-3's goals are twofold. They first need to access the ETD data exposed by Team-1, and train Machine Learning models on this data. Secondly, they need to store this data back into the storage layer exposed by Team-1.

4. Team-4's goals are quite similar to Team-3, from our perspective. They too need to access data, train models on it, and store the results in a storage layer.

5. Common goals: Each team needs version-control [8], maintenance, and testing their code, seamlessly.

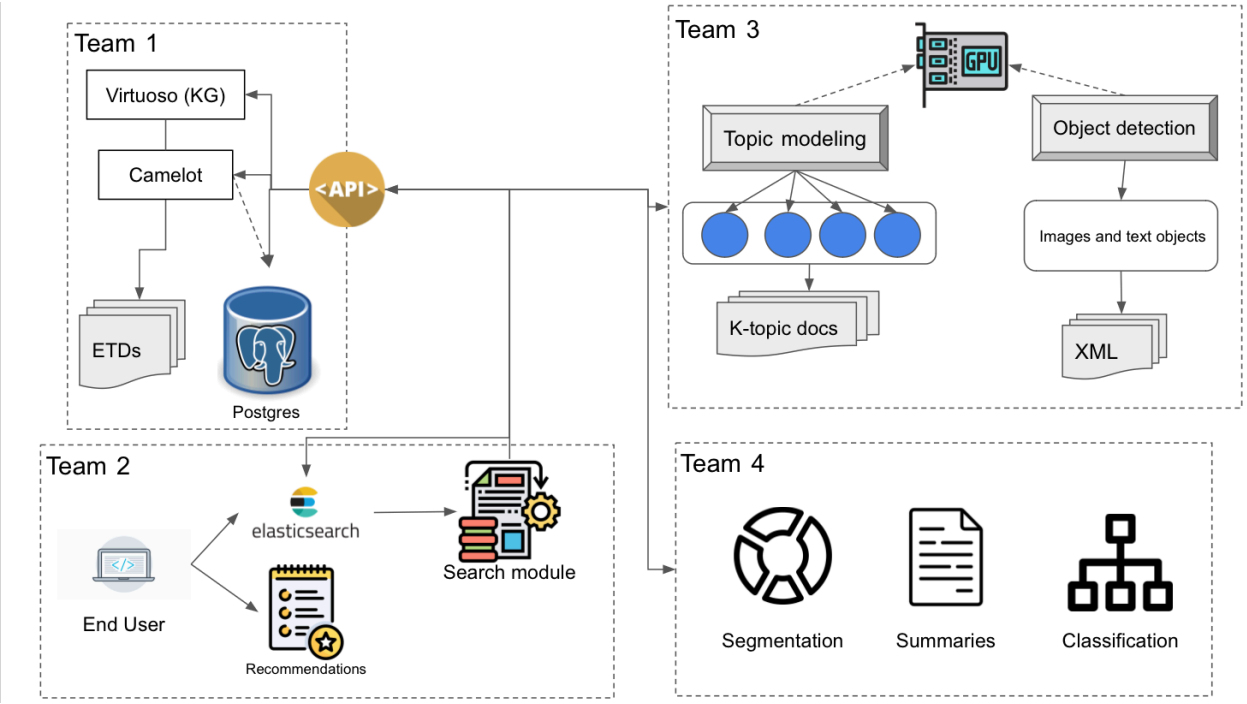

Figure 5.1: Overall system vision

After understanding the holistic view of the system and identifying the different components and their interactions, we proceeded to design an architecture that would facilitate these requirements. Figure 5.2 shows our system architecture, and identifies four different areas.

1. Continuous Integration, Continuous Delivery: The teams will be able to maintain, test, and deploy their code via GitLab [9] as the Version Control System (VCS).

2. The Camelot and Ceph filesystems: The Camelot and Ceph filesystems are where all the Electronic Theses and Dissertations (ETDs), knowledge graphs, machine learning Jupyter notebooks, PostgreSQL static files, and workflow automation temporary files will be stored. They are managed by Virginia Tech's Computer Science department's cloud infrastructure team and our team will be provided an interface to use them via static internal network IPs and credentials for passing access control.

3. Kubernetes and cloud infrastructure: We identified four different classes of containers, namely: storage (PostgreSQL), service, training, and reasoning.

   (a) storage containers: These containers are responsible for the RDBMS and other services that will eventually store data, metadata, and trained models.

   (b) service containers: These containers are responsible for serving business logic in the form of web payloads such as Javascript Object Notation (JSON) [7] or Extensible Markup Language (XML) [33] and serving front-end web pages.

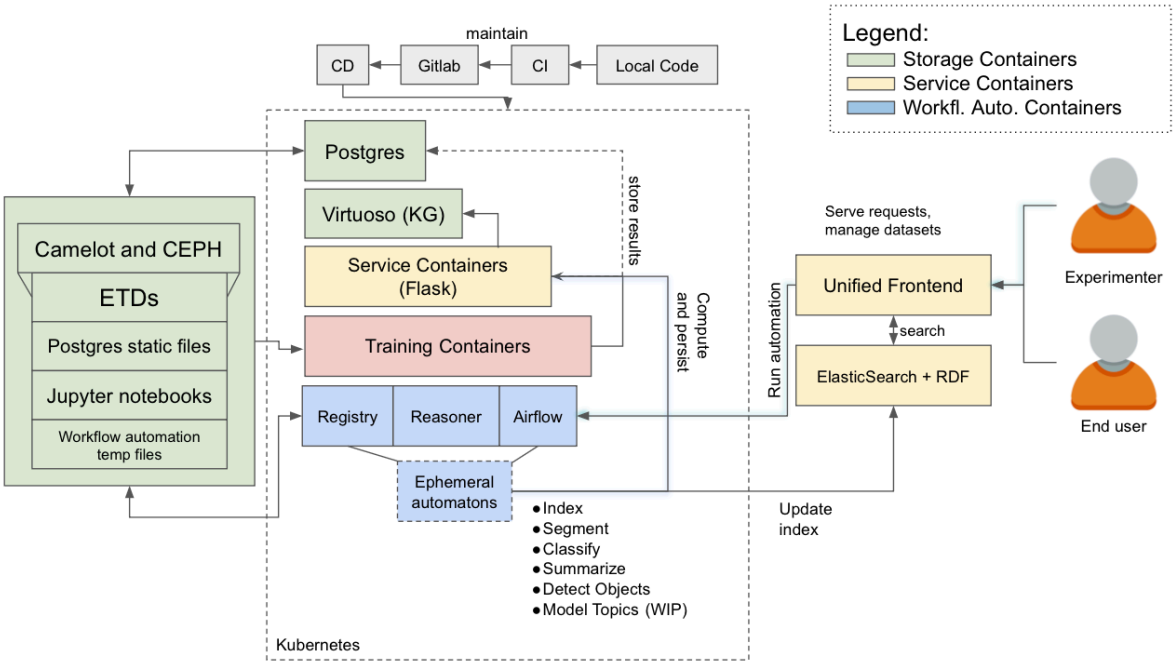

Figure 5.2: System Architecture

(c) training containers: These containers are utilized to train Machine Learning models on the ETD data.

(d) Workflow automation containers: These containers are used to serve the knowledge graph and workflow manager that will be developed by Team-5.

4. Apache Airflow: Orchestrating all these service containers will be done by Apache Airflow which provides a programmatic way to define workflows.

5.2 CI/CD Pipeline

One of the main aims of CI/CD is automating certain Developer tasks which tend to be repetitive and time-consuming. These tasks include deployment, unit testing, packaging, etc. To automate these, we can take advantage of DevOps pipelines which are built to execute code defined in an instructions file. GitLab Runner is one such tool that executes commands as defined in a .gitlab-ci.yml file.

We have identified certain cases wherein code can be containerized and deployed on the CS Cloud using Docker.

Each such use case has the following 3 files defined:

1. Dockerfile: This is the instruction set that defines how to dockerize aka containerize the code and also defines a command or script to start the container when run with no dependencies needed after containerizing it. The Dockerfile is also immensely useful for creating Developer Environments which are portable custom software environments that have all code dependencies installed.

2. .dockerignore: This file instructs Docker to ignore paths defined in it during the build process. Its usefulness comes from the fact that there are certain files/folders which are not needed to run an application and can therefore cause unnecessary bloat in Docker images.

3. .GitLab-ci.yml: The GitLab CI/CD YAML file is the instruction file which in simple terms, instructs the GitLab Runner what to do with the code. Because it takes bash scripts as inputs, the Runner can do anything with the code. For logical division, jobs can be divided into stages such as build, test, and deploy with inter-job dependencies, artifact publishing, and much more.

The Runner uses the bitnami/kubectl image for running kubectl commands which can update images after they have been pushed to the code.vt.edu Container Registry. Due to timing constraints during development, some images are in Docker Hub while the rest are in code.vt.edu Container Registry. For an uncomplicated design, it is recommended for all images to be in code.vt.edu Container Registry.

These images are tagged with the Git Short Commit and then updated in Cloud CS.

5.3 Workflow Automation Pipeline

How does one efficiently run workflows without much infrastructure cost? A Workflow Manager is a solution to this problem and Apache Airflow is one such Workflow Manager. Airflow is based upon the concept of DAGs or Directed Acyclic Graphs, which means tasks that do not have circular dependencies. Each DAG consists of multiple tasks which are defined using Operators. These Operators perform some operations on data and then return results. Let us take the example of PDF chapter summarization. When someone tries to upload a PDF and selects Chapter Summarization, only the appropriate service which is built to handle Summarization should be triggered and others should not. The advantage of Airflow is that these workflows are programmatic and defined using code written in Python. For our current model, we have defined Goals, Workflows, and Services. Following are their definitions:

- Goal: A Goal is an overall output that one wishes to achieve. Goals can include text summarization, segmentation, etc.

- Workflow: A Workflow describes what needs to be done to achieve a particular goal. It is important to note that each Workflow has exactly one goal but a goal can have many different workflows.

- Services: Lastly we have services which are the atomic tasks needed to complete a workflow. A workflow can thus have many different services, especially if it is complex. It can also have different combinations and linear/nonlinear services.

5.3.1 Overall Workflow Design

The overall workflow design has been shown in Figure 5.3. We used the Python script from the 2020 INT team to launch the workflow automation service. Developers can specify goals, and services, using the Flask UI which will then be populated into the database. Then they have to re-generate the generated context-free-grammar file referencing the workflow tables (details of the tables structure can be found in the next subsection). Then, end users can call the *generate_workflow* API to generate DAGs. They can also pass in variables such as *ETD_ID* as environment variables for each service. After that, they can trigger the DAGs through the Flask UI.

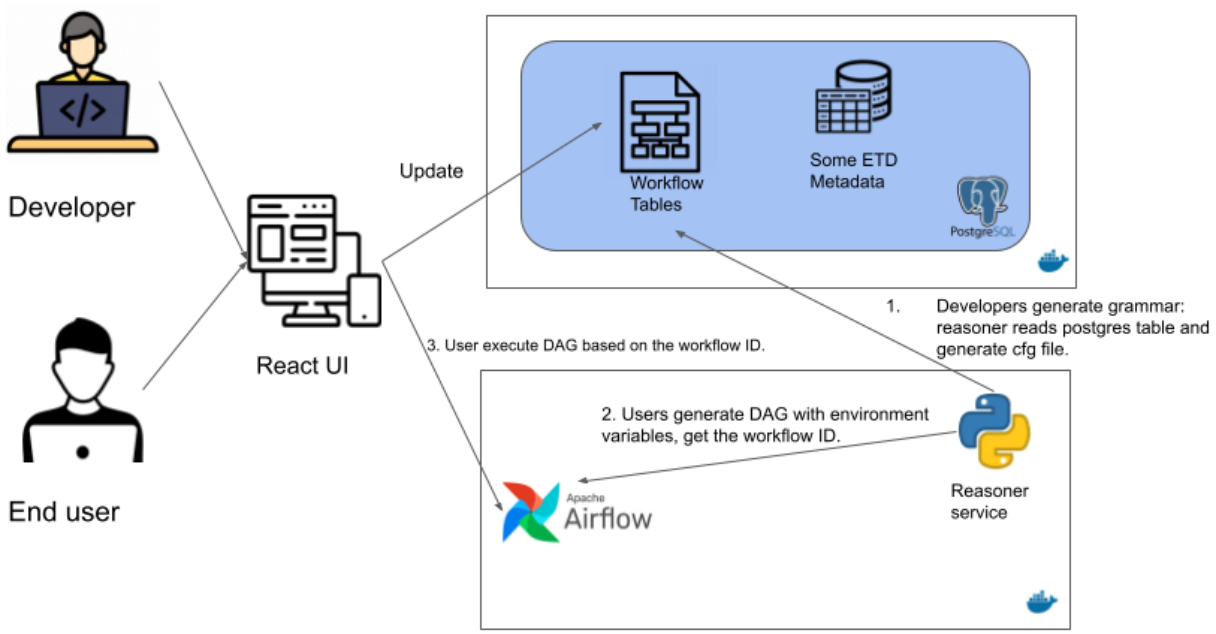

Figure 5.3: Workflow Automation Design

5.3.2 Workflow Database Design

Besides other metadata tables required for ETDs, we mainly have four tables that together provide the data for workflow functionality. We have designed our current schema keeping in mind the relationships between goals, workflows, and services. The table descriptions are as follows. The UML diagram for the same is shown in Figure 5.4.

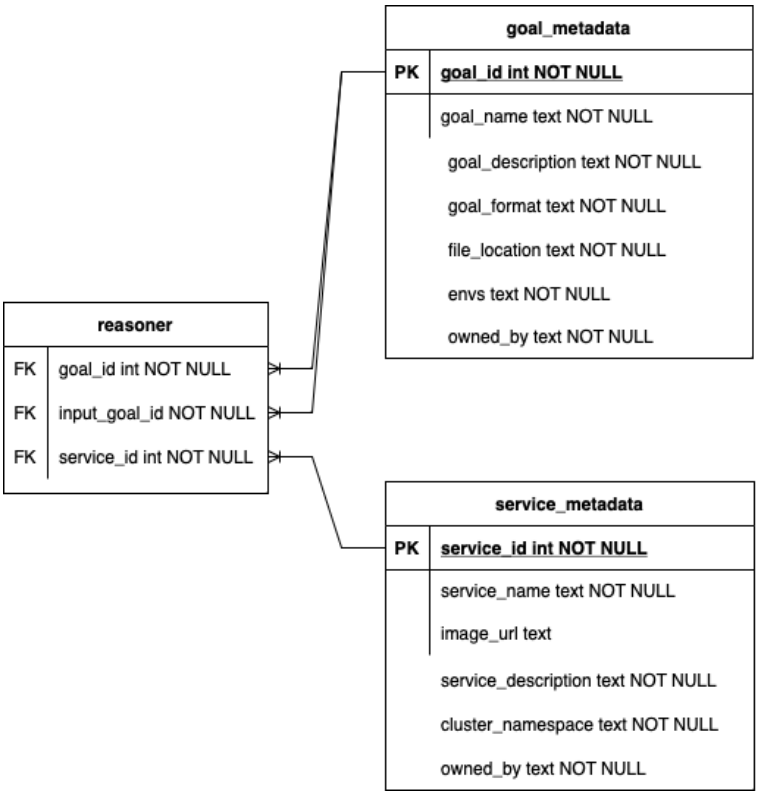

Figure 5.4: ER diagram for Workflow Automation

Goal Metadata Table

This table stores goal-related information that will be used as inputs and outputs to the services metadata. It also includes the environment variables each goal requires. The schema contains the following columns:

1. goal_id: (Integer) This primary key of the table uniquely identifies a goal.
2. goal_name: (String) A human-readable string that represents each goal's functionality.
3. goal_description: (String) A brief description of the goal, that helps users understand its functionality.

4. goal_format: (String) Represents the format of the output represented by the goal.
5. file_location: (String) Represents the directory/filename associated with the goal.
6. envs: (String) Some environment variables that can be used inside the container.
7. owned_by: (String) The owner of the goal.

Service Metadata Table

The Service Metadata table describes all services required by workflows. Each service picks up its input from a file location and deposits the output of its operations at another file location. This is how we achieve communication between services in a workflow. Note that a service might be an API defined by team 1 or any of the general services provided by teams 3 or 4. The schema contains the following columns:

1. service_id: (Integer) This primary key of the table uniquely identifies a service.
2. service_name: (String) A human-readable string that represents each service's functionality.
3. service_description: (String) A brief description of the service, that helps users understand its functionality better.
4. image_url: (String) Stores the Docker image URL on Docker Hub[1], for example, "lyuze/postgres:latest".
5. cluster_namespace: (String) Represents the cluster namespace, e.g., etd.
6. owned_by (String): (String) The owner of the service.

Reasoner Table

The reasoner table contains the task order for the workflow. The reasoner service will reference this table to generate the context-free-grammar (cfg) file.

1. goal_id: (Integer) The foreign key referenced from the primary key of the Goal Metadata table. This column indicates the Output Goal of a particular service.
2. input_goal_id: (Integer) The foreign key referenced from the primary key of the Goal Metadata table. Certain services need Input Data/files, etc. That is indicated by this column.

[1]While ideally all images should be in code.vt.edu Container Registry, due to Time constraints, some Docker Images are on Docker Hub while the Rest are in code.vt.edu Container Registry.

3. service_id: (Integer) The foreign key referenced from the primary key of the Service Metadata table.

Chapter 6

Implementation

6.1 Workflow Automation

In this section, we first describe how we used the 2020 INT team's code to generate the context-free-grammar file to represent DAG. Then we showcase all related reasoner APIs for workflow automation. After that, we show how the UI interacts with the reasoner APIs. Finally, we will give a detailed demo of how we automate team-4's services.

6.1.1 Context-Free Grammar

A Context-Free Grammar (CFG) is a set of recursive rules used to generate patterns of strings. A context-free grammar in general can be described by four components V, Σ, R, and S, where V is a set of non-terminal symbols, Σ is a set of terminal symbols, R is a set of production rules that defines a mapping from a non-terminal symbol to a string $s \in (V, E)$, and $S \in V$ is a start symbol. Context-Free Grammars find application in areas such as theoretical computer science, compiler design, and linguistics [29].

In our approach, which is built on the reasoner implementation of Team INT-2020, a Context-Free Grammar is used as a way to represent goals and services as components of a workflow. A goal identifies a result state, i.e., some stored data, that comes from running a service. The stored data herein is referred to as 'input goal' since it can be an input to another service. Each service has a set of input goals and a single (result) goal. Input goals are starting directories or files that a service needs to use and work upon. Once completed, the output for such a service is a (result) goal which can either be a file or a directory.

The Python "nltk" package provides a "grammar" module which is used for generating a CFG from a registry of services. The advantage of generating a CFG is that it can help generate multiple possible workflows that can help to achieve a particular end goal. Although in our current approach each goal is mapped to a single workflow, in the future the implementation can be extended for cases where multiple workflows are possible for a given goal.

How are workflows generated using a Context-Free Grammar? Let G be a Goal, s be a service, and G_I be an Input Goal. Then, a production rule can be represented as $G \rightarrow sG_I$. We intentionally discard the start symbol as it has no use while generating workflows. For a general set of input goals, a generic production rule can be represented

as $G \rightarrow sG_{I1}G_{I2}G_{I2}..G_{In}$. With the help of the reasoner table, the production rules can be generated for the entire table. Let's take an example:

Goal ID	Service ID	Input Goal ID
1	Null	Null
2	S_1	1
3	S_2	2

Before we attempt to construct the rules, let's give a brief explanation of the table. The tables state that:

- 1 is the start Goal (Service ID is Null).
- Goal 1's data is the input for Service S_1 and the output Goal is 2. Similarly for Service S_2.

This workflow, as shown, is linear, i.e., Service 3 executes after Service 2 which executes after Service 1. If by the above generic format we generate a production rule using the values, we obtain the following (S is the start state):

$$S \rightarrow G_1$$
$$G_3 \rightarrow S_2G_2$$
$$G_2 \rightarrow S_1G_1$$
$$G_1 \rightarrow \epsilon$$

If we now construct the string using these rules and consider G_3 as a final goal, then we obtain the following string,

$$G_3 \rightarrow S_2S_1\epsilon$$

This string means that to achieve Goal G_3, Service S_1 and then S_2 need to be executed. If we now give Goal G_3 as the Goal ID to the generate goal API, then we will obtain a workflow with services S_1 and S_2.

Currently, the reasoner table, which helps generate these rules, is manually populated as per requirements from other teams. The following are the rules:

```
71 -> '37' 115
112 -> '38' 71
115 ->
```

```
89 -> '48' 73
73 -> '47' 114
119 ->     117
116 ->     121
121 -> '60' 120
114 -> '39' 112
90 -> '49' 89
120 -> '59' 90
117 -> '62' 115
124 -> '62' 121
122 -> '63' 124
123 ->      122
```

6.1.2 Reasoner API

The reasoner API consists of four consecutive APIs, which are generate_grammar_API, generate_workflow_API, run_workflow_API, and get_workflow_status_API. We explain each of them in detail here.

Generate Grammar API

This API sends a GET request to the reasoner server that queries the current reasoner table. Then it updates the cfg file. Required input: None. Output is shown in Figure 6.1.

```
1 {
2     "savedTo": "reasoner.cfg"
3 }
```

Figure 6.1: Example Output of Generate Grammar

Generate Workflow API

This API sends a POST request to the server, which reads the updated cfg file and generates the workflow DAG. Required input: *workflowId* (which is also the end goal_id of the workflow to execute, for example, the goal_id of *End* in Figure 3.1). Optional input: *Env* for specified services within the workflow. We show an example in which all services need *ETDID=7* set as one of the Environment Variables. See Figure 6.2 for the Code.

```
1  {
2      "env":[
3          {
4              "service_id":"all",
5              "service_env":[
6                  "ETDID=7"
7              ]
8          }
9      ]
10 }
```

Figure 6.2: Example of Optional Environment variable passed to Generate Workflow

Output: a JSON object that contains the requested goal ID, workflow ID, service metadata, goal metadata, and workflow metadata. The example output is shown in Figure 6.3.

Run Workflow API

Once the DAG is generated, this API can be called along with the workflow ID from the previous API, to execute the DAG. Required input: a JSON with *args* and *service_names*. The first element in *args* should be the workflow ID. The second is the *execution_date*. If the *execution_date* is before the current date, the DAG is triggered immediately. *service_names* is optional; all tasks will be executed if this field is left blank. An example input is in Figure 6.4 Output: a *key* of current execution, a *result_url* that can be used to trace logs, and *status* will show as *running* (Figure 6.5).

Get Workflow Status API

This API can be used to check the status of an execution. Required input: the *key* produced by the previous API. Output: a JSON object that shows the information of the execution, such as error message, start time, end time, logs, and return code.

API order

The order to run each is as follows:

1. Generate Workflow API
2. Run Workflow API

```
1  {
2      "requestedGoal": "71",
3      "workflow": [
4          [
5              37
6          ]
7      ],
8      "workflowId": [
9          "VebHoUpb"
10     ],
11     "serviceMetadata": [
12         [
13             {
14                 "service_id": 37,
15                 "service_name": "curl-get-etd",
16                 "service_description": "Fetches PDF given URL",
17                 "image_url": "ano2202/cs5604-curl-get-etd-pdf:0.2.13-linux",
18                 "cluster_namespace": "etd",
19                 "owned_by": "INT"
20             }
21         ]
22     ],
23     "goalMetadata": [
24         [
25             {
26                 "goal_id": 115,
27                 "goal_name": "start",
28                 "goal_description": "start",
29                 "goal_format": "<Directory>",
30                 "file_location": "/mnt/data/team5/segmentation_input",
31                 "environment_variable": "START",
32                 "owned_by": "INT"
33             },
34             {
35                 "goal_id": 71,
36                 "goal_name": "fetch_etd_output",
37                 "goal_description": "Stores fetched ETD for segmentation, summ, class, etc.",
38                 "goal_format": "<Directory>",
39                 "file_location": "/mnt/data/team5/segmentation_input",
40                 "environment_variable": "ORIGINAL_DATASET",
41                 "owned_by": "INT"
42             }
43         ]
44     ],
45     "workflowMetadata": [
46         [
47             {
48                 "goal_id": 71,
49                 "service_id": 37,
50                 "input_goal_id": 115
51             }
52         ]
53     ]
54 }
```


Figure 6.3: Example Output of Generate Workflow

```
{
    "args":["VebHoUpb","2020-10-10"],
    "service_names":[]
}
```

Figure 6.4: Example of Request Body of Run Workflow

```
{
    "key": "07bc268b",
    "result_url": "http://team2020-airflow.discovery.cs.vt.edu/workflow/run?key=07bc268b",
    "status": "running"
}
```

Figure 6.5: Example Output of Run Workflow

3. Get Workflow Status API

Note that the Generate Grammar API needs to be called only when services or goals are added/deleted or the reasoner table is modified in any way.

6.1.3 Interface

To allow for CRUD (Create, Read, Update and Delete) operations to be done on the registry by various team members for their respective services, we developed a frontend consisting of services, goals, and workflows.

This page can be accessed from the navigation menu of the main frontend website given in Figure 6.6.

Goals, Services, and Reasoner details and tables are described with their figures. Figure 6.7 displays the Goals table, Figure 6.8 displays a screenshot of the services table, and Figure 6.9 displays a screenshot of the Reasoner table. The table metadata has been defined in the following section based on the schema of the database that helps populate these pages.

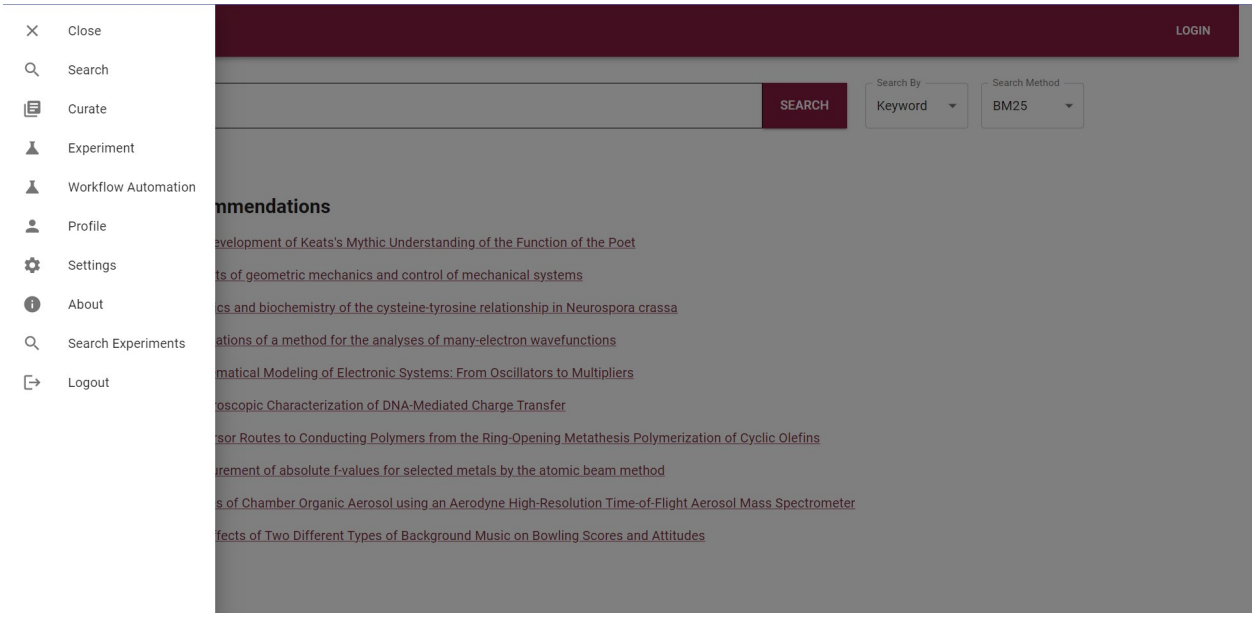

Figure 6.6: Navigation to Workflow Automation Page

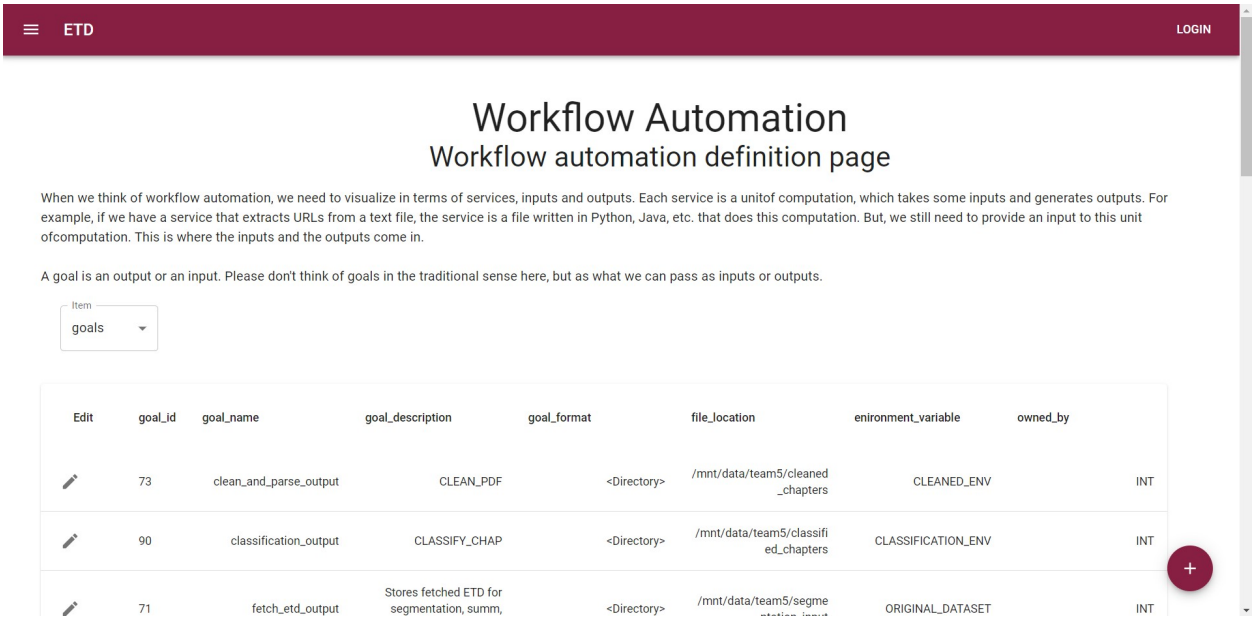

Figure 6.7: Goals Table

ETD LOGIN

Workflow Automation

Workflow automation definition page

When we think of workflow automation, we need to visualize in terms of services, inputs and outputs. Each service is a unitof computation, which takes some inputs and generates outputs. For example, if we have a service that extracts URLs from a text file, the service is a file written in Python, Java, etc. that does this computation. But, we still need to provide an input to this unit ofcomputation. This is where the inputs and the outputs come in.

A service is a unit of computation. We represent it here as a Docker image that is run in a container and performs a unit of computation.

Item
services

Edit	service_id	service_name	service_description	image_url	cluster_namespace	owned_by
	47	clean-chapters	Converts Chapter PDF in TXT files	outerspace1920/clean_pdf:1.5	etd	INT
	49	classification	classification chapters	outerspace1920/classification:1.0	etd	INT
	39	curl-save-chapters	Saves PDF in Remote Location	lyuze/upload_seg:0.3	etd	INT
	1	service1	Discards negative numbers from a dataset	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/service1:latest	etd	INT

Figure 6.8: Services Table

ETD LOGIN

Workflow Automation

Workflow automation definition page

When we think of workflow automation, we need to visualize in terms of services, inputs and outputs. Each service is a unitof computation, which takes some inputs and generates outputs. For example, if we have a service that extracts URLs from a text file, the service is a file written in Python, Java, etc. that does this computation. But, we still need to provide an input to this unit ofcomputation. This is where the inputs and the outputs come in.

The reasoner is a mapping between inputs, outputs and services.

Item
reasoner

Edit	goal_id	service_id	input_goal_id	created_at
	71	37	115	Tue, 29 Nov 2022 17:47:15 GMT
	112	38	71	Tue, 29 Nov 2022 17:47:15 GMT
	114	39	112	Tue, 29 Nov 2022 17:47:15 GMT
	115			Tue, 29 Nov 2022 17:47:15 GMT
	73	47	114	Tue, 29 Nov 2022 17:47:15 GMT

Figure 6.9: Reasoner Table

For each of these tables, a team member can add or edit data. For example, Figure 6.10 shows the pop-up menu that allows a user to fill in the required data for adding a new goal. Submitting this data populates the database with the provided Goal details and displays them on the Goals table accordingly.

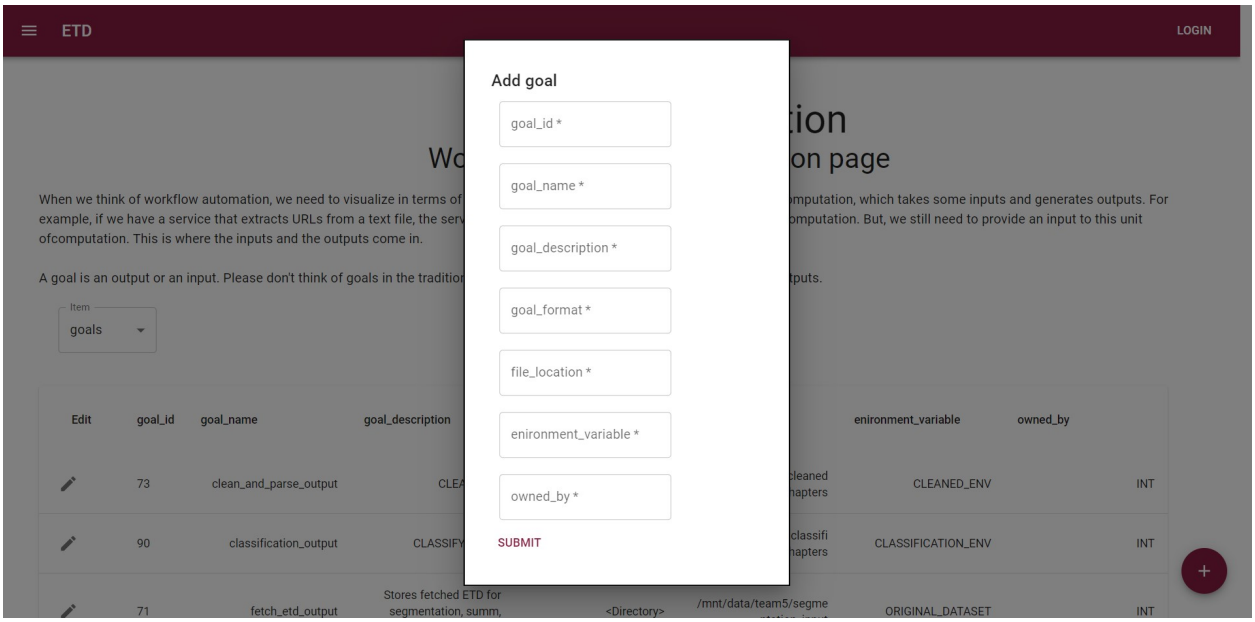

Figure 6.10: Addition of a new Goal

Populating the registry with goals, services, and reasoner metadata

Ideally, the teams were supposed to use the registry UI to populate the data related to workflow automation. However, while the UI was being developed in parallel with the backend and other parts of workflow automation, we populated the database manually. Henceforth, since the UI is ready, any experimenter or team in the future will be able to populate the database with the registry UI, and manual data entry to the database will not be required.

6.1.4 Facilitating easy adoption of the workflow automation system into frontend application

With the interface, database, and database APIs in place, our next steps involved connecting these to allow users to be able to generate and run workflows directly from the user interface. This was accomplished by creating a code wrapper written in JavaScript, which consists of functions that make the required API calls. Thus, the developers who wish to integrate this system into their frontend applications need only to import these functions and call them.

The three functions defined in this wrapper include the following which perform the API functionalities mentioned in Section 6.1.2:

1. generateWorkflow()
2. runWorkflow()
3. getWorkflowStatus()

The code for this wrapper and its functions can be found at https://code.vt.edu/aaron2000/cs5604-front-end/-/blob/main/src/api/AirflowReasoner.js

6.1.5 Registry

The registry stores the workflow automation metadata which is referenced by the reasoner as well as Apache Airflow to construct and run workflow automations. For this purpose, we need to store the following information:

1. Services
2. Inputs and outputs (also called the Goal metadata)
3. How the various services are connected via inputs and outputs (also called the reasoner metadata)

We represent this schema in the form of three PostgreSQL database tables, whose Data Definition Language (DDL) is given below:

1. Goal Metadata

```sql
CREATE TABLE goal_metadata (
    goal_id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    goal_name text NOT NULL,
    goal_description text NOT NULL,
    goal_format text NOT NULL,
    file_location text NOT NULL,
    environment_variable text NOT NULL,
    owned_by text NOT NULL
);

-- Indices ------------------------------------------------------------

CREATE UNIQUE INDEX goal_metadata_pkey ON goal_metadata(goal_id int4_ops)
    ;
```

2. Service Metadata

```
CREATE TABLE service_metadata (
    service_id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    service_name text NOT NULL,
    service_description text NOT NULL,
    image_url text NOT NULL,
    cluster_namespace text DEFAULT captionpos'captionposcs5604captionpos-
        captionposintcaptionpos-captionpostestcaptionpos'::text,
    owned_by text NOT NULL
);

-- Indices ------------------------------------------------------------

CREATE UNIQUE INDEX service_metadata_pkey ON service_metadata(service_id
    int4_ops);
```

3. Reasoner

```
CREATE TABLE reasoner (
    goal_id integer REFERENCES goal_metadata(goal_id) ON DELETE CASCADE,
    service_id integer REFERENCES service_metadata(service_id) ON DELETE
        CASCADE,
    input_goal_id integer REFERENCES goal_metadata(goal_id) ON DELETE
        CASCADE
);
```

This Postgres database is hosted on the CS Cloud Infrastructure using Team 2020 INT's Docker image: container.cs.vt.edu/cs−5604−fall−2020/int/team−int−repo/postgres:latest at the URL: team2020db.discovery.cs.vt.edu

6.1.6 Registry Backend

We have created a backend server responsible for exposing an interface to the registry, which stores the Goals, Workflows, and Services, i.e., the Knowledge Graph around workflow automation as described in Figure 5.4. We have achieved this by creating a server that runs Python's Flask framework as a runtime and persists data into the PostgreSQL instance described in Section 6.1.5. The registry is exposed to the end-user in the form of a REST API containing CRUD (Create, Read, Update, Delete) operations on the Goals, Workflows, and Services. We used Postman [38] as an API platform to design our API. This server can be accessed at https://registry-backend.discovery.cs.vt.edu/ and the collection of REST API endpoints that can be used to perform operations on the registry can be found in the Postman collection at https://documenter.getpostman.com/view/4087380/2s8YzL2kT9.

The registry backend runs the following Docker image: ano2202/registry−backend:0.0.6−linux and is based on the work done by Team 2020 INT. The codebase for the same can be found at https://code.vt.edu/aaron2000/cs5604-f22-team-5-repo-2.

6.1.7 Indexing - Workflow Automation

We show the indexing workflow in Figure 6.11. The indexing workflow performs indexing of the existing and newly added ETDs, and updates the index accordingly. The first service of this workflow involves fetching the ETDs and the metadata from the database of Team-1. The second service performs a transformation of the fetched data to a format suitable for the API of Team-2. In the final service, a call is made to the API developed by Team-2 for indexing the ETDs along with updating the existing index.

The following Docker image was created to facilitate this workflow:

- ano2202/cs5604-team-2-update-index:0.8

Figure 6.11: Indexing workflow

6.1.8 Object Detection - Workflow Automation

The object detection workflow shown in Figure 6.12 detects objects in the input PDFs and stores the results as XML files. The ETDs are fetched from Team-1's database, and the objects in the PDFs are then detected by the service that hosts the Python script developed by Team-3 for object detection. The results of the detection, and other metadata related to each object, are stored in the form of XML files on the Ceph storage system under the path "/mnt/data/team3".

The following Docker images were created to facilitate this workflow:

- ano2202/cs5604-curl-get-etd-pdf:0.2.9-linux
- ano2202/cs5604-team3-processing-suite:0.5

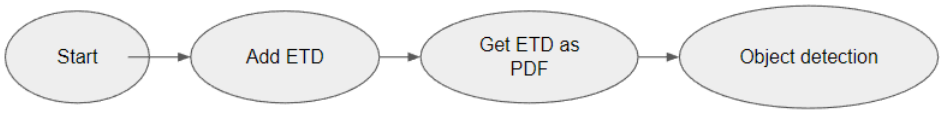

Figure 6.12: Object detection workflow

6.1.9 Segmentation, Summarization and Classification - Workflow Automation

The segmentation, summarization, and classification of ETD chapters are combined into a single workflow, shown in Figure 6.13. The segmentation service segments the input PDFs into individual chapters. Each segmented chapter from the input PDF is then stored as a separate PDF in the file system. The clean and parse service extracts and cleans the textual content from the separated chapter PDFs which are then saved in the form of text files. In the summarization step, the extracted content is shortened and summarized and stored as JSON files. Finally, in the classification stage, a multi-label classification is performed on each chapter to identify the two fields to which they are most related. The outputs generated at the end of the segmentation, clean and parse, summarization, and classification steps are all uploaded to Team-1's database by making a call to their API.

The following Docker images were created to facilitate this workflow:

- ano2202/cs5604-curl-get-etd-pdf:0.2.13-linux
- ano2202/cs5604-segmentation:0.4
- lyuze/upload_seg:0.7
- outerspace1920/clean_pdf:1.5
- outerspace1920/summarization:1.0
- outerspace1920/classification:1.1
- lyuze/upload_summ:0.2
- lyuze/upload_class:0.1
- outerspace1920/upload_cleaned_chapters:1.1
- lyuze/update_index:0.1

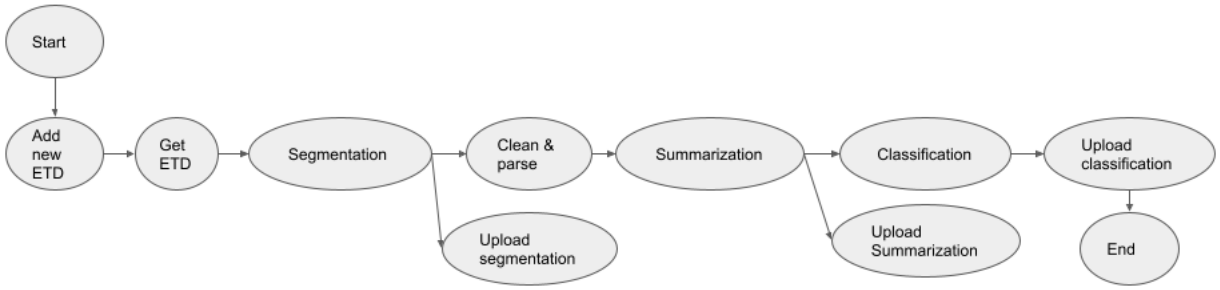

Figure 6.13: Segmentation, Summarization and Classification Workflow

6.1.10 Alternate Workflow Automation

The workflows discussed above, and their development, can be generated in other ways supported by our system. As is explained in Section 6.1.4, the operations of other teams w.r.t. workflows involves their use of calls that include runWorkflow() and getWorkflowStatus(). If, to achieve a goal, by running a workflow, other teams can cause Airflow to run a different yet suitable entry in the /dags folder, all should proceed properly.

One way to produce suitable entries in the /dags folder is for a developer to edit an existing entry in the /dags folder. This allows a developer to take advantage of other features, like parallel operation, that Airflow supports. The resulting benefit would then be achieved if the edited file can be given to Airflow so it can run the workflow that the edited entry describes.

A second way to produce suitable entries in the /dags folder is for a developer to manually create and upload a suitable entry in the /dags folder for a workflow. They would not use the Generate Grammar API or the Generate Workflow API, since their hand-coding of the entry in the /dags folder for a workflow would just be a programming exercise aimed to create something appropriate that Airflow supports. In this situation, the so-called Reasoner would not be employed, since the workflows would be hand-coded. Further, there would be no need for reasoner entries as discussed in Figure 7.10, and no need for the Reasoner table described in Section 6.1.5.

Each of the above two variations could lead to system operation that bypasses the Reasoner. Each would result in having suitable entries in the /dags folder. The rest of the system could make use of a new API call, e.g., runDAGentry(), that would be given an identifier for a suitable entry in the /dags folder. This API would be similar to the existing runWorkflow() API. When a part of the overall system wanted to address some goal, instead of calling generateWorkflow() and runWorkflow(), it would call runDAGentry(), with argument being the /dags folder entry created as above.

6.1.11 Sample workflow Demonstration

A video demonstration for this workflow was created to be shown as part of the final presentation. It can be found as the video file named 'Team5INTdemo.mp4' and shows this workflow automation being executed in its entirety. A tabular description of the demonstration video follows as Table 6.1 that contains timestamps in the video as one column and the actions or changes occurring at the corresponding timestamp as the other column. This demonstration combines the "Segmentation, Summarization and Classification" and the "Indexing" workflows as a single, chained workflow automation.

Timestamp	Description
00:01	Navigating to the API request that generates a new workflow instance. The REST API client UI used here is 'Postman'.
00:08	API request to generate a new workflow instance of the "Segmentation, Summarization and Classification" workflow automation is sent.
00:25	API request to run the newly created workflow instance is sent (This will trigger Airflow and set the automation in motion).
00:30	API request to check the status of the running workflow instance is sent.
00:38	The first step of the workflow automation ('Get ETD') begins getting executed in the form of a new container that is created on Rancher.
00:45	The ETD that was to be processed has been fetched via the 'Get ETD' step and has been persisted in the file system.
00:51	The 'Segmentation' step begins execution in the form of a container being created on Rancher.
07:08	The 'Segmentation' step has finished running, which has persisted the segmented chapters in the file system.
07:14	The segmented chapters are seen in the file system. There are three chapters that were generated by the automation for this ETD PDF. The segmented chapters have also been uploaded to Team-1's database.
07:18	The 'Clean and Parse' step of the workflow begins.
07:39	The 'Clean and Parse' step of the workflow finishes executing.
07:46	The 'Summarization' step of the automation begins.
08:15	The 'Summarization' step of the automation finishes executing.
08:18	The 'Classification' step of the automation begins.
09:13	The 'Classification' step of the automation finishes executing.
09:14	The 'Upload Summarization' step of the automation begins.
09:17	The 'Upload Summarization' step of the automation finishes executing.
09:20	The 'Upload Classification' step of the automation begins.
09:23	The 'Upload Classification' step of the automation finishes executing.
09:24	The 'Upload cleaned chapters' step of the automation begins.
09:28	The 'Upload Cleaned Chapters' step of the automation finishes executing.
09:30	The 'Fetch Metadata and Update Index' step of the automation begins.
09:43	The 'Fetch Metadata and Update Index' step of the automation finishes executing.
09:46	API request to check the status of the running workflow instance is sent again.
09:48	The API request is successful and shows the result of the run. The generated report contains the Docker container logs for all the steps that were run as part of the automation.
10:42	The artifacts created as part of the run are now displayed. These include, the segmented chapters, the parsed and cleaned chapters, classified chapters and the summarized chapters,

Table 6.1: Explanation of the demonstration video: Team5INTdemo.mp4

6.2 Developer Operations

6.2.1 Continuous Integration and Continuous Deployment

Pipelines in CI/CDs are nothing but a piece of hardware and software which executes the supplied code. We are using GitLab as our main pipeline runner. GitLab's Runner is open-source software that can build, test, deploy, etc. and it is fully featured.

To set up CI/CD for each team, it is a prerequisite that we have a Virtual Machine running the GitLab Runner with Docker as its executor. The Developer Manual in Section 8.4 has a guide to setting up the GitLab Runner on a Virtual Machine.

Since the GitLab Runner needs access to each project, individual project tokens need to be registered.

Once registered, each team needs to set up a .gitlab-ci.yml file which contains the code to run the pipeline.

Following is a very simple .gitlab-ci.yml file that builds and pushes a Docker image to the Docker hub.

```
stages:
  - build
build-job:
  stage: build
  image: docker:latest
  script:
    - docker build -t simplenode .
    - echo "Build complete"
    - echo "Deploying application..."
    - docker push simplenode
```

This yaml code assumes a simple Node.js application with the following Dockerfile:

```
FROM node:14-alpine
WORKDIR /app
COPY package*.json /app/
ENV NODE_ENV=development
RUN npm i
COPY . /app/
CMD npm run start
```

Since yaml is infinitely scriptable, exceedingly complex pipelines can be created.

6.2.2 CI/CD Pipeline setup

To deploy to the cloud, manually setting the tag for each deployment is a tedious process. To automate this and to inculcate automated testing, CI/CD pipelines are being set up for each team, taking into account their needs, and thus tailoring each container image.

Each Team Repository is set up to use the GitLab Runner which will run scripts as defined in a .gitlab-ci.yml

The Runner is hosted at containers.cs.vt.edu and uses the Docker executor. The VM behind containers.cs.vt.edu is managed by the CS Department but to host the GitLab Runner, any Virtual Machine can be used.

The images that are built as a result of a Dockerfile, will be stored in GitLab's Container Registry.

Figure 6.14 shows the runner listed in the frontend team repository.

Runner #1236 specific

Property Name	Value
Active	Yes
Protected	No
Can run untagged jobs	Yes
Locked to this project	No
Tags	docker frontend
Name	gitlab-runner
Version	15.4.0
IP Address	128.173.237.163
Revision	43b2dc3d
Platform	linux
Architecture	amd64
Description	gitlab runner for front end team
Maximum job timeout	
Last contact	23 minutes ago

Figure 6.14: Frontend Team's GitLab Runner

The following details each Team's container requirements:

- Team 1 - uses Flask as their Python Server Framework wherein all APIs have been defined as well as their processing functions. A standalone PostgreSQL and Virtuoso container is set up which is the main repository of the class.

- Team 2 - They have 2 containers, i.e., Recommendation System and the main search server.

- Team 3 - Team 3 uses an XML based preprocessing technique to display different objects of an ETD. This server also uses Flask as the framework. Topic Modelling also has its code containerized and deployed to Rancher via GitLab CI/CD pipelines.

- Team 4 - Team 4 doesn't need a Flask server but their UI is integrated in the main frontend repository.

- Lastly, the frontend team has a separate container that uses ReactJS as the Frontend Framework. The Dockerfile has all steps to create a production build of the App and to use Nginx as the Web Server.

Each of the 4 teams and the frontend team which has used Flask and ReactJS have their containers with associated code bases connected to a CI/CD pipeline.

6.2.3 Troubleshooting for Other Teams

- Helped to set-up GPU backed containers. Initially, Teams 3 & 4 faced the issue with connecting to GPUs and were unable to run nvidia-smi on their containers.

- Helped people to troubleshoot whenever their Containers were inaccessible from the public internet.

- Demonstrated how to build a Dockerfile and enabled people to create their own custom Docker images as per their requirements.

- Supported Team 1 in setting up their Flask app and Jupyter Dev container.

6.2.4 Containerization

We have successfully spun up Kubernetes Deployments that contain Pods, which in turn run Docker containers based on the requirements specified by various teams. The containers are all running within the 'Discovery' Kubernetes cluster maintained by the CS Cloud Infrastructure. These pods are all healthy and are being used by the teams for their purposes. The steps followed for spinning up all of the containers mentioned here, involve using the user interface served by Rancher, as mentioned in Section 8.1.

The following is a comprehensive list of all the running containers, along with their descriptions. See 6.15 for a Visual Representation.

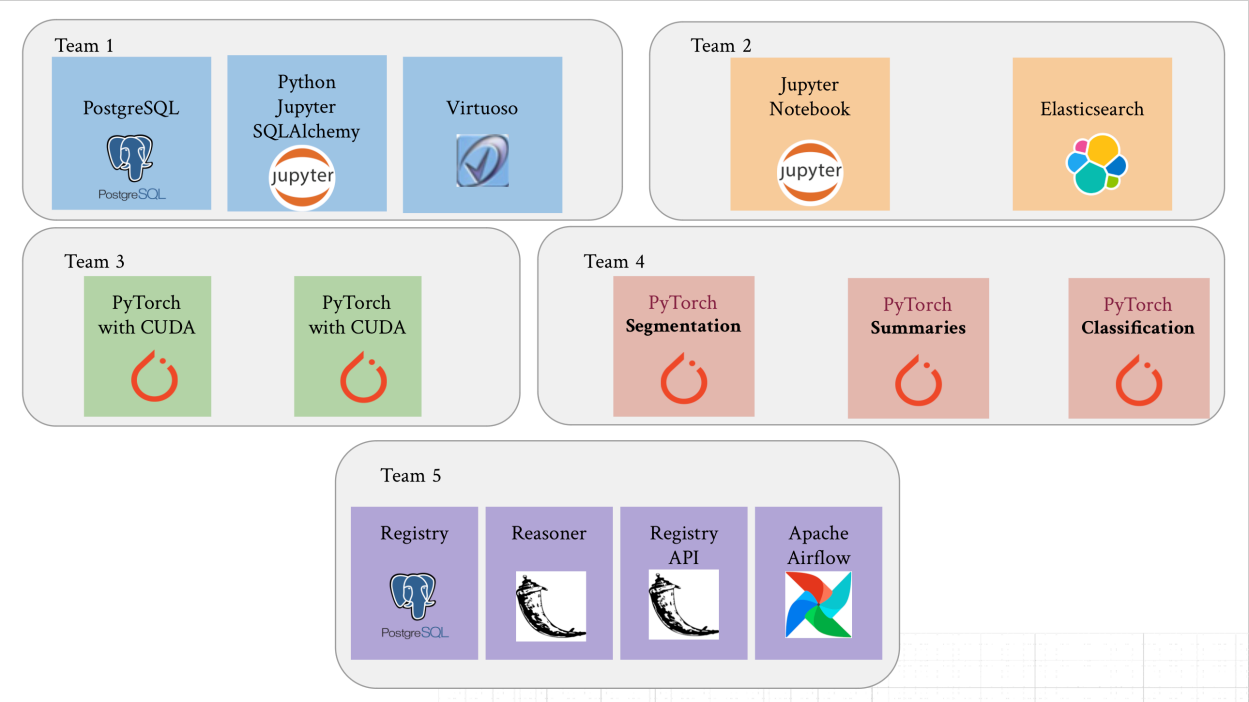

Figure 6.15: Containerization overview

1. Team 1 - Team 1's focus is on data curation. Hence their first and foremost container would be a database container, i.e., PostgreSQL. Additionally, for computing, setting up APIs, etc. we have also installed a base Python image with other packages installed as required. Additionally, a Virtuoso container is created to provide the functionalities of a high-performance virtual database engine that can support a knowledge graph made with RDF triples.

   (a) Container 1 - PostgreSQL

   (b) Container 2 - Base Image - Python, Jupyter, Pandas, Numpy, Matplotlib, Sqlalchemy, Flask

   (c) Container 3 - Virtuoso

2. Team 2 - Team 2 has 3 requirements. To run the recommendation system, a Jupyter notebook with GPU access is required. The backend[1] is written in Python using Flask as the framework. Finally, Elasticsearch is used as the search backend. The Python App is a custom Docker image that is hosted inside the Team 2 repository in `https://code.vt.edu/`. A pipeline will be used to build the Docker images and they

[1] In order to transform queries that can be read by Elasticsearch, Team 2 uses a backend that handle queries from the frontend and query results by Elasticsearch. Moreover, it handles Authentication/Authorization using Google Oauth.

will be pushed to the Container Registry of the same code repository. Elasticsearch runs inside a Pod on Rancher.

   (a) Container 1 - Jupyter notebook with GPU
   (b) Container 2 - Flask App
   (c) Container 3 - Elasticsearch

3. Team 3 - We spun up GPU-backed containers for Team-3, and asked them about the list of dependencies that they require for running their workflows. However, they mentioned that their Subject Matter Expert had asked them to use their cluster for training the model and generating weights. After some clarifying questions were asked, they further clarified that they will still need containers to meet any further training demands that are generated by the workflow automation that our team will work on. For this purpose, we provisioned GPU-backed Docker containers running a custom Docker image (discussed in Section 6.2.5) which included Jupyter. The descriptions of the containers and their URLs are as follows:

   (a) Container 1 - GPU-backed container running Jupyter Notebooks+Pytorch (URL: https://team3-container-1.discovery.cs.vt.edu)
   (b) Container 2 - GPU-backed container running Jupyter Notebooks+Pytorch (URL: https://team3-container-2.discovery.cs.vt.edu)

4. Team 4 - Team 4 requires three Docker containers running on Nodes that have native GPU support for supporting Machine Learning tasks that require hardware acceleration. These containers were provisioned on the CS Cloud Infrastructure. They are running Jupyter notebooks based on custom images (discussed in Section 6.2.5) and are exposed through the URLs mentioned alongside the container names and their intended use cases below:

   (a) Container 1 - For segmenting ETDs into chapters (URL: https://team4-container-1.discovery.cs.vt.edu)
   (b) Container 2 - For generating summaries of ETDs (URL: https://team4-container-2.discovery.cs.vt.edu)
   (c) Container 3 - For classification (URL: https://team4-container-3.discovery.cs.vt.edu)

5. Team 5 - Team 5 has 4 containers that it uses to orchestrate workflow automation. These are listed below. More information about the same is also available in Section 6.1.

   (a) PostgreSQL server (Registry), which houses the data about all the services, goals, and reasoner metadata (URL: https://team2020db.discovery.cs.vt.edu)

(b) Registry API server, which is the interface between the user and the registry database (URL: https://team2020-api.discovery.cs.vt.edu)

(c) Reasoner backend, which handles the generation and maintenance of the Context Free Grammar used to represent workflows (URL: http://reasoner.discovery.cs.vt.edu/)

(d) Apache Airflow, which is used for the execution of workflows that are represented as DAG files (URL: http://team2020-airflow.discovery.cs.vt.edu/)

6.2.5 Spinning up GPU Backed Containers

We faced many challenges while spinning up GPU-backed containers for Teams 3 and 4. The following steps were taken.

Cloud Hardware

CS Cloud Infrastructure has provisioned physical nodes running NVidia GPUs for our use, on which we can run Docker containers. These nodes are connected to the Kubernetes cluster and are available to us through the Kubernetes control plane.

Specifying usage of GPU Nodes

We use Kubernetes' NodeAffinity constraint to tie the deployment to a particular node. These nodes are selected by the labels that are assigned to each node. By specifying a nodeSelector label while defining a deployment, we can tie a particular deployment so that it only gets deployed on that node. The GPU-backed nodes in the CS Cloud infrastructure are assigned the label 'gpu'. The following YAML snippet allows us to do this:

```
spec:
  affinity:
    nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: type
                operator: In
                values:
               - gpu
```

Building a custom Docker Image

Team-4 communicated their requirements to us and mentioned that they required the NVidia base Docker image (https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch) along with the following dependencies:

1. nltk
2. tensorflow-gpu
3. pandas
4. scikit-learn
5. pickle
6. lxml
7. argparse
8. BeautifulSoup
9. nlp
10. transformers
11. spacy
12. seaborn
13. Matplotlib

To help the team get started quickly, we decided to package all of the dependencies along with the base Docker image in a custom Docker image (https://hub.docker.com/r/ano2202/cs5604_pytorch) that we published to Docker Hub. This image requires that the following hardware dependencies be specified when trying to run this image on our hardware [12]:

1. Docker Engine (https://docs.docker.com/get-docker/)
2. NVIDIA GPU Drivers (https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html)
3. NVIDIA Container Toolkit (https://github.com/NVIDIA/nvidia-docker)

Deployment

Following the steps that are mentioned in Section 8.1, in the image section of deploying the workload, we specify the following image name when entering the details: 'ano2202/cs5604_pytorch'. Since this image is publicly deployed on Docker Hub, it can be accessed by all containers in any cluster.

Testing

Once the containers are up we need to test whether the image and the container that it is running are indeed running on a GPU-backed node as well as able to leverage the capabilities of hardware acceleration. NVidia offers a simple command to test this called 'nvidia-smi'. To test this, navigate to the container in question and click on 'Execute Shell' as shown in Figure 6.16.

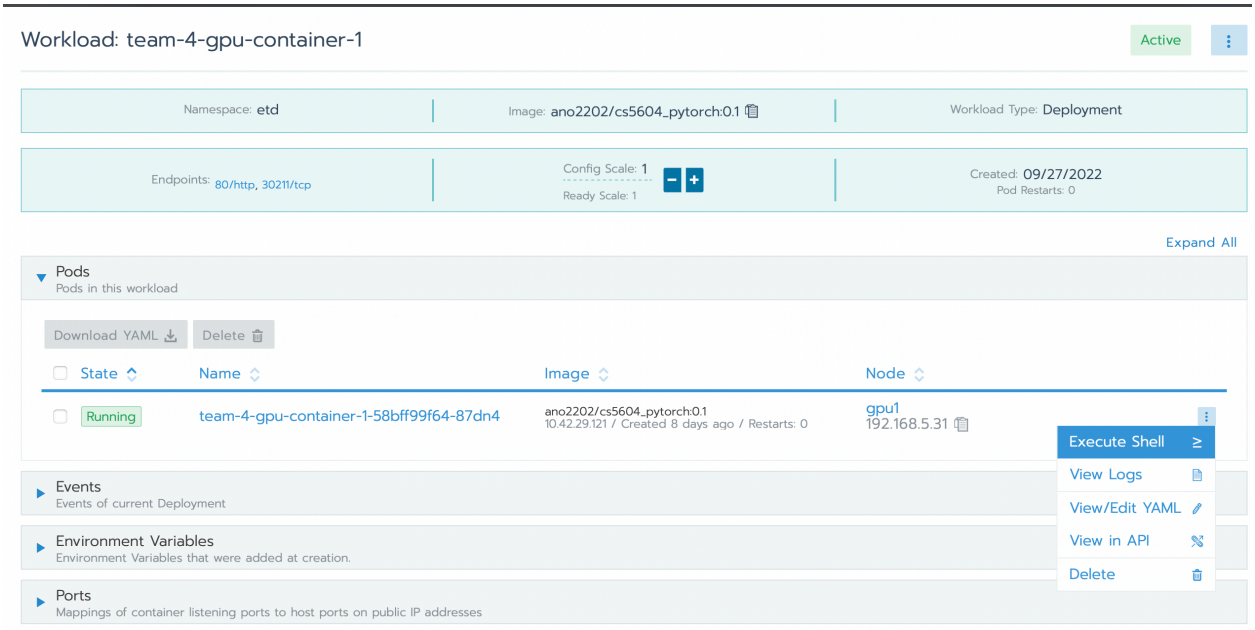

Figure 6.16: The option to execute a shell inside the container, present in the Rancher UI

If successful, it should display the result shown in Figure 6.17.

≥ Shell: team-4-gpu-container-1

ProTip: Hold the Command key when opening shell access to launch a new window.

```
root@team-4-gpu-container-1-58bff99f64-87dn4:/mnt# nvidia-smi
Thu Oct  6 06:48:15 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:10.0 Off |                  Off |
| N/A   38C    P0    27W /  70W |  12751MiB / 16384MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     18069      C                                   12749MiB |
+-----------------------------------------------------------------------------+
root@team-4-gpu-container-1-58bff99f64-87dn4:/mnt#
```

Figure 6.17: Successful execution of the 'nvidia-smi' command

6.3 Project Management

We decided to follow Beyer's User Centered Agile Methods [4] for project management and decided to employ Scrum [30] to manage our project. The monograph prescribes an extensive study of end-user requirements in the way of user interviews at their workplace, describing user personas, story-boarding, etc., and describes ways in which a typical software project can be managed such as sprint planning, the product owner role, user stories, etc. While the approach defined in the monograph is extensive, we decided to pick up a few concepts from it, for project management and user experience design. The following points highlight our project management approach:

1. Sprints: We decided to have one-week sprints. This was done because we wanted quick feedback while keeping in mind the overall duration of the course. A sprint board was created to track the progress of our tickets. Figure 6.18 displays the sprint board as of 09/15/2022.

2. User stories: We created rudimentary user stories out of the requirements as mentioned in Section 2. We suggest as part of Future Work to create more detailed user stories,

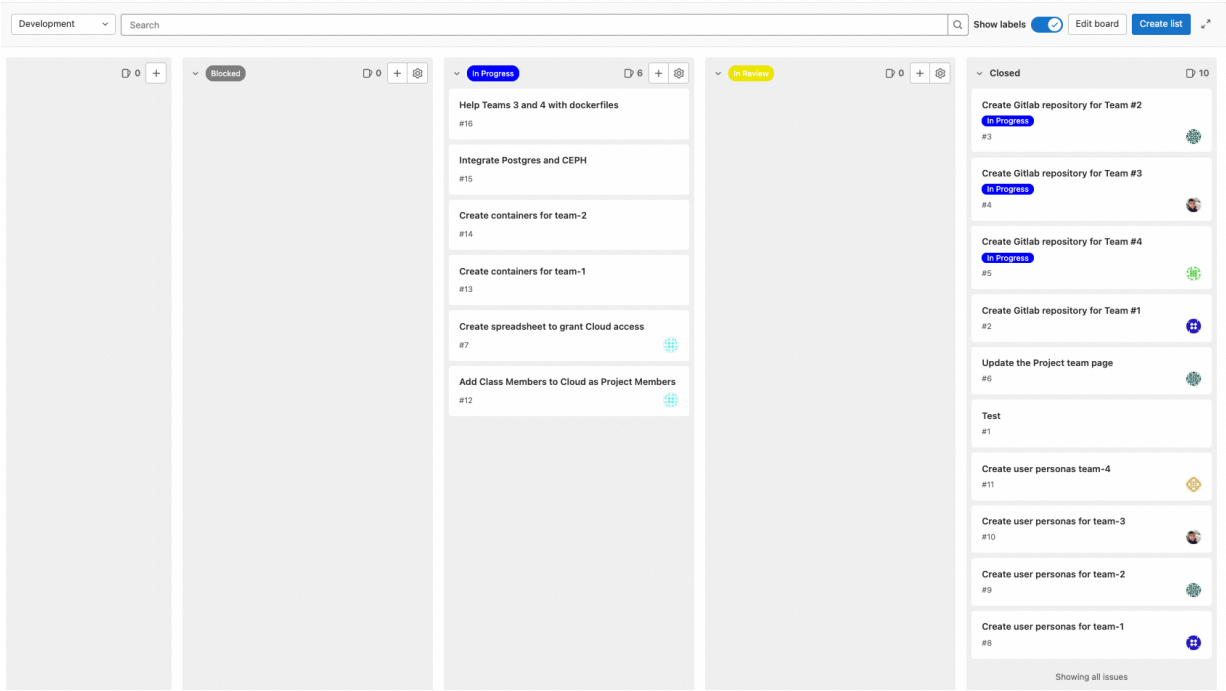

Figure 6.18: Sprint Board on GitLab as of 09/15/2022

based on the user personas that we have identified and listed in Section 2.2 and any other incoming requirements from the other teams.

3. Team meetings: Our team convenes in three fifty minutes long meetings on Tuesday and Thursday mornings as well as Sunday afternoons. While we discuss the day-to-day items on the weekdays, we reserve the Sunday meeting for sprint planning and also overall goal planning.

4. Product owner: Rather than appointing a person from our team as a representative for the user, we treat the liaisons from other teams as product owners. Their input is our source of user personas and stories.

6.3.1 Interviews

Interview Methodology - Liaisoning

We have appointed members to be liaisons of each team. The effort of liaisons is two-fold. For one, these liaisons will talk to their respective teams and understand their ideas and requirements to coordinate the creation of Docker containers, web servers, etc. Secondly, these liaisons will also help formulate the overall architecture, so that components with inter-dependency can be easily identified and built upon.

6.4 Timeline

#	Description	Week	Status
1	Conducting interviews with other teams	1	Done
2	Conducting onboarding and reading sessions to address knowledge gaps	1	Done
3	Finalizing team requirements	1	Done
4	Literature review of tools, technologies, and best practices to identify possible solution approaches	2	Done
5	Design and plan system and architecture diagram from requirements	3	Done
6	Setting up containers with the specific requirements	3	Done
7	Setting up training containers	4-5	Done
8	Setting up service containers	4-5	Done
9	Setting up a Continuous Integration pipeline to ensure code integration and testing practices are managed as a prototype within Team 5	6-7	Done
10	Setting up a Continuous Deployment pipeline prototype within Team 5	6-7	Done
11	Conducting literature review and understanding the workflow automation completed thus far	6	Done
12	Setting up a Continuous Integration and Continuous Deployment pipeline for each team	8-9	Done
13	Developing a prototype of the workflow automation platform using Airflow and a singular goal of object detection	8-9	Done
14	Setting up a functioning repository for the knowledge graph, goals, workflows, and services	10-11	Done
15	Planning and designing our content for the registry User Interface	12	Done
16	Implementing the registry User Interface	12	Done
17	Liaison with teams to get their services containerized - We will be starting with Team-4	12	Done
18	Setting up the reasoner/optimizer	12	Done
19	Deploying the entire set-up on Rancher and testing it	15	Done
20	End-to-end tweaking/testing along with final demonstration	17	Done
21	Making PostgreSQL highly available and redundant with the use of persistent storage such as Ceph or a VM	NA	Future work
22	Enabling observability and monitoring of entities within the cluster using Grafana	NA	Future work

Table 6.2: Project Timeline

6.5 Milestones and Deliverables

1. Manual deployment of all teams' containers
2. Provide version control for all teams
3. Containers should be up and running with reliability, speed, and resilience
4. Team should be able to version control using Git; integrate, test, and deploy their business logic to the cloud, seamlessly
5. Create container and version control repository for the front-end team
6. Team artifacts (containers) should be able to communicate with each other and teams should be able to integrate their responsibilities
7. Develop Continuous Integration and Continuous Deployment pipeline that helps to test, build and deploy the codes of the various teams
8. Integrate the workflow management system based on user-defined goals
9. Identify and understand the pre-existing work done concerning workflow management
10. Design an efficient architecture for the workflow management pipeline
11. Set up the database with the required services, goals, knowledge graph data
12. End-to-end prototype of the workflow management system
13. User Interface that helps to create and interact with the workflow management system
14. Be available to provide support in case of systemic anomalies/failures and technical difficulties faced by other teams

Chapter 7

User Manual

7.1 Rancher: Accessing the Containers

After the containers have been deployed, there are a few steps that must be followed on the Rancher UI to access the containers.

1. Login to the VT CS Cloud website (Figure 7.1), then navigate to the group the containers reside on.

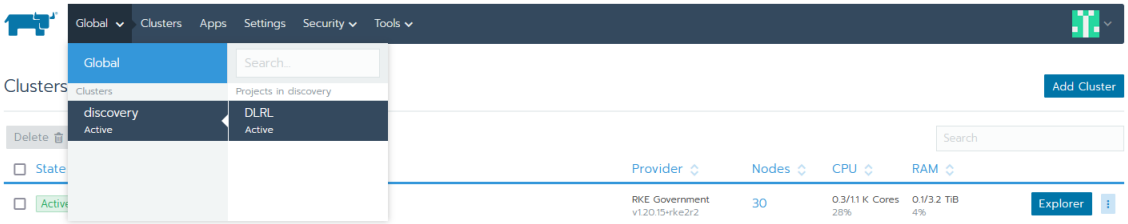

Figure 7.1: CS Cloud Website

2. In the Workloads tab, select the container you wish to connect to (Figure 7.2).

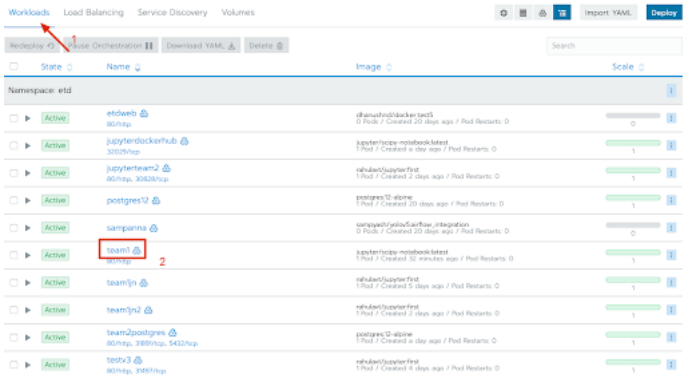

Figure 7.2: Container selection

3. Once you have been brought to the container's settings page, navigate to the pod section and click on the menu icon, and then select 'View Logs' (Figure 7.3).

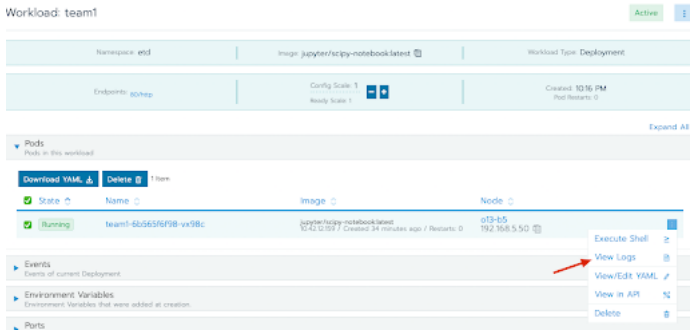

Figure 7.3: Containers logs

4. Inside the terminal that will pop up, scroll until you can see a URL ending with 'token=' and a large string of characters (Figure 7.4). Copy these characters, save them somewhere on your machine and then you can exit the terminal.

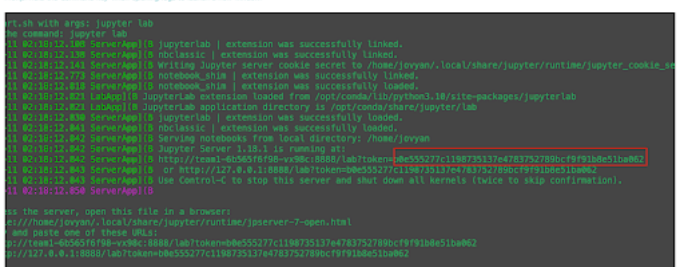

Figure 7.4: Finding the token

5. Next, navigate back to the workloads page that contains all of the containers. Select the '80/http' option directly below the desired container.

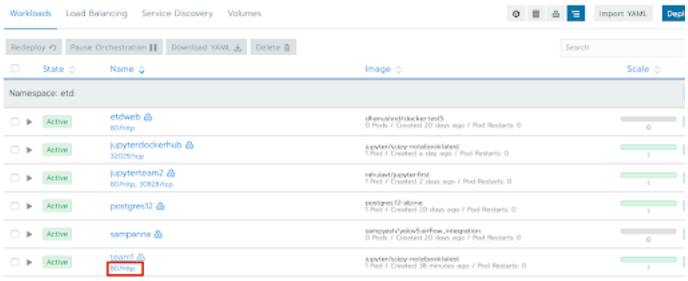

Figure 7.5: Container's HTTP Connection

6. Finally, you will be brought to a new page with a Jupyter notebook page open. In the text box at the top of the page, paste the token in and then select login.

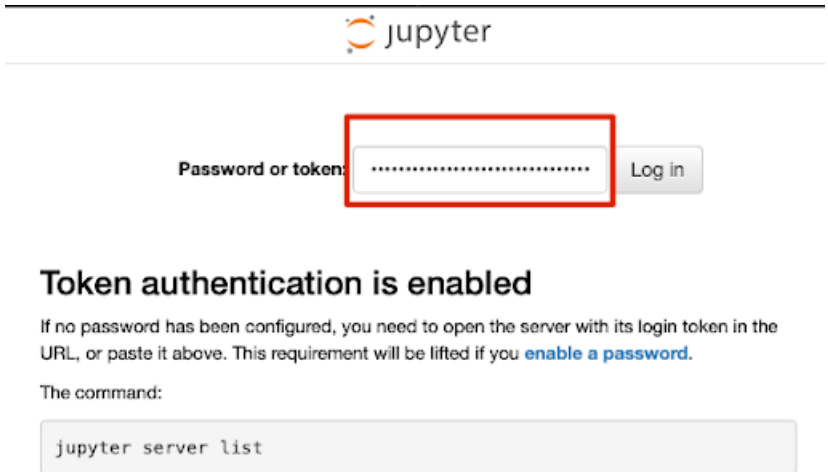

Figure 7.6: Jupyter Login

7. Now you will officially be connected to the container's Jupyter Notebook, and you will have access to the file system and be ready to develop.

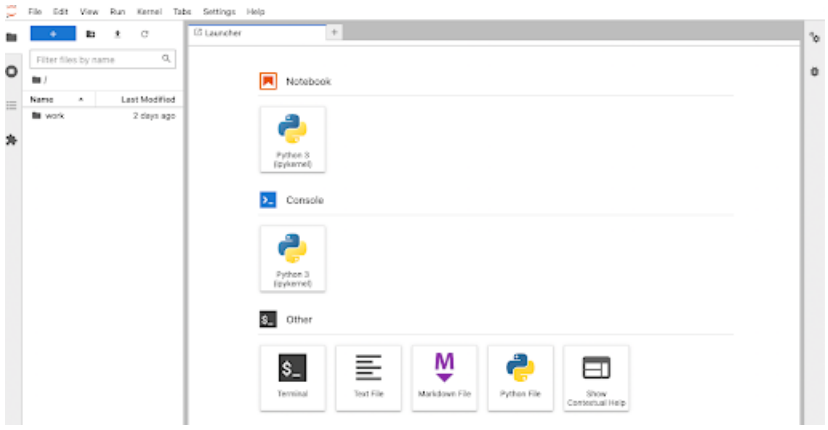

Figure 7.7: Jupyter Homepage

7.2 Adding a new workflow automation entry

The steps to create a new workflow automation entry to the registry are as follows:

1. Navigate to `https://frontend.discovery.cs.vt.edu/workflows` and log in.

2. Create a goal(s)

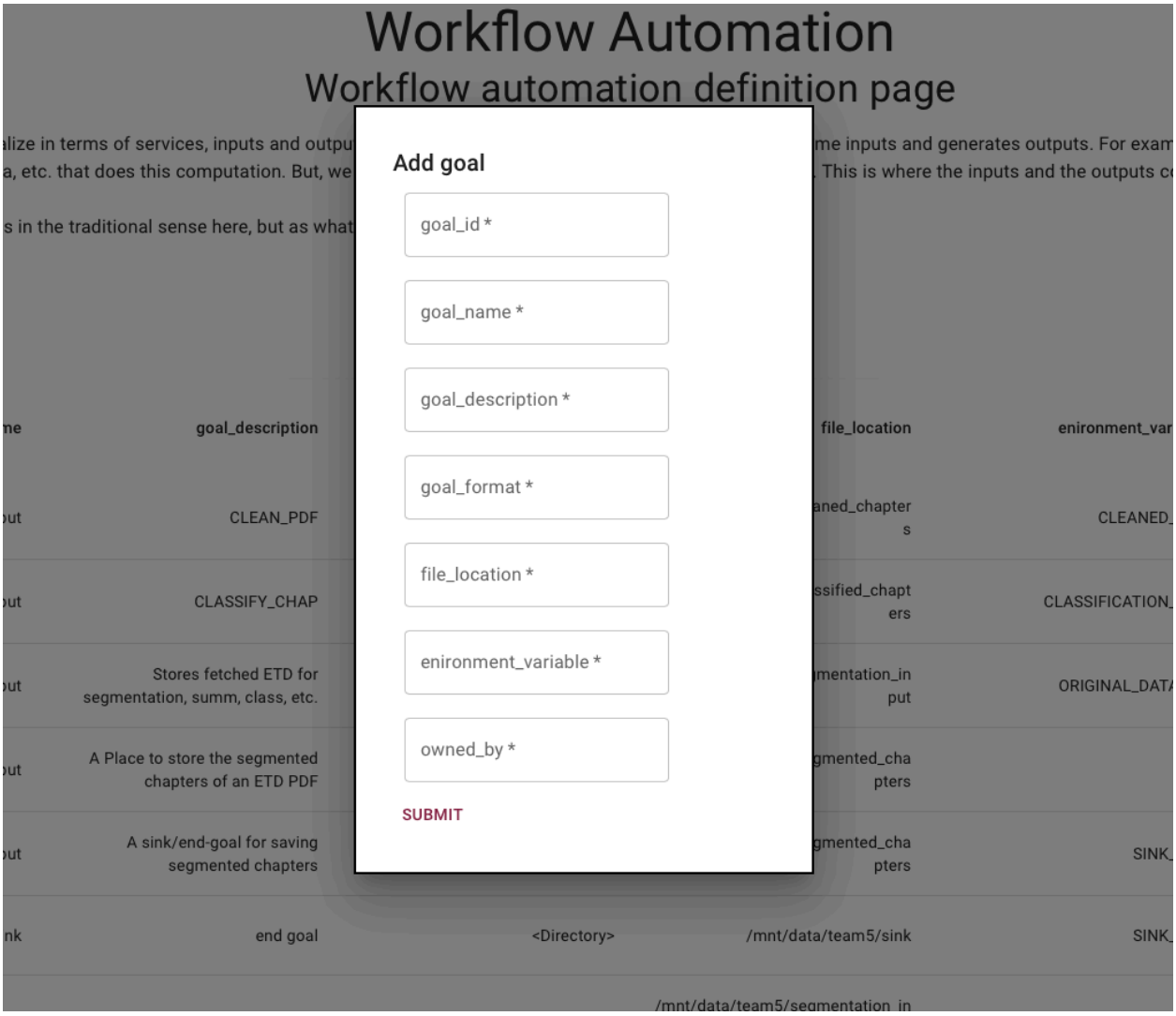

Figure 7.8: Add a goal

3. Create a service(s)

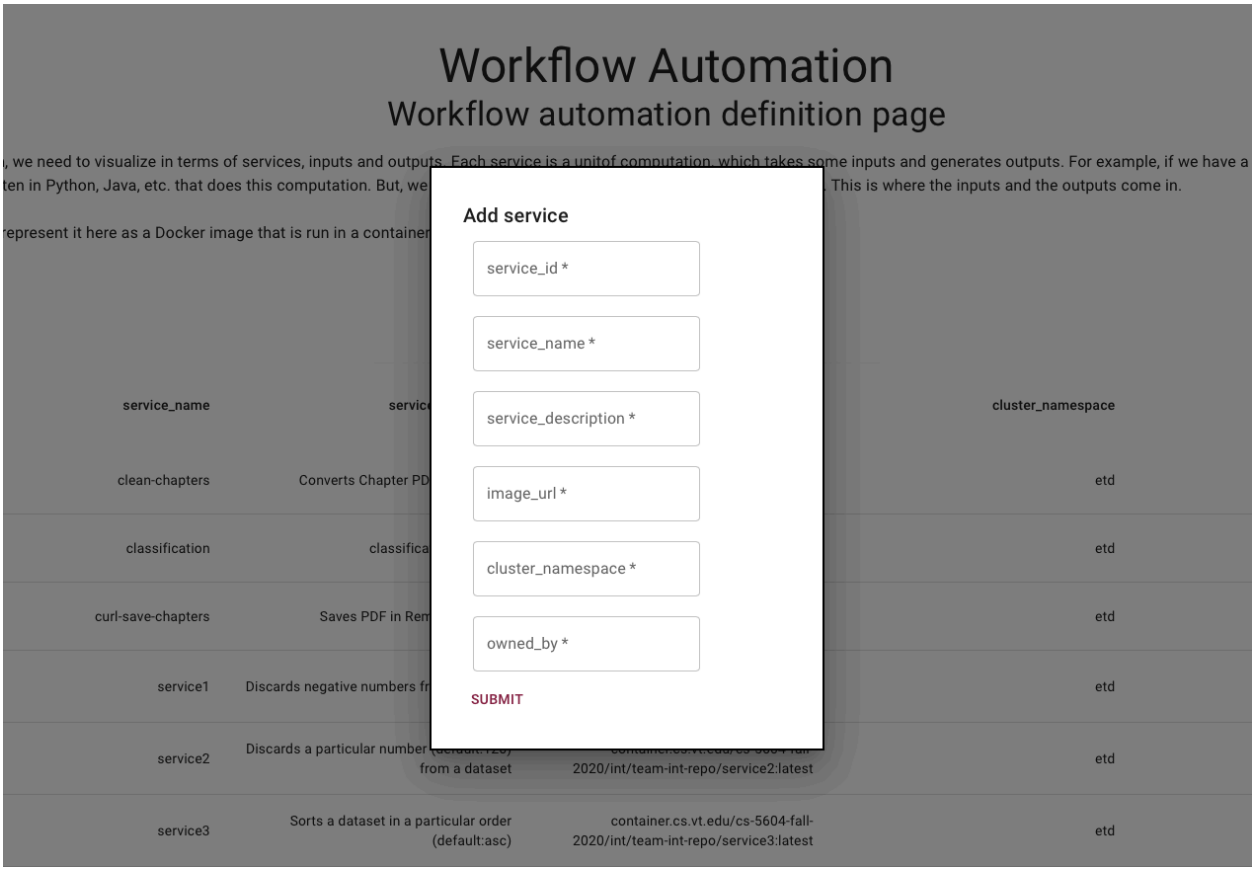

Figure 7.9: Add a service

4. Link the created goals and services using the reasoner

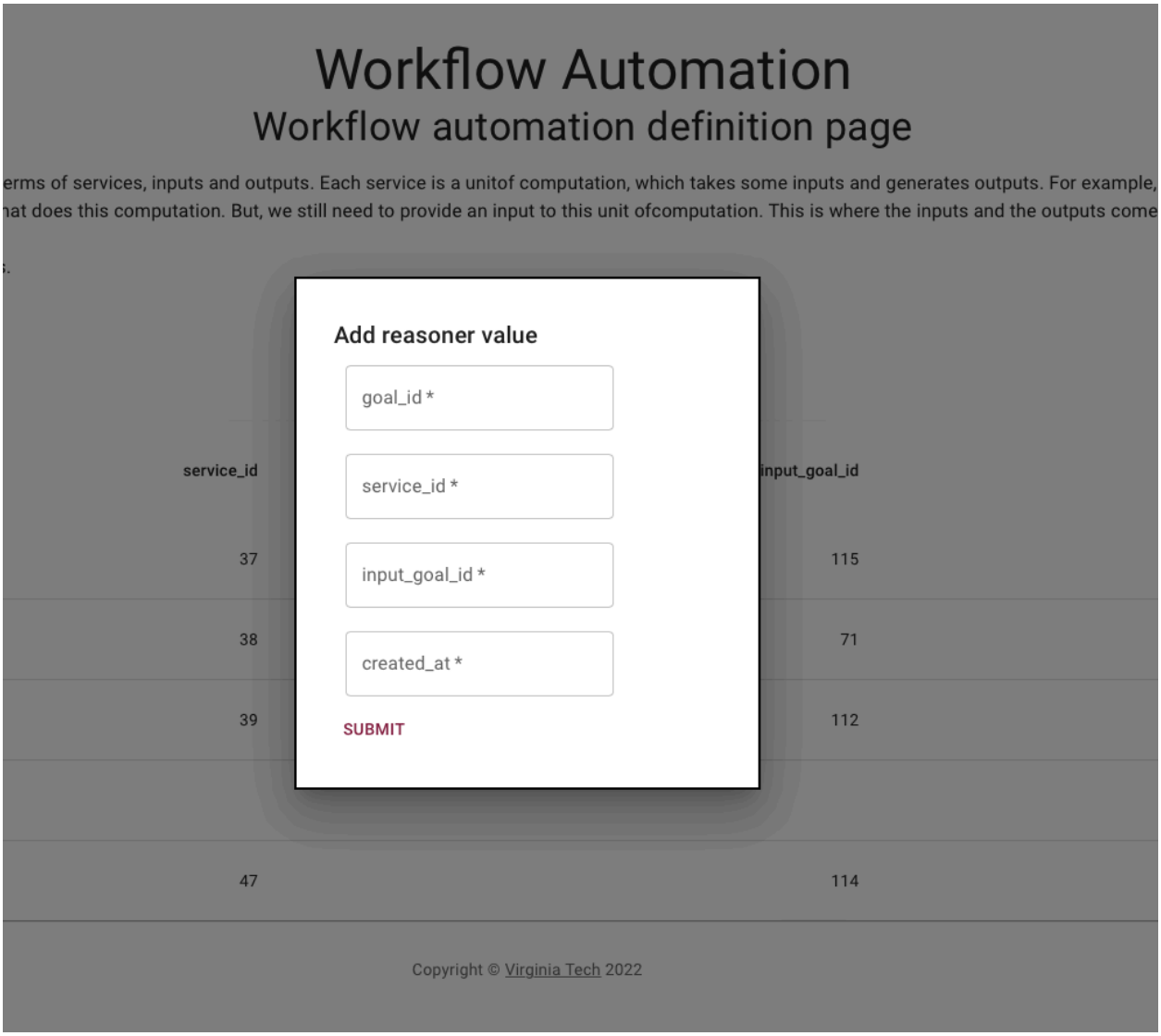

Figure 7.10: Link goals and services together by creating a reasoner entry

7.2.1 Result

This will result in a new entry being created in the Registry database that houses all the data necessary to generate a workflow. This will result in an entry being created similar to the database entries shown below:

goal_id	goal_name	goal_description	goal_format	file_location	environment_variable	owned_by
89	summarization_output	SUMMARIZE_CHAP	<Directory>	/mnt/data/team5/summarized_chapters	SUMM_ENV	INT
90	classification_output	CLASSIFY_CHAP	<Directory>	/mnt/data/team5/classified_chapters	CLASSIFICATION_ENV	INT

Figure 7.11: A sample list of goals created in the Registry database

service_id	service_name	service_description	image_url	cluster_namespace	owned_by
49	classification	classification chapters	outerspace1920/classification:1.1	etd	INT

Figure 7.12: A sample service created in the Registry database

goal_id	service_id	input_goal_id
90	49	89

Figure 7.13: A sample reasoner metadata entry created in the Registry database

Chapter 8

Developer Manual

8.1 Deploying a Container

Requirements: Owner-level role access to the DLRL cluster on CS Cloud

Additional steps on how to expose this container using a Kubernetes Ingress can be found in a video guide we prepared [27].

1. Navigate to https://cloud.cs.vt.edu (Figure 8.1).

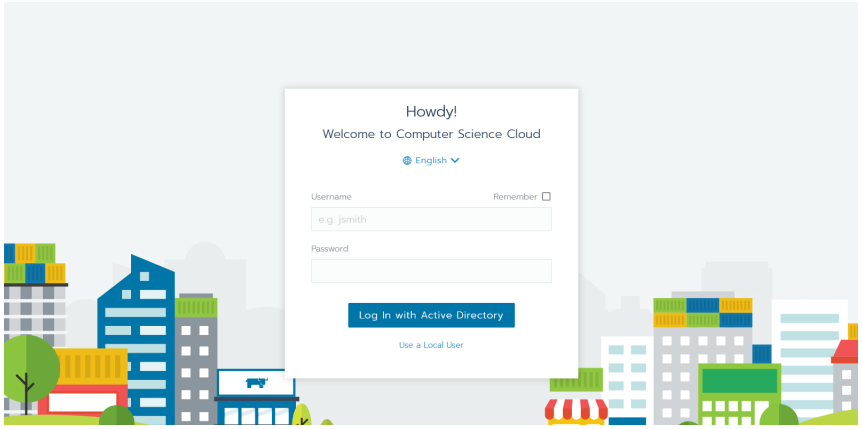

Figure 8.1: Cloud CS Login Page

2. Use your CS username and password to log in
3. Click on the "Discovery" cluster and navigate to the DLRL project (Figure 8.2).

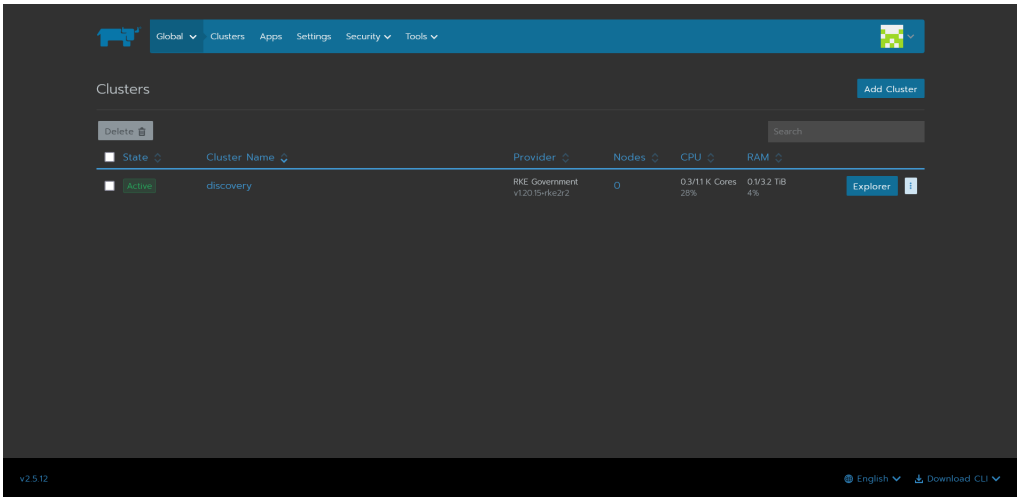

Figure 8.2: CS Cloud Homepage

4. Click on deploy (Figure 8.3).

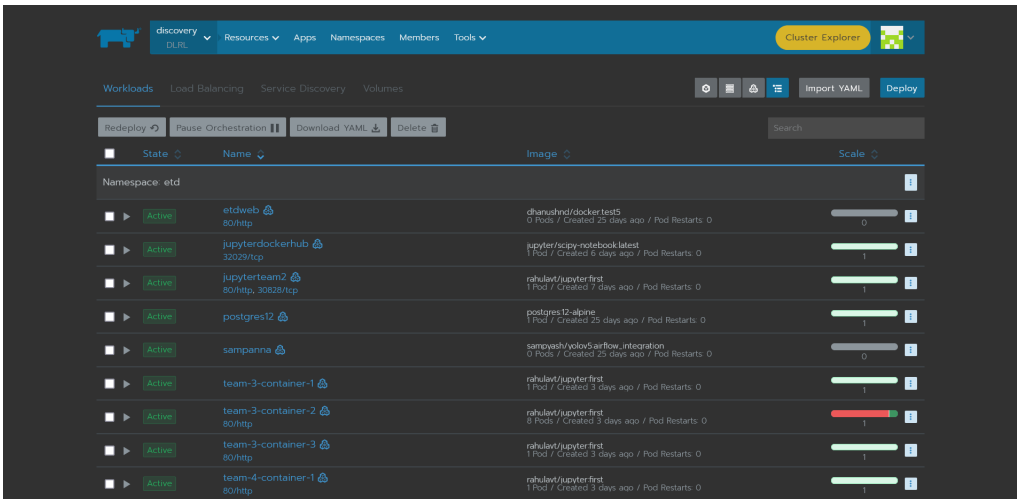

Figure 8.3: Workloads

5. Figure 8.4 shows the configuration for the container. The name is just the name of the container. The Docker Image can be any image from the Docker Hub. A namespace can be thought of as a logical folder space wherein workloads/services can be placed. Ports can be mapped. A node port is a port that can be used to explicitly expose the port of a pod on a node. Hostport, like the standard Docker port, is used to expose the container port. A Node port has a value between 30,000 to 33,000 and is generally used along with a load balancer.

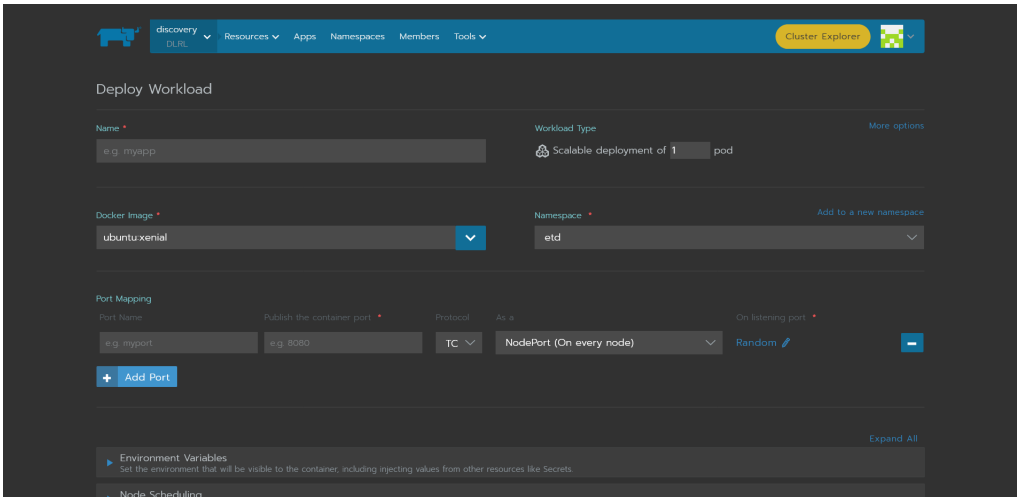

Figure 8.4: Workload deployment page

6. After that we see the environment variables configuration options along with node scheduling which can be used to deploy to specific nodes (Figure 8.5).

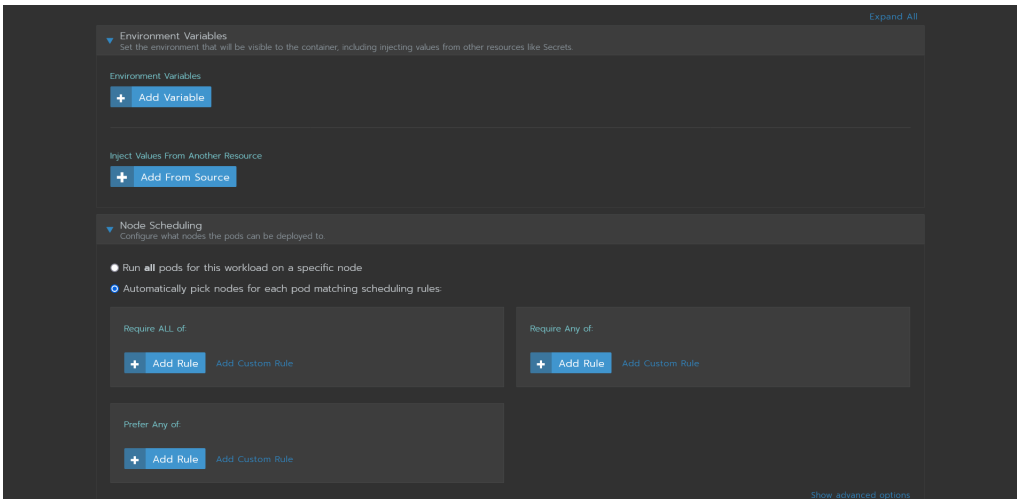

Figure 8.5: Environment Variables and Scheduling

7. The most important section is the volumes. To have persistent storage across restarts and failures, we need to use a certain persistent mount point or storage. The CS cloud has a managed Ceph container setup that is ready to use. Input the supplied folder and credentials, and monitor IPs to gain access (Figure 8.6).

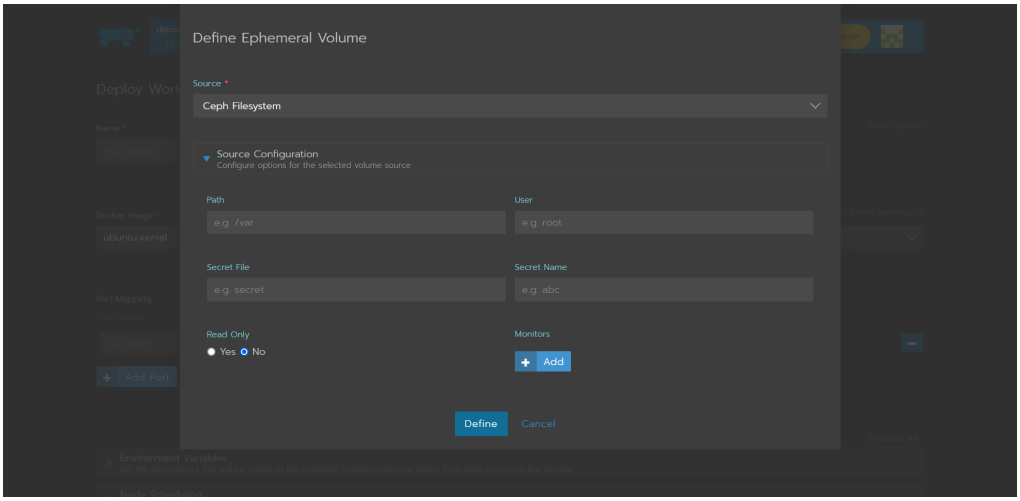

Figure 8.6: Attaching a Ceph filesystem

8. After attaching Ceph, we define the mount point (Figure 8.7).

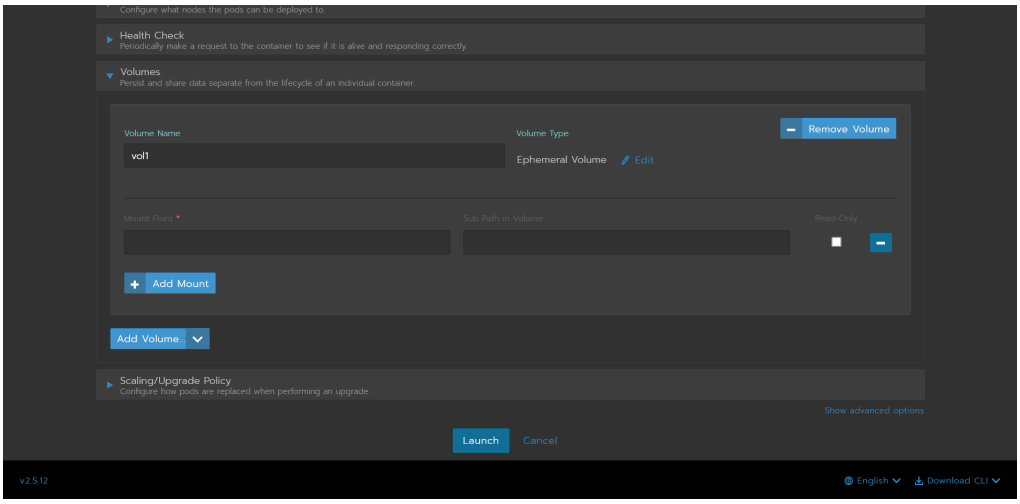

Figure 8.7: Mount configurations

9. Once we click on Launch, the system will spin up the image. If that succeeded, a pod will be created successfully as seen in Figure 8.8.

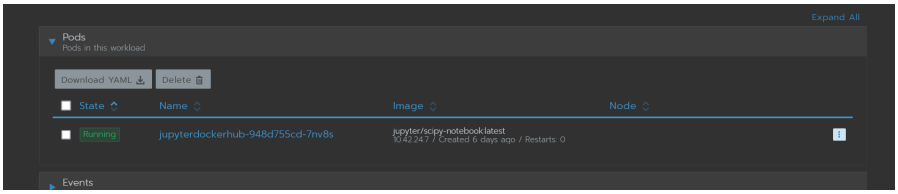

Figure 8.8: Running Pods

10. We expose various containers that we create using the Kubernetes resource definition of the Ingress, which is internally implemented as a Load balancer in Level-7 of the OSI model. We define this in the 'Load Balancer' tab. Click on Add and a configuration page will open up (Figure 8.9).

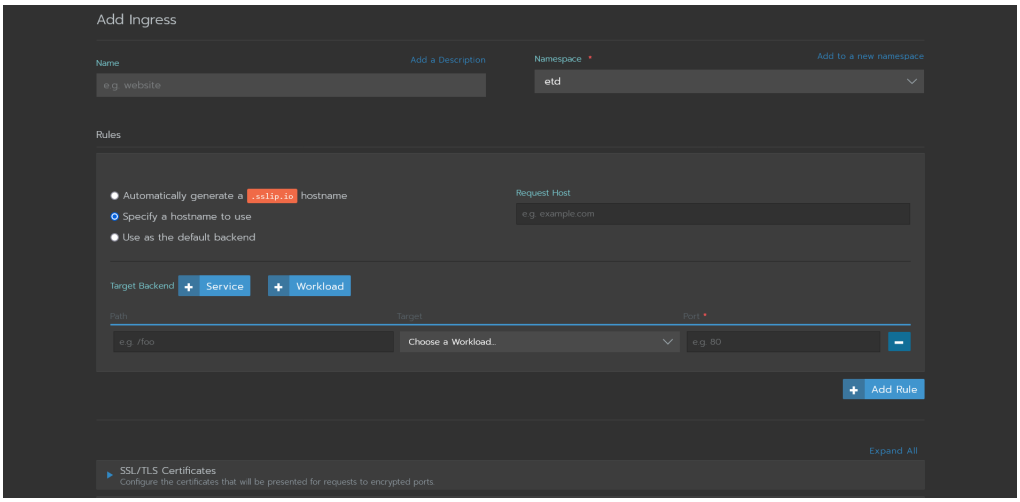

Figure 8.9: Load Balancer Config

11. These deployments which contain pods are not publicly accessible. They can be used by other pods in other deployments to exchange data; this exposed IP is a local IP. To access, for example, a Web Server, a service called Ingress is required. This ingress acts as a connection between the outside internet and the deployment. Add a user-friendly name and assign it a namespace. Click on 'Specify hostname' to use a hostname of the form xxxx.discovery.cs.vt.edu. Use '/' as the path, and the workload will be the container that we just set up. The port will be the one that we choose to expose. Click on save and if it succeeds, the newly created container will be available on xxxx.discovery.cs.vt.edu if it happens to be a web server-based container.

8.1.1 Troubleshooting

(a) Frequently, you might encounter the error **CrashLoopBackOff: "Back-off 10s restarting failed container=web pod=xxxxxxx(461c937d-d870-11e6-**

98de-005056040cc2)”. All this error means is that your container failed to initialize. This could be due to various errors such as incorrect storage credentials, entry point script errors, insufficient resources, etc. The following 2 things might be useful for diagnosis:

i. Container Logs: This is the most useful way to find the reason why the container is crashing. To access the logs of a container, click on a pod and then click on the 3 dots near the top right to see a small menu (see Figure 8.10). Click on ‘View Logs’. This will open up a pop-up with all the logs displayed as seen in Figure 8.11

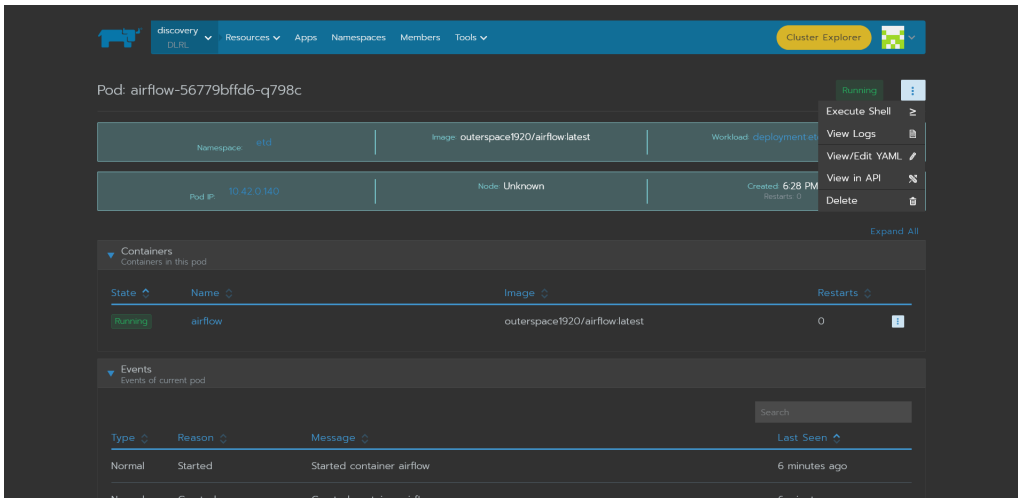

Figure 8.10: Option to see Container Logs

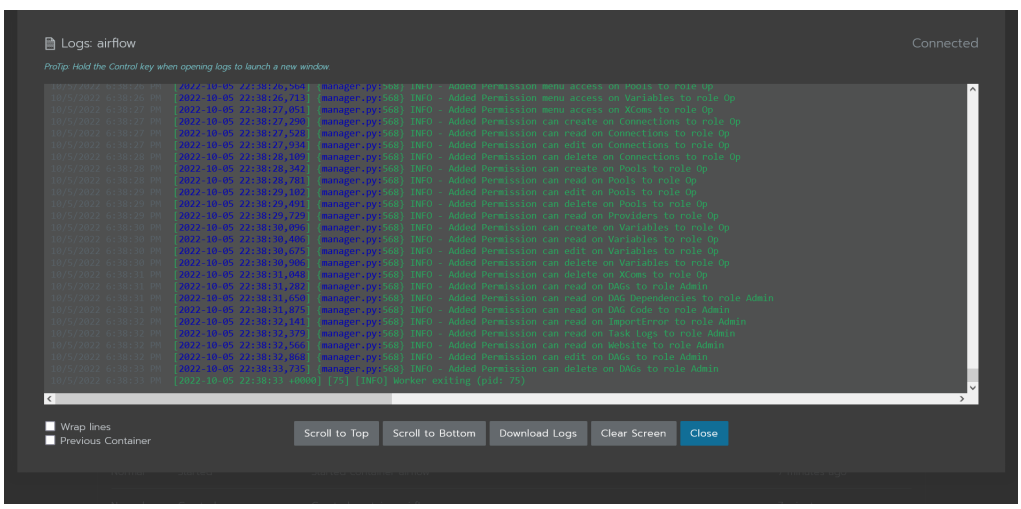

Figure 8.11: Container Logs

ii. Pod Events: Once you click on the pod, you can view the events such as image pull, container initialization, etc. in the 'Events' section as seen in Figure 8.12. Although not the most useful as it does not state reasons why a Pod crashed, it can still output errors during image pull, for example a 401 error when attempting to pull from a private registry whose access tokens have not been set up in Rancher.

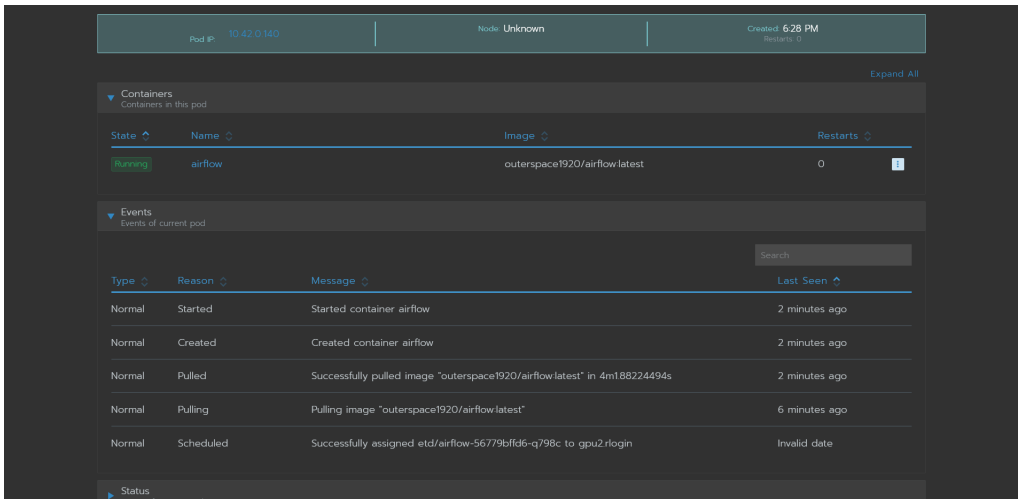

Figure 8.12: Pod Events

8.2 Granting Access to Team Members on CS Cloud and GitLab

1. Students from the CS department can directly set up their account on cloud.cs.vt.edu by setting up a password

2. Students from other departments need specified access permissions to be able to access their accounts for cloud.cs.vt.edu

3. The INT team is given Owner Access to be able to create and modify containers, adding other members

4. Other team members are given access by assigning Project Member status

8.3 Using the Reasoner APIs

In order to use the Reasoner and Airflow APIs, there are 2 ways: use the existing functions in the Frontend Repository, or use the bare APIs. Following are the ways to use each.

8.3.1 Functions

These functions are available only if you are integrating any code into the Frontend code. The file with the following URL https://code.vt.edu/aaron2000/cs5604-front-end/-/blob/main/src/api/AirflowReasoner.js contains the functions. We will now give a description for each, and explain their order of execution.

1. generateWorkflow: This functions takes 2 parameters as input: workflowID and the payload. The Workflow ID is the ID used to run a Workflow. For example, for summarization and classification, the workflowID is 124. Payload is the JSON object that contains the environment variables to be passed. The Response Body contains the generatedWorkflowID that needs to be passed to the runWorkflow function.

2. runWorkflow: As stated above, it takes the generatedWorkflowID as input and an array of service names that can be used to provide user understandable names for various services run. The result of this is a status link that provides the status of the run along with the run key.

3. getWorkflowStatus: The input to this function is the run key which is obtained from the runWorkflow function. The output gives the status. When the workflow has succeeded or failed, the logs are returned.

Now, the order to run these is just 1, 2, and then 3.

8.3.2 Bare APIs

For projects not using the Frontend UI, the APIs will need to be integrated directly. Section 6.1.2 has a detailed explanation for each API along with the order to run each.

8.4 GitLab CI/CD

To set up Continuous Integration and Continuous Deployment, a DevOps Pipeline needs to be set up. This pipeline takes code, performs some tasks such as building or testing, and then deploys it to some location. This is the gist of what a pipeline does. Pipelines can also

be increasingly complex. Since we are using GitLab as our main repository for storing all the code, we will also make use of GitLab's own DevOps, also called the GitLab Runner. Following are the ways to set up the GitLab runner and also configure it to make use of the Docker executor.

As mentioned before, you need to have Docker installed on the VM. It is sufficient to install the Docker Engine since the only interface will be the command line. To install Docker, the following steps need to be performed which are for Ubuntu and are hence, OS-dependent.

1. Go to https://docs.docker.com/engine/install/ and select your Operating System. Follow the steps to install Docker.

2. Once installed, start the Docker service using the command

```
sudo service docker start
```

3. Verify that the Docker install and startup were successful by running a test image for example

```
sudo docker run hello-world
```

Now that the Docker Engine is installed, the GitLab Runner can be installed. We recommend using the Linux Distribution so that the local Docker Engine can be used without the need for the installation of any specific Programming Language libraries.

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

2. Verify the installation by running

```
sudo gitlab-runner status
```

3. We now need to Register the Runner. In your Repository (here we are using a Repository in https://code.vt.edu), Go to Settings (Left Menu) -> CI/CD -> Runners (see Figure 8.13)

4. Note the URL and Registration token.

5. Now run the following command

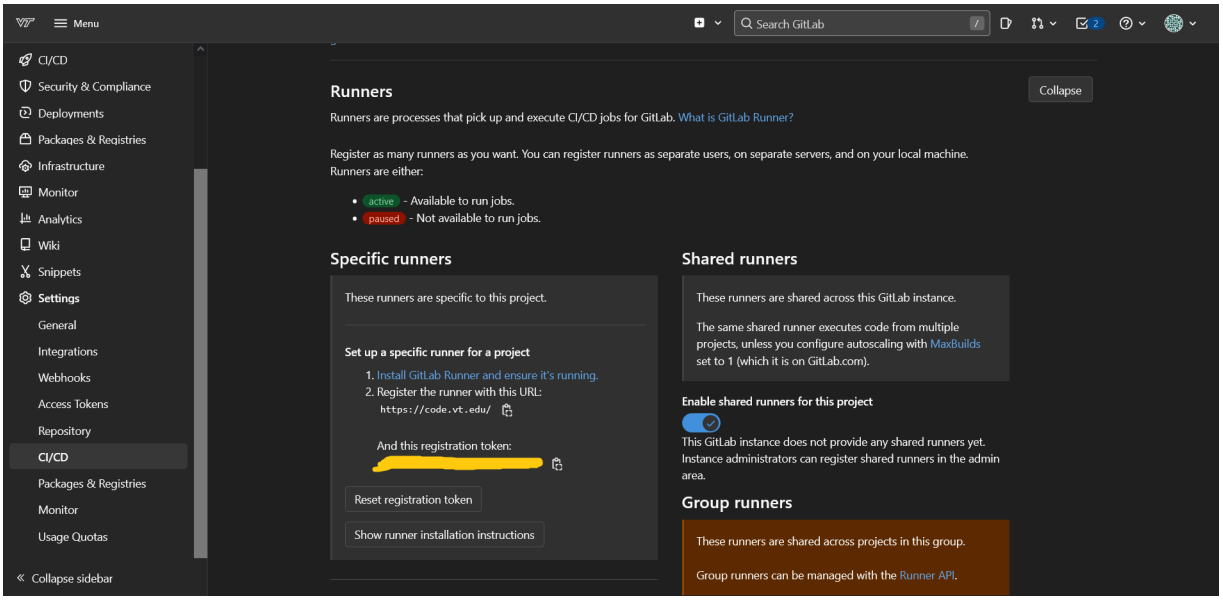

Figure 8.13: GitLab Runner Configuration

```
sudo gitlab-runner register
```

6. Enter all the details as asked and select Docker as the executor and docker:stable as the default image.

7. Run the following command to start the runner

```
sudo gitlab-runner start
```

   and once started, it can be seen in the 'Specific Runners' section as shown in Figure 8.14.

8. Unfortunately, the pipeline will fail due to the Docker executor not having access to the host's Docker engine. To fix this, the config.toml file needs to be modified. This file is located at

```
/etc/gitlab-runner/
```

9. Once opened, the privileged option should be set to true and the volumes should include the Docker socket as in the host Docker socket should be mounted to the Containers.

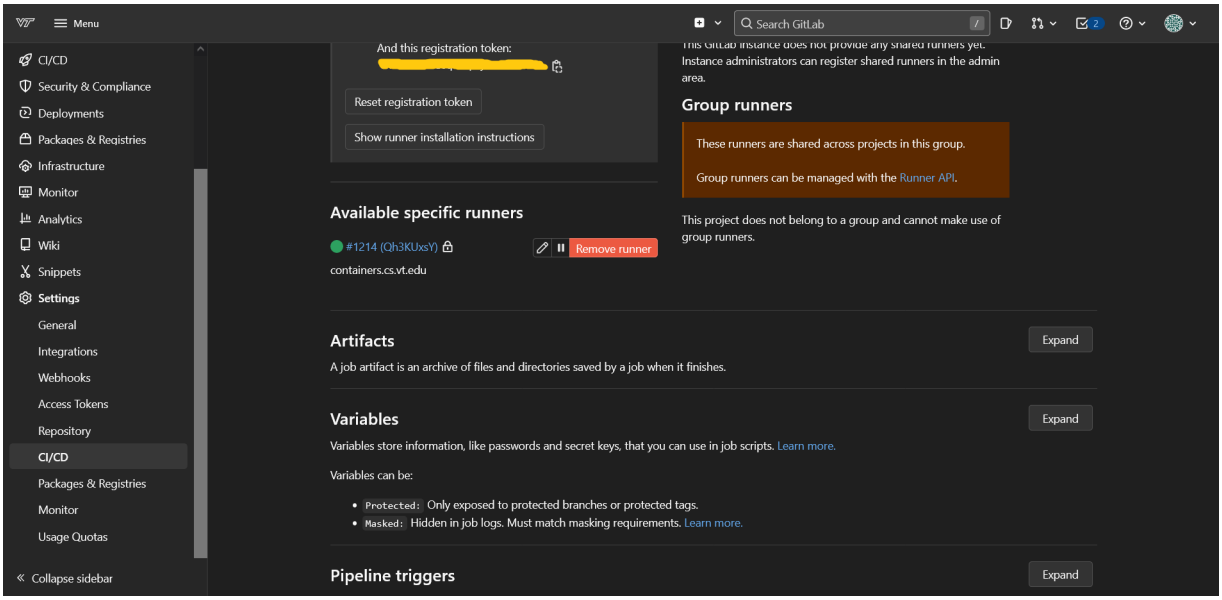

Figure 8.14: Available GitLab Runners

```
privileged = true
volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]
```

Figure 8.15 has the working config file with the changes highlighted.

8.4.1 Troubleshooting

One of the main problems arises when one tries to deploy the GitLab Runner onto Kubernetes, for example Rancher. This happens when Docker is selected as the executor. Since the Container within the Pod does not have full root access to the underlying file system, various permission errors get thrown.

Another problem arises when Podman is used instead. Podman is a daemonless container management system similar to Docker. Unfortunately, Podman fails to execute as well within a pod since it requires additional subuids and subgids. If you are trying to deploy your instance of the GitLab Runner, we, therefore, recommend running the GitLab Runner inside a dedicated VM with full root access.

```
concurrent = 1
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "containers.cs.vt.edu"
  url = "https://code.vt.edu/"
  id = 1214
  token_obtained_at = 2022-09-29T15:11:45Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "docker:stable"
    privileged = true
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache", "/home/aaron/kube/config:/.kube/config"]
    shm_size = 0
```

Figure 8.15: GitLab Runner's Config file

8.5 Cloud CS and GitLab Runner Resilience

As with a system, both Cloud CS and the Virtual Machine upon which the Runner is hosted can experience outages. It is good to note that code.vt.edu is not expected to have significant issues since it is hosted on Amazon Web Services. The following 2 sections talk about what measures have been put in place and what more can be done.

8.5.1 GitLab Runner

Both Docker and the GitLab Runner have been configured to run during OS initialization. This prevents manual intervention. Moreover, because the Runner uses Docker as the executor, no additional software needs to be installed as all software needed can be used from Docker images.

8.5.2 Cloud CS

Since Cloud CS runs Kubernetes, in case of Pod failure, they get restarted automatically. This restart only happens if a deployment is created. A deployment can manage pods and their scaling.

Although Kubernetes automatically manages pods, their individual file system is ephemeral. It thus becomes pertinent to mount specific data folders to persistent storage such as Ceph.

PostgreSQL

PostgreSQL, which is the main Database of the class, needs to be mounted. Otherwise, after a restart, all data will be lost. As seen in Figure 8.16, the env PGDATA is defined which specifies where the PostgreSQL data should be stored, and we mount Ceph to /var/lib/postgresql/data.

```
171      containers:
172      - env:
173        - name: PGDATA
174          value: /var/lib/postgresql/data/team1/postgres/data
175        - name: POSTGRES_PASSWORD
176          value: docker
177        - name: POSTGRES_USER
178          value: docker
179        volumeMounts:
180        - mountPath: /var/lib/postgresql/data
181          name: vol1
```

Figure 8.16: Team 1s Database volume configuration

If we now view the contents of the directory, /team1/postgres/data, we can see all PostgreSQL database data. See Figure 8.17.

Elasticsearch

Similar to PostgreSQL, Team 2's Elasticsearch is also mounted in Ceph. This provides persistence for the indexes. See Figure 8.18.

```
root@ano-test-container-7f5cd9f56c-bgw2z:/mnt/data/team1/postgres/data# pwd
/mnt/data/team1/postgres/data
root@ano-test-container-7f5cd9f56c-bgw2z:/mnt/data/team1/postgres/data# ls -la
total 38
drwx------ 1 70 root    24 Oct 29 00:03 .
drwxr-xr-x 1 70 root     1 Sep 27 02:35 ..
-rw------- 1 70   70     3 Sep 27 02:35 PG_VERSION
drwx------ 1 70   70     6 Oct 31 07:54 base
drwx------ 1 70   70    58 Oct 31 07:47 global
drwx------ 1 70   70     0 Sep 27 02:35 pg_commit_ts
drwx------ 1 70   70     0 Sep 27 02:35 pg_dynshmem
-rw------- 1 70   70  4821 Sep 27 02:36 pg_hba.conf
-rw------- 1 70   70  1636 Sep 27 02:35 pg_ident.conf
drwx------ 1 70   70     3 Dec  3 22:47 pg_logical
drwx------ 1 70   70     2 Sep 27 02:35 pg_multixact
drwx------ 1 70   70     0 Sep 27 02:35 pg_notify
drwx------ 1 70   70     0 Sep 27 02:35 pg_replslot
drwx------ 1 70   70     0 Sep 27 02:35 pg_serial
drwx------ 1 70   70     0 Sep 27 02:35 pg_snapshots
drwx------ 1 70   70     0 Oct 29 00:03 pg_stat
drwx------ 1 70   70     6 Dec  3 22:47 pg_stat_tmp
drwx------ 1 70   70     1 Dec  3 22:17 pg_subtrans
drwx------ 1 70   70     0 Sep 27 02:35 pg_tblspc
drwx------ 1 70   70     0 Sep 27 02:35 pg_twophase
drwx------ 1 70   70     6 Dec  3 22:27 pg_wal
drwx------ 1 70   70     8 Dec  3 17:42 pg_xact
-rw------- 1 70   70    88 Sep 27 02:35 postgresql.auto.conf
-rw------- 1 70   70 28718 Sep 27 02:35 postgresql.conf
-rw------- 1 70   70    24 Oct 29 00:03 postmaster.opts
-rw------- 1 70   70   114 Oct 29 00:03 postmaster.pid
root@ano-test-container-7f5cd9f56c-bgw2z:/mnt/data/team1/postgres/data#
```

Figure 8.17: Contents of the /team1/postgres/data directory in Ceph

```
      volumeMounts:
      - mountPath: /usr/share/elasticsearch/data
        name: vol2
    volumes:
    - cephfs:
        monitors:
        - 192.168.5.15
        - 192.168.5.16
        - 192.168.5.17
        path: /courses/cs5604/team2/elastic/data
        secretRef:
          name: ceph-key
        user: cs5604
      name: vol2
```

Figure 8.18: Team 2's Elasticsearch volume configuration

Chapter 9

Future Work

1. Workflow Automation

   (a) Support Future Services Automation: Currently, we only integrated indexing, object detection, segmentation, summarization, classification, clean & parse, and saving objects into the workflow. But services such as topic modeling, or any other coming services could be automated and tested in the future.

   (b) Reasoner Optimization: Currently, we only automated the basic functionality for each service. However, future optimization will be considered such as persisting intermediate data. For example, if two workflows overlap in some steps, the succeeding workflow can directly read the intermediate output produced by the former workflow, instead of executing the workflow end-to-end. See also Section 6.1.10.

   (c) System Performance Analysis: We recommend to run some experiments to consider the trade-off between system performance along with the decoupled services. For example, from a software engineering and DevOps perspective, we ensure each service/container image is simple enough to perform only one task. However, in some cases executing several consecutive services/containers will drag down the efficiency, compared to executing only one monolithic service. Thus, we need to consider merging some insignificant/lightweight services, for better response time.

   (d) Efficiency: Google's use of commercial hardware [45] to run distributed databases is an industry method to lower costs while also performing on par with multi-core processors. Therefore, this should also be a goal, i.e., to have the capability of processing multiple ETDs without using expensive hardware. This means, workflows should be able to run in parallel but be limited so as to not crash individual nodes. This can be easily achieved with the help of a Queue such as with Apache Kafka or RabbitMQ. Queues need to be durable so that in the event of an outage, the messages in the queue are not lost. Workflows are queued and a Master uses heuristics to determine if a particular workflow can be run or not depending on various factors such as No. of Workflows in progress, Current Available Capacity of System, whether the Workflow has special requirements, etc. Services can also be modified to run processes in batches, thus eliminating pod spin up time for each ETD or ETD chapter. We suggest, in order to process 500,000 ETDs in a reasonable amount of time, a combination of the above should be used with the aim of using all of the current nodes. While we do not expect

'reasonable amount of time' to be on the order of hours, a couple of days would be reasonable.

2. Using Virtuoso Team 1's Virtuoso can also be extended to contain a graph of all ETDs and their digital objects. This database can thus contain the entire network of ETDs.

3. CI/CD

   (a) Unifying Workflow Registry PostgreSQL container with the main one: As the final step in the project, we need to unify all our Data Models, SQL scripts, and databases into one main PostgreSQL instance as we do not want multiple containers running PostgreSQL.

   (b) Unit Testing: To prevent developer errors from being pushed to the Cloud, Software Testing is important. Various tools are available that help with automated testing. These tools can also be integrated into a CI/CD pipeline, thus enabling a complete development life-cycle. For improved testing, the concept of Code Coverage should also be introduced, thus lowering software bugs. For UI based testing, tools such as selenium and Browserstack are popular. For server testing, PyTest for Python can help. For JavaScript, Supertest and Jest are popular.

4. User Interfaces

   (a) A System Health Monitoring Dashboard: Another suggestion is to have a monitoring system for easy viewing and analysis of the various containers, their health, and activity, along with tracking the logs of the various API calls made between the system parts.

5. Migrations and Public Cloud Usage

   (a) Migrations: What if this system needs to be migrated to another Kubernetes instance? Thankfully, the use of Docker has simplified migrations as we no longer have to worry about dependencies or architectures (except arm/arm64). To add to the ease of migration, each Deployment in Kubernetes has its own yml file which can be run to spin up an identical Deployment on any Kubernetes instance. The only major hurdle is Ceph and Camelot. Our current setup is very tightly coupled with Ceph and Camelot. Camelot contains all 500,000 ETDs and Ceph contains Team 1's PostgreSQL data and Team 2's Elasticsearch data.

   (b) Public Cloud Usage: Migrating this system to a public cloud like Amazon Web Services, Microsoft Azure, or Google Cloud has a fair number of advantages. Firstly, there would be access to better compute and storage. Some of the configurations available include: Multi GPUs nodes for workflows which needs multiple GPUs, Multi Core Nodes for workflows that require immense computing power, and High Memory Nodes for memory intensive workflows. See for example, Amazon EC2 Instance Types(https://aws.amazon.com/ec2/

instance-types/). For Deep Learning services that utilize TensorFlow, Google provides TPUs (Tensor Processing Units). This involves Machine Learning based Hardware Accelerator Application-Specific Integrated Circuit (ASIC) technology, which provides much better performance than GPUs for specific AI workflows.

Bibliography

[1] The Apache Software Foundation 2022. About Apache Airflow. [Accessed on 07-Sep-2022]. URL: https://airflow.apache.org.

[2] Apache Airflow. Architecture Overview — Airflow Documentation, 2022. [Accessed on 07-Sep-2022]. URL: https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html.

[3] Ceph authors and contributors. What is Ceph?, 2016. [Accessed on 08-Sep-2022]. URL: https://docs.ceph.com/en/latest/start/intro/.

[4] Hugh Beyer. User Centered Agile Methods, 2022. [Accessed on 10-Sep-2022]. URL: https://www.morganclaypool.com/doi/abs/10.2200/S00286ED1V01Y201002HCI010.

[5] TrustRadius Blog. Understanding CaaS, IaaS and PaaS, 2022. [Accessed on 08-Sep-2022]. URL: https://www.trustradius.com/buyer-blog/do-you-need-iaas-caas-paas-or-faas.

[6] Michael Bose. Kubernetes vs Docker – What Is the Difference?, May 2019. [Accessed on 01-Sep-2022]. URL: https://www.nakivo.com/blog/docker-vs-kubernetes/.

[7] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format, Dec 2017. [Accessed on 12-Sep-2022]. URL: https://datatracker.ietf.org/doc/html/rfc8259.

[8] GitLab B.V. What is Version Control?, 2022. [Accessed on 12-Sep-2022]. URL: https://about.gitlab.com/topics/version-control/.

[9] GitLab B.V. Why Gitlab? - The One DevOps Platform, 2022. [Accessed on 12-Sep-2022]. URL: https://about.gitlab.com/why-gitlab/.

[10] CNCF. CNCF - Kubernetes, 2022. [Accessed on 13-Sep-2022]. URL: https://www.cncf.io/projects/kubernetes/.

[11] Wikipedia Contributors. Directed acyclic graph, 2022. [Accessed on 20-Sep-2022]. URL: https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[12] NVIDIA Corporation. Hardware pre-requisites for running the NGC published PyTorch Docker image, Nov 2022. [Accessed on 22-Sep-2022]. URL: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch.

[13] Virginia Tech Department of Computer Science. VT CS Cloud Wiki, Feb 2022. [Accessed on 29-Aug-2022]. URL: https://wiki.cs.vt.edu/wiki/Cloud_Quickstart.

[14] Docker. Docker Hub Jupyter Notebook, 2022. [Accessed on 31-Aug-2022]. URL: https://hub.docker.com/r/jupyter/scipy-notebook.

[15] Docker. Dockerfile, 2022. [Accessed on 31-Aug-2022]. URL: https://docs.docker.com/engine/reference/builder/.

[16] Docker. PostgreSQL image for Docker, 2022. [Accessed on 01-Sep-2022]. URL: https://hub.docker.com/_/postgres.

[17] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud Narendra, and Bo Hu. Everything as a Service on the Cloud: Origins, Current and Future Trends. pages 621–628, 06 2015. [Accessed on 02-Oct-2022]. doi:10.1109/CLOUD.2015.88.

[18] GitLab. Gitlab runner. [Accessed on 19-Oct-2022]. URL: https://docs.gitlab.com/runner/.

[19] GitLab. Currently supported executors offered by the GitLab Runner, 2022. [Accessed on 01-Oct-2022]. URL: https://docs.gitlab.com/runner/executors/#selecting-the-executor.

[20] Google. Container Registry documentation, 2022. [Accessed on 25-Aug-2022]. URL: https://cloud.google.com/container-registry/docs.

[21] The PostgreSQL Global Development Group. What is PostgreSQL?, 2022. [Accessed on 01-Sep-2022]. URL: https://www.postgresql.org/about/.

[22] Alexander Hicks, Mohit Thazhath, Suraj Gupta, Xingyu Long, Cherie Poland, Hsinhan Hsieh, and Yash Mahajan. Integration and Implementation (INT) CS 5604 F2020 team term project report, Virginia Tech, Dec 2020. [Accessed on 25-Aug-2022]. URL: http://hdl.handle.net/10919/101544.

[23] Amazon Web Services Inc. What is an API?, 2022. [Accessed on 01-Sep-2022]. URL: https://aws.amazon.com/what-is/api/.

[24] Discord Inc. What is Discord?, 2022. [Accessed on 22-Aug-2022]. URL: https://discord.com/servers.

[25] Oracle Inc. Oracle DB, 2022. [Accessed on 29-Aug-2022]. URL: https://www.oracle.com/database/.

[26] Zoom Video Communications Inc. Zoom, 2022. [Accessed on 22-Aug-2022]. URL: https://explore.zoom.us/en/about/.

[27] 2022 INT. Use Rancher to add Kubernetes Ingress. [Accessed on 01-Nov-2022]. URL: https://drive.google.com/file/d/10lLHnHWlW6P7cLXLeGS_LsMJFqx5zTzG/view.

[28] IPython. How IPython works. [Accessed on 12-Dec-2022]. URL: https://ipython.org/ipython-doc/3/development/how_ipython_works.html.

[29] Alex Chumbley Karleigh Moore and Jimin Khim. Context free grammars, 2022. [Accessed on 02-Dec-2022]. URL: https://brilliant.org/wiki/context-free-grammars/.

[30] Jeff Sutherland Ken Schwaber. Scrum Definition, 2020. [Accessed on 22-Aug-2022]. URL: https://scrumguides.org/scrum-guide.html#scrum-definition.

[31] Grafana Labs. Grafana, 2022. [Accessed on 04-Oct-2022]. URL: https://grafana.com/.

[32] Microsoft. Microsoft SQL, 2022. [Accessed on 29-Aug-2022]. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[33] Gonzalo Camarillo Miguel A. Garcia-Martin. What is XML?, Oct 2017. [Accessed on 12-Sep-2022]. URL: https://datatracker.ietf.org/doc/html/rfc5364.

[34] MySQL. MySQL, 2022. [Accessed on 29-Aug-2022]. URL: https://www.mysql.com.

[35] Virginia Tech Department of Computer Science. Computer Science Cloud at VT, 2022. [Accessed on 22-Aug-2022]. URL: cloud.cs.vt.edu.

[36] Stack Overflow. Stack Overflow 2021 Developer Survey, 2021. [Accessed on 02-Oct-2022]. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-databases.

[37] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019. doi:10.1109/TCC.2017.2702586.

[38] Postman, Inc. About Postman, 2022. [Accessed on 18-Oct-2022]. URL: https://www.postman.com/company/about-postman/.

[39] Real Python. Jupyter Notebook: An Introduction, 2022. [Accessed on 03-Sep-2022]. URL: https://realpython.com/jupyter-notebook-introduction/.

[40] Rancher. Rancher, 2022. [Accessed on 04-Sep-2022]. URL: https://rancher.com/.

[41] Redhat. What is CaaS?, 2020. [Accessed on 25-Aug-2022]. URL: https://www.redhat.com/en/topics/cloud-computing/what-is-caas.

[42] RedHat. What is CI/CD?, May 2022. [Accessed on 25-Aug-2022]. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[43] Bit Pattern Revision. Camelot introduction, 2020. [Accessed on 08-Sep-2022]. URL: https://camelot-project.readthedocs.io/en/latest/introduction.html.

[44] Amazon Web Services. What is Elasticsearch? – Amazon Web Services, 2022. [Accessed on 05-Sep-2022]. URL: https://aws.amazon.com/opensearch-service/the-elk-stack/what-is-elasticsearch/.

[45] Stephen Shankland. Google spotlights data center inner workings, 2022. [Accessed on 12-Dec-2022]. URL: https://www.cnet.com/culture/google-spotlights-data-center-inner-workings/.

[46] Synopsys. CICD, 2022. [Accessed on 25-Aug-2022]. URL: https://www.synopsys.com/glossary/what-is-cicd.html#F.

[47] PostgreSQL Team. What is PostgreSQL?, 2022. [Accessed on 05-Sep-2022]. URL: https://www.postgresqltutorial.com/postgresql-getting-started/what-is-postgresql/.

[48] Wikipedia. ACID compliant databases, Dec 2022. [Accessed on 10-Sep-2022]. URL: https://en.wikipedia.org/wiki/ACID.