Final Report
CS 5604: Information Storage and Retrieval

Integration and Implementation (INT) Team:

Alex Hicks, Cherie Poland, Suraj Gupta,
Xingyu Long, Yash Mahajan, Mohit Thazhath, Hsinhan Hsieh

December 18, 2020

Instructed by Professor Edward A. Fox

Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

The first major goal of this project is to build a state-of-the-art information storage, retrieval, and analysis system that utilizes the latest technology and industry methods. This system is leveraged to accomplish another major goal, supporting modern search and browse capabilities for a large collection of tweets from the Twitter social media platform, web pages, and electronic theses and dissertations (ETDs).

The backbone of the information system is a Docker container cluster running with Rancher and Kubernetes. Information retrieval and visualization is accomplished with containers in a pipelined fashion, whether in the cluster or on virtual machines, for Elasticsearch and Kibana, respectively. In addition to traditional searching and browsing, the system supports full-text and metadata searching. Search results include facets as a modern means of browsing among related documents. The system supports text analysis and machine learning to reveal new properties of collection data. These new properties assist in the generation of available facets. Recommendations are also presented with search results based on associations among documents and with logged user activity.

The information system is co-designed by five teams of Virginia Tech graduate students, all members of the same computer science class, CS 5604. Although the project is an academic exercise, it is the practice of the teams to work and interact as though they are groups within a company developing a product.

The teams on this project include three collection management groups – Electronic Theses and Dissertations (ETD), Tweets (TWT), and Web-Pages (WP) – as well as the Front-end (FE) group and the Integration (INT) group to help provide the overarching structure for the application.

This submission focuses on the work of the Integration (INT) team, which creates and administers Docker containers for each team in addition to administering the cluster infrastructure. Each container is a customized application environment that is specific to the needs of the corresponding team. Each team will have several of these containers set up in a pipeline formation to allow scaling and extension of the current system. The INT team also contributes to a cross-team effort for exploring the use of Elasticsearch and its internally associated database. The INT team administers the integration of the Ceph data storage system into the CS Department Cloud and provides support for interactions between containers and the Ceph filesystem. During formative stages of development, the INT team also has a role in guiding team evaluations of prospective container components and workflows.

The INT team is responsible for the overall project architecture and facilitating the tools and tutorials that assist the other teams in deploying containers in a development environment according to mutual specifications agreed upon with each team. The INT team maintains the status of the Kubernetes cluster, deploying new containers and pods as needed by the collection management teams as they expand their workflows. This team is responsible for utilizing a continuous integration process to update existing containers. During the development stage the INT team collaborates specifically with the collection management teams to create the pipeline for the ingestion and processing of new collection documents, crossing services between those teams as needed. The INT team develops a reasoner engine to construct workflows with information goal as input, which are then programmatically authored, scheduled, and monitored using Apache Airflow.

The INT team is responsible for the flow, management, and logging of system performance data and making any adjustments necessary based on the analysis of testing results. The INT team has established a Gitlab repository for archival code related to the entire project and

has provided the other groups with the documentation to deposit their code in the repository. This repository will be expanded using Gitlab CI in order to provide continuous integration and testing once it is available.

Finally, the INT team will provide a production distribution that includes all embedded Docker containers and sub-embedded Git source code repositories. The INT team will archive this distribution on the Virginia Tech Docker Container Registry and deploy it on the Virginia Tech CS Cloud.

The INT-2020 team owes a sincere debt of gratitude to the work of the INT-2019 team. This is a very large undertaking and the wrangling of all of the products and processes would not have been possible without their guidance in both direct and written form. We have relied heavily on the foundation they and their predecessors have provided for us. We continue their work with systematic improvements, but also want to acknowledge their efforts Ibid. Without them, our progress to date would not have been possible.

Contents

List of Tables **6**

List of Figures **8**

1 Overview **9**
1.1 Project Management . . . 9
1.2 Problems and Challenges . . . 9
1.3 Solutions Developed . . . 11
1.4 Future Work . . . 11

2 Literature Review **13**

3 Requirements **14**
3.1 Overall Project Requirements . . . 14
3.2 INT Team Requirements . . . 14

4 Design **16**
4.1 Elasticsearch . . . 16
4.2 Docker Containers . . . 16
4.3 Kubernetes, Rancher, kubectl, and Containers . . . 18
4.4 Gitlab . . . 19
4.5 Ceph . . . 19
4.6 NFS . . . 20
4.7 Apache Airflow . . . 20
4.8 Virtual Machines . . . 20
4.9 Reasoner Engine . . . 21
4.10 Service API . . . 22
4.11 Apache Kafka . . . 22
4.12 System Architecture . . . 22

5 Implementation **24**
5.1 Timeline . . . 24
5.2 Milestones and Deliverables . . . 26

6 User Manual **29**
6.1 Rancher UI: Deploying Containers and Accessing Persistent Storage . . . 29
6.2 Elasticsearch . . . 30
6.3 Service Registry REST API . . . 34
6.4 Interacting with workflow related REST APIs . . . 35
6.4.1 Reasoner API . . . 35
6.4.2 Airflow API . . . 37
6.4.3 Modifying goals using user inputs . . . 39
6.5 Containerizing a Service for Airflow . . . 40
6.5.1 Creating a Dockerfile . . . 40

6.5.2 Pushing the Service to a Container Registry 40
6.5.3 Using Input/Output Environment Variables 41
6.6 CI/CD with Gitlab 41

7 Developer Manual **46**
7.1 Providing Access to Ceph Persistent Storage from KGI and ELS VMs 46
7.2 Kubectl Installation and Introduction 47
7.3 Deploying Containers from Rancher Catalogs 48
7.4 Deploying Containers from Docker Hub 48
7.5 Configuring and Deploying Containers through Gitlab Container Registry 50
7.6 Mounting Ceph storage on containers 53
7.7 Running Airflow on the Kubernetes cluster 58
7.7.1 Setting up the container 58
7.7.2 Issues faced 60
7.8 Running Postgres on the Kubernetes cluster 61
7.8.1 Setting up the container 61
7.9 Accessing Elasticsearch through the Elasticsearch VM 61
7.9.1 Issues faced 62
7.10 Enabling pod to pod communication within a namespace 64
7.11 Installing Gitlab-Runner 64
7.12 Installing Elasticsearch 65
7.13 Service Registry Database 67

Bibliography **73**

A Manual Service Registry Artifacts **74**

List of Tables

1 Tasks and Timeline 24
2 Milestones 26
3 Deliverables 27

List of Figures

1 Representative configuration of Elasticsearch 17
2 The difference between a container and virtual machine. [10] 17
3 Container administration hierarchy 19
4 The components of Apache Airflow 21
5 Kafka topics (blue) and producers and consumers 2019 23
6 System diagram for information retrieval and analysis system 23
7 CS cloud cluster 29
8 Pods and namespaces 30
9 Container running entrance 31
10 Elasticsearch VM Confirmational Information 31
11 Elasticsearch VM running Ubuntu 32
12 ES v7.9.2 in a 3-node cluster in a Docker Container 32
13 Example Ingestion Schema of Elasticsearch VM 32
14 Ways to Interact with Elasticsearch 33
15 Indexing with Elasticsearch 34
16 Indexing on Elasticsearch 35
17 Index Mapping 36
18 Index Shard Response 37
19 Posting to the Service API 37
20 Posting to the Goal API 37
21 Frontend interaction with Airflow 38
22 Response of the /generateWorkflow API endpoint 39
23 Airflow API usage example 40
24 An example Dockerfile for a service 41
25 Example 1: .gitlab-ci.yml file 43
26 Example 2: .gitlab-ci.yml file 44
27 Gitlab CI/CD stages 45
28 Terminal view of the tasks executing in the build stage as seen through the Gitlab CI/CD interface 45
29 Kubectl configuration 48
30 Kubeconfig File 49
31 List of pods in the namespace `cs5604-fe-db` 50
32 Navigating to CS5604-INT namespace 50
33 Launching a Rancher App 50
34 Launching MySQL App 51
35 Configuring the App 51
36 Testing if MySQL App has been launched successfully 52
37 Search for container on Docker Hub 52
38 Search container on Docker Hub 53
39 Gitlab Container Registry 53
40 Filling deploy token details 54
41 Deploy Token Username and Password 54
42 Registry Credentials on Rancher 55

43 Adding deploy token details 55
44 Deploying container from Gitlab Container Registry 56
45 Deployed Container "centostest" present under the namespace 56
46 Configure Docker image details on deploy portal 57
47 Select Volumes and Add Volume to mount storage on container 57
48 Select 'Ceph Filesystem' as the Source on Ephemeral Volume portal' 58
49 Enter Configuration details for the Ceph Filesystem on Ephemeral Volume portal . 59
50 Snapshot of a pod under a namespace and "View/Edit YAML" button 59
51 Adding Ceph Secret on Rancher 60
52 VPN Connection 62
53 SSH through Putty 63
54 VM Directory for Elasticsearch 63
55 VM Directory for Elasticsearch 64
56 Creating a service name for pod to pod communication 65
57 Connecting the CI/CD system 66
58 Adding a Gitlab Runner to Gitlab 67
59 Schema of various tables used for registering services 69
60 WP Service Metadata 74
61 WP Goal Metadata 74
62 WP Reasoner Metadata 74
63 ETD Service Metadata 75
64 ETD Goal Metadata 75
65 ETD Reasoner Metadata 76
66 TWT Service Metadata 77
67 TWT Goal Metadata 77
68 TWT Reasoner Metadata 78

1 Overview

1.1 Project Management

There are seven members in our team, so team management is crucial to accomplish the project goal. We currently meet twice a week during class sessions using the provided Zoom meetings [5] to discuss tasks and issues and to plan our next steps to ensure progress is being made towards the course goals. Because the INT team's role is to ensure an operating portable container environment by integrating all of the teams' efforts, we communicate with the other teams in class sessions and on designated Discord [37] channels (e.g., #team-int) to discuss system design requirements and resolve integration and implementation issues that hinder the overall project progress.

As mentioned above, we have adopted several tools to achieve efficient collaboration. We use Discord and Zoom for team communication, Ally.io [53] for task management, Gitlab [29] for code collaboration, as well as Docker Hub [22] and the Virginia Tech Container Registry [52] for container distribution.

1.2 Problems and Challenges

We have faced a few challenges including:

a. Acquiring specific container requirements from the teams at an early stage of development.

b. We have been informed that last year's team experienced challenges in transferring large processed datasets into Ceph File System storage. We anticipate utilizing the methodology outlined in the Ceph Tutorial to overcome these potential problems. §7.1.

c. We have been informed that last year's team experienced challenges related to the container's shell on the Computer Science (CS) cloud including time-outs and early deletions. Solutions for these prior challenges were provided by last year's team and are outlined in the kubectl tutorial. §4.3.

d. We have been informed that last year's team experienced challenges related to the inability to save and package a modified container as a new container image by using kubectl or Rancher (see §2 and §4.3). Tools and dependencies installed after deploying the container will not become part of the container image. This is because containers are ephemeral (short lived) and Kubernetes, the underlying container orchestration for Rancher, does not allow committing changes to a container. The solution is to update the container's Dockerfile [20] rather than the running container itself.

e. We have also been informed that last year's team experienced challenges related to the fact that some of the Docker container images may be very large. We anticipate that this will slow deployment and the scaling of the services that these containers provide. A solution to this problem is under development. One possible solution is to shrink the size of Docker images by using tools such as Docker-Slim [3] or by creating very efficient Dockerfiles relying on Docker best practices and caching mechanisms. In order to focus on the second solution, teams will iterate development of their containers by selecting the correct base images, installing only what is necessary, and removing any unused or unneeded packages and libraries.

f. Another challenge was installing Elasticsearch on the dedicated VM. This semester, we chose to install Elasticsearch on a virtual machine instead of to the cluster on the recommendation of the previous INT team. To this end, a CentOS 8 VM was provisioned by the CS Techstaff, but we were unable to install a Dockerized version of Elasticsearch to this VM. We assume this issue came from system configuration and security systems such as SELinux, but do not know the root cause of the issue. In order to address this issue, we requested that the VM be reprovisioned as an Ubuntu VM, which we independently tested with the Docker Elasticsearch stack and verified that it would be a much more suitable host for the stack. We then installed Elasticsearch with help from the documentation to configure the correct environment [14].

g. We also ran into issues determining where the raw data for the collections would be stored. Two of the content teams had their data located in the Ceph storage where it was easily accessible. The third team received their data in Google Drive, which was not a productive location to access it for indexing into Elasticsearch. To solve this issue, we worked with SMEs, other researchers working with this data, and CS Techstaff in order to move all the raw data into an NFS share that is mounted in all of the Kuberentes namespaces for easy access by the content teams.

h. In order to run our Continuous Integration system, we needed a Gitlab Runner that was configured with a Docker executor to run the builds of our containers for deployment to the cluster [32]. This system requires a deployment outside of the cluster in order to provide support, so this needed to be installed on an external VM. We gained access to a VM provisioned for this project (kgi) and installed the Gitlab Runner and registered it with our Gitlab instance (git.cs.vt.edu).

g. We faced a lot of issues deploying pods through Airflow and running Airflow's scheduler on the cluster. More about it can be read in Section 7.7.2.

h. We faced issues in building and running Docker containers through .gitlab-ci.yml file for CI/CD. We had set up our Gitlab Runner using a specific configuration file and we did not realize that it was mounted incorrectly. As such, any changes we made to reconfigure the runner were not reflected in the running instance. In order to solve this problem, we restructured how we attached the configuration file to the runner.

i. Once we installed Elasticsearch on the virtual machine and made it accessible for the content teams, we encountered several issues. First, we realized that we will need to enable the security features on Elasticsearch and make sure access is locked down. Currently, everything is located within the Virginia Tech system, but as we develop the application further, security should be a top concern. Along with that point, we had to make sure that we were allowing the correct access to the FE team to provide to the outside world. They currently have access, but again, that should be secured further to avoid unnecessary exposure of the data. Then, we experienced some data loss from the Elasticsearch database. We determined the cause of our Elasticsearch problems, after the database was found empty of indices and a GDPR-based ransomware notice was found. As a result, we deleted the entire operating system and started over with a clean VM and reinstalled Elasticsearch. We believe that the source of the vulnerability was the fact the the Elasticsearch stack automatically exposes itself to the open internet, overriding any existing firewall rules, thus our previous idea that the installation was safe enough in the short term was incorrect. In order to solve this problem, we deleted and reimaged the virtual machine that was running Elasticsearch and on the second install, made sure our stack was using SSL and a secure username and password that was safely distributed to each of the other teams. Along with these updates, we decided Kibana was no longer necessary, since we were not planning to keep it long term anyway and it would have required additional time to set up securely.

j. We encountered an issue with CI/CD where our builds were failing due to rate limits on Dockerhub. We found that this was due to Docker adding rate limits to their free tier accounts. In order to solve this issue, we discussed it with Mike Irwin and Chris Arnold and they recommended we update the pull_policy of our Gitlab Runner to avoid making as many requests. They also suggested, if that does not completely solve the problem, to mirror the images we are pulling from Dockerhub somewhere local, whether it be on the VM or using one of Virginia Tech's Docker registries.

1.3 Solutions Developed

Throughout the project, we faced several challenges and correspondingly solutions were developed to address those challenges. We reached out to Michael Irwin, a Docker Captain here at Virginia Tech for advice developing our containers as well as for recommendations for technologies to fulfill some tasks, including message passing using Kafka. He in turn put us in contact with Brent Kremer, who is a technical manager and product owner for the Virginia Tech data lake. A third resource we have been put in contact with is Chris Arnold, a member of the Computer Science Technical Staff at Virginia Tech. His efforts were integral to the previous INT team's efforts and we leveraged his expertise as a resource when necessary. A fourth resource we have been put in contact with is Rob Hunter, a member of the Computer Science Technical Staff at Virginia Tech. His expertise was critical for us in setting up the virtual machines securely. Additionally, we had the benefit of learning from the previous team's decisions and were able to plan ahead on some of the issues that they encountered. Specifically, this helped us build our containers using Dockerfiles from the ground up as well as building on their research into which technologies they selected for various tasks including a continuous integration and continuous deployment (CI/CD) pipeline. As discussed above in the Problems and Challenges section, we ran into several issues getting Elasticsearch and the Gitlab Runner set up as well as where the raw data was located. Unfortunately, the solutions for these problems required stepping outside of the cluster into an NFS share and two virtual machines. This did complicate our architecture design, but necessarily so, and it has been reformatted and communicated to the rest of the teams and course staff as further issues have arisen.

1.4 Future Work

The INT team has been working with each of the content teams to get their services working within the cluster. Despite reaching this point, we still have yet to finish the service API to automate service registration and provide it to the end user from the front end. Additionally, the Elasticsearch system is secure, but it is currently using self-signed certificates which are causing issues with the Front End team's workflow, and should be upgraded to correctly signed certificates from Virginia Tech. Also relating to Elasticsearch, we could add additional accounts for different team's services in order to further secure the system and avoid issues of automated services touching the wrong data. We should finish solving the Docker rate limit problem with Gitlab CI by mirroring the images we need to somewhere local to Virginia Tech's network. Airflow currently supports only inputs in the form of files; in the future we expect it to be able to read inputs from Elasticsearch as well. A generalized service can be created for achieving this. The developed service can then be prepended to appropriate workflows.

Another idea for future work would be to connect the service API to the reasoner table in the database. Currently, the API creates the correct entries in the goals and service tables, but it doesn't populate the reasoner table because that requires the additional user input of which goals are required as input for the specific goal. This would require either an additional API or a significant modification to the current API to access another table in the database.

2 Literature Review

Containers [42, 51] are replacing virtual machines and are being adopted to modernize applications. Containers are favored for their lightweight, isolation, and portability features [43]. Such features are possible by leveraging Linux primitives such as cgroups and namespaces [50]. The container market will be a \$4.3 billion market by 2022 [6]. Docker, the leading container management platform [19], is the main agent that is driving the adoption of containers [43]. While Docker remains current state of the art, several other container runtimes, including Podman [4] by Red Hat, provide stable alternatives. The development and success of Podman has led to a lack of support by Red Hat and CentOS for Docker [34]. The fact remains, however, that Docker is the best suited container runtime due to its longer history and better integration with the current systems as discussed below. This may change going forward as the Docker runtime will no longer be supported on Kubernetes as of late 2021, but the Docker build system will still be supported [39].

For this project, we are leveraging the CS cloud [2]. The CS cloud deploys Rancher [45] that provides services on top of Kubernetes [33] which provides the perfect infrastructure for us to deploy our multi-container information retrieval system. Rancher is an open source platform that provides a user interface for a Kubernetes cluster and handles the management and facilitates the monitoring of Docker [21] container clusters. Rancher provides a simple workload environment, a centralized control plane, and enterprise-grade technical support.

Kubernetes [33], an open-source cluster manager from Google, has become the leading platform for powering modern cloud-native containerized micro-services in recent years. Kubernetes is a Greek word meaning helmsman of a ship or pilot. This naming has continued the container metaphor used by Docker. Its popularity is driven by the many benefits it provides, one of which is the ease of install on a small test bed (as small as one virtual machine or physical server). However, running Kubernetes at scale with production workloads requires more resources in addition to more thought and effort [44]. 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. Pods, a Kubernetes abstraction, host an application instance. Each pod represents one or more containers and some shared resource for those containers such as a network.

The ELK Stack consists of three services, Elasticsearch, Logstash, and Kibana [14, 13, 12, 16]. Elasticsearch is a distributed, open source search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. It is built on Apache Lucene and uses a simple REST API that is distributed, fast, and scaleable. Elasticsearch is the central component of the Elastic Stack, a set of open source tools for data ingestion, enrichment, storage, analysis, and visualization. The Elastic Stack also includes Beats, a collection of lightweight shipping agents for sending data to Elasticsearch. The ingested documents are pre-processed to extract both text data and metadata. Metadata of the ingested data is indexed by Elasticsearch and stored in Ceph. Kibana interacts with Elasticsearch to provide a framework for initial data analysis. The Logstash portion of the stack analyzes user logs to provide efficient recommendations.

3 Requirements

3.1 Overall Project Requirements

Since the goal of the project is to build an Information Storage and Retrieval System, the overall project requirements with respect to the above goal can be noted as:

- The unit of processing should be of either an entire document, or an additional document that is derived from an original document, such as by segmentation/extraction.
- Searching should be facilitated to support both full-text and the metadata of a document.
- Searching and browsing should be supported based on facets connected with the data or metadata.
- Searching and browsing should be supported based on facets associated with information derived from documents, through analysis, classification, clustering, summarizing, or other processing.
- Logs should be collected, of user queries and clicks, and analyzed to support users. Recommendations should be identified and made available to users.
- Selection of techniques, including indexing and weighting, should ensure that operations are effective.
- Ranking of search results should be based on the most effective method.
- Pre-processing should be tailored to the content collection, to handle page images (i.e., a suitable method of OCRing, as would be needed if an ETD has only page images, rather than text) and to manage linguistic issues (e.g., stemming or part-of-speech disambiguation or phrase identification or entity recognition).
- Data and software produced must be released to the project for further refinement and utilization. Doing so would benefit from students working with https://git.cs.vt.edu/ (VT Computer Science Gitlab).

3.2 INT Team Requirements

Our team is responsible for the integration and implementation of all the teams' efforts. This includes:

- Designing and deploying customized Docker containers for each content team.
- Managing the Kubernetes cluster on the CS Cloud via Rancher and kubectl.
- Connecting containers to Ceph storage for each team for data retrieval and storage.
- Developing a Reasoner Engine and Workflow Engine to take requests, generate workflow, execute workflows, and send responses back.

- Coordinating with Operations on associated VMs and external storage.
- Evaluating and testing the cluster components at various stages of development.
- Developing a CI/CD Pipeline to test and automate deployment of the application.

4 Design

The class project development and production phases will leverage the CS cloud infrastructure that is running Rancher which is based on the Kubernetes container management platform. Since all of the class teams will be initially working on the *testing* cluster on the CS cloud, we have created projects, one per team, to provide a level of organization in the *testing* cluster. The five projects are CS5604-ETD, CS5604-TWT, CS5604-WP, CS5604-FE, and CS5604-INT. Under each project, we are planning to deploy a collection of containers, connected in order to form a workflow to ingest and index the various forms of data this project is investigating. One of these containers will be a CentOS container that is connected to a Ceph storage virtual machine running Ceph File System, that will be mounted on the CS Cloud cluster for access by the teams.

4.1 Elasticsearch

Elasticsearch is a distributed, open source search and analytics engine for data ingestion, enrichment, storage, and analysis, for all types of data, including textual, numerical, geospatial, structured, and unstructured. Elasticsearch is built on Apache Lucene, released under an Apache license, and contains a simple RESTful API that is distributed, fast, and scaleable. It is Java-based and available for many platforms that can search the index documents files in diverse formats. Data is stored in a schema-less format in JSON documents, similar to NoSQL databases, and interfaces with Java APIs. The main use cases for Elasticsearch are scraping and combining public data, full-text search, event data and metrics, and logging and log analysis. It performs near-real-time searches, provides multi-tenancy support, automatically indexes JSON documents, and indexes using type-level identifiers.

An Elasticsearch cluster can contain multiple indexes which can contain multiple types (tables) each holding multiple documents (rows) and each document has properties (columns). Data are stored in indexes that are functionally separated into primary and replica shards. Elasticsearch supports multiple indices. Multiple indices can be created and separate indices may be used to store different types of data. Data may be downloaded using curl or via HTML through a browser.

Ingested documents are pre-processed to extract both text data and metadata. Metadata of the ingested data are indexed by Elasticsearch and stored in Ceph. Kibana will be available for parts of the development process as a visualization platform that interacts with Elasticsearch in order to provide a framework for initial data analysis. However, an API developed by the FE team will be utilized for the final accessible end-user product.

In this year's project each separate team will be responsible for developing their own index structure for their data.

Elasticsearch permits multiple queries at the same time. Different types of search queries may be developed by the different teams that best represent the different types of data being ingesting into Elasticsearch. Additionally, we may be integrating Apache Kafka into the workflow to avoid overwhelming Elasticsearch as discussed in Section 4.11

4.2 Docker Containers

Docker is a tool designed to make it easier to create, deploy, and maintain the applications by using different customized containers. It benefits both system administrators and software developers,

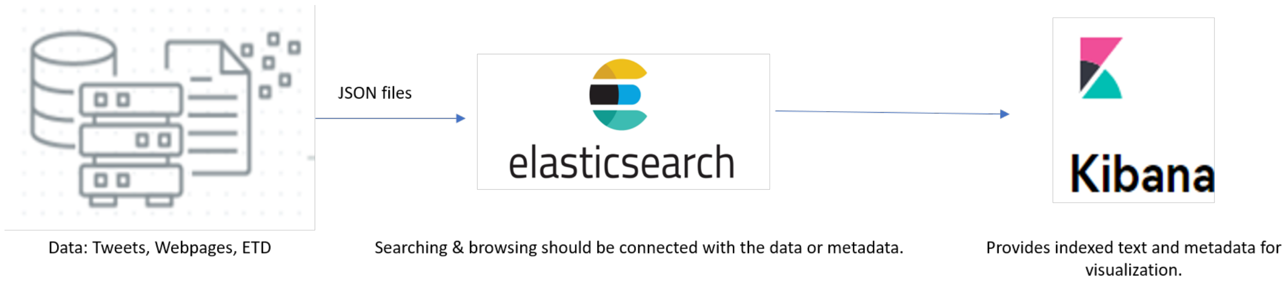

JSON files
elasticsearch
Kibana
Data: Tweets, Webpages, ETD
Searching & browsing should be connected with the data or metadata.
Provides indexed text and metadata for visualization.

Figure 1: Representative configuration of Elasticsearch

which is the crucial part of DevOps technologies. In addition, Docker containers are an OS-level virtualization whereby the operating system isolates the applications and limits its resources (see Figure 2).

Containers enable developers to run applications in a reliable and portal way by packaging code and all of its dependencies, so the application runs quickly and reliably. This facilitates fast and consistent application deployment regardless of the deployment environment. That has led to the widespread adoption of container technology. [38].

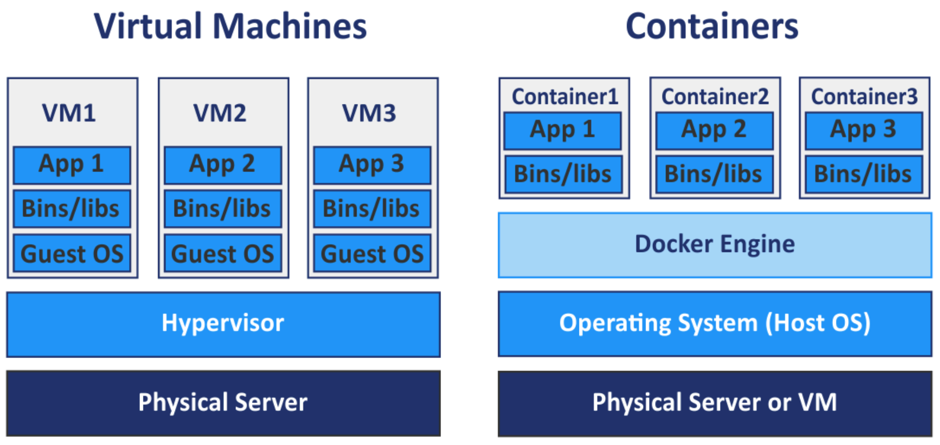


Figure 2: The difference between a container and virtual machine. [10]

Containers leverage Linux kernel features[1] – control groups (i.e., cgroups) and namespaces – to achieve the desired isolation and portability features [17]. Docker is currently the world's leading container management platform, followed by CoreOS rkt [35], Mesos [27], and LXC [18]. Besides, Docker simplifies the standard applications by helping with packing and running applications on the Docker platform.

Containers are created from customized images that are stored in a container registry both privately and publicly such as Docker Hub [22], Quay [49], and Google Container Registry [1]. Users

[1]Docker can run atop other operating systems by leveraging a Linux-based hypervisor.

can easily access different kinds of images which support various services and also can contribute their own images into the community by a few commands through your terminal. Virginia Tech also provides a container registry [52] that is hosted locally.

The INT team will create, deploy, and maintain containers based on requirements from each team. But, because of the portable nature of containers, all of the teams can deploy Docker containers on their PC and change the images according to their project needs and also can access the images by pulling and pushing commands. The simplicity of Rancher and kubectl allows us to work in the CS cloud cluster. At the end, containers from other teams can be integrated into one container cluster.

4.3 Kubernetes, Rancher, kubectl, and Containers

Large-scale applications often involve a number of services that live in their own containers and then are deployed on clusters or across multiple machines. As the traffic increases, more instances of the services can be signed up which may not scale linearly and can get difficult to manage. Hence, to address this issue an orchestration engine is employed, such as Kubernetes. Kubernetes (K8s) is an open-source system that automates the deployment, scaling, and management of container clusters. It is utilized as a higher-level abstraction and simplifies the management of container clusters by grouping the containers that form an application into logical units [33].

Rancher is an open-source platform that provides services on top of Kubernetes [45]. In addition to providing Kubernetes-as-a-Service, Rancher has the following capabilities:

- Rancher gives unified administration of clusters and the containers running in them. Since each team has its own project and namespace, operational and security challenges can be overseen proficiently.

- Rancher is resource-agnostic. It provides the ability to bring up clusters from different providers, migrate resources, and manage any kubernetes-based platform in both private and public clouds [46]. Rancher unifies the individual deployments as a single Kubernetes Cloud and presents them through a single front-end interface.

- Rancher provides easy authentication policies for different users and groups. Admins can enable self-administration by assigning the organization of Kubernetes clusters or projects and namespaces directly to individual users or groups [47]. This is particularly valuable since it permits us to control memberships, ownerships, permissions, and capabilities of different team projects.

Kubectl [8] is the Kubernetes command-line tool to control the Kubernetes cluster manager. Kubectl requires a file named 'config' in the $HOME/.kube that needs to be configured with the cluster credentials one needs to connect. Kubectl can be used to deploy and manage pods (made up of a container, or several closely related containers) in the Kubernetes cluster. It is also used to achieve CI/CD. Kubectl would be installed in each team member's local host so that one can easily execute necessary deployments directly to the cluster.

Figure 3 shows the relationship between Kubernetes, Rancher, kubectl, Kubernetes pods, Docker containers, and Kubernetes-managed physical nodes. Rancher generally provides a user interface and API for users to interface with the Kubernetes cluster. Kubectl on the other hand provides a command line interface to manage and work with the Kubernetes cluster. Kubernetes manages nodes (i.e., worker machines). A node may be a VM or a physical machine. Each node contains the services necessary to run Pods including the container runtime (i.e., Docker), kubelet, and kube-proxy [7]. Pods are the smallest deployment unit in Kubernetes which host the application instance. Each Pod represents a single container or multiple tightly coupled containers that share resources. The Pod is an environment in which containers run; it exists until the container is terminated and the pod is deleted [9].

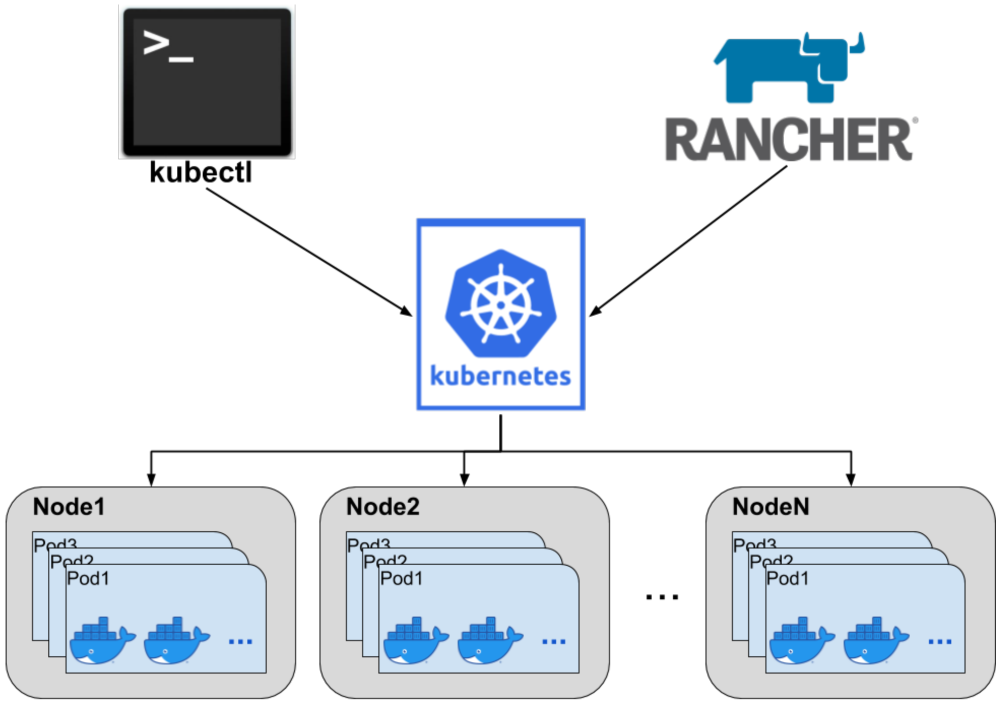


Figure 3: Container administration hierarchy

4.4 Gitlab

For this project, the teams are using Gitlab in order to store and version control their respective codebases. In order to maintain the security of any sensitive data analyzed in this project, the teams are using an internal, self hosted instance of Gitlab set up by the Virginia Tech Computer Science Department at `https://git.cs.vt.edu/`. Additionally, Gitlab provides a hosted and integrated Continuous Integration environment that interfaces with Kubernetes and allows the project to remain completely within the Virginia Tech network. In order to create the CI environment, a Gitlab Runner was installed on the kgi VM in order to provide the runner to build the Docker images for deployment to the Kubernetes cluster.

4.5 Ceph

In this project, the Ceph [58, 48] storage system is being used as our persistent shared storage that has been mounted on different components of our storage and retrieval system such as containers, ETDs VM, and an Elasticsearch VM. Ceph is an open source software set up to facilitate exceptionally adaptable *object*, *block*, and *file* based storage under one unified system. Ceph uses the Ceph Block device, a virtual disk that can be mounted to either physical Linux based servers or virtual machines (VMs). It is highly scalable, flexible, and reliable, and is widely used in modern data centers, and is supported by many cloud computing vendors. Ceph is being used to provide persistent storage for the ephemeral containers that will make up the workflow pipelines.

4.6 NFS

For this project, in addition to Ceph, we have created a large NFS share that will store the raw data that will be consumed by the content teams and ingested into Elasticsearch. As additional datasets are included, they will be stored here as well. This share will also be mounted on the Docker containers in order to give the workflow pipelines access to the raw data on which they run their workflows.

4.7 Apache Airflow

Apache Airflow is an open-source platform built using Python to schedule, manage, create, and run workflows [26]. NFS also supports archiving results, such as output, in addition to input capability. Airflow will be used to tie all services together through workflows. Each workflow will represent a set of services that can be run on the final system. The aim of a workflow is to take a certain input and produce a desired output. Workflows are idempotent, which means that the final output is always reproducible regardless of the number of times it is run. Workflows are represented as Directed Acyclic Graphs (DAGs). Each node in the graph represents a task to be performed in the workflow. The core components of Airflow are as follows (ref. Fig 4).

- Webserver - The webserver hosts an application which serves as a user interface (UI) for Airflow. The UI can be used to manage and run existing workflows, check the logs generated by tasks, and monitor the status of each task in a workflow.
- Metadata Database - The database stores the current state of tasks and can be accessed from the UI as well.
- Scheduler - The scheduler looks at a DAG, decides the tasks that need to be run, and stores the status of each task in the database. It also decides when each task needs to be run.
- Executor - The executor is the component that runs the scheduled tasks. Airflow supports execution locally, on distributed task queues (such as celery), or on a Kubernetes cluster. In our project, the Kubernetes executor will be used to run tasks. Using this helps Airflow deploy pods based on the workflow in execution.

4.8 Virtual Machines

This project makes use of two virtual machines, a knowledge graph virtual machine located at kgi.cs.vt.edu, and an Elasticsearch virtual machine located at elasticsearch.cs.vt.edu. The kgi VM

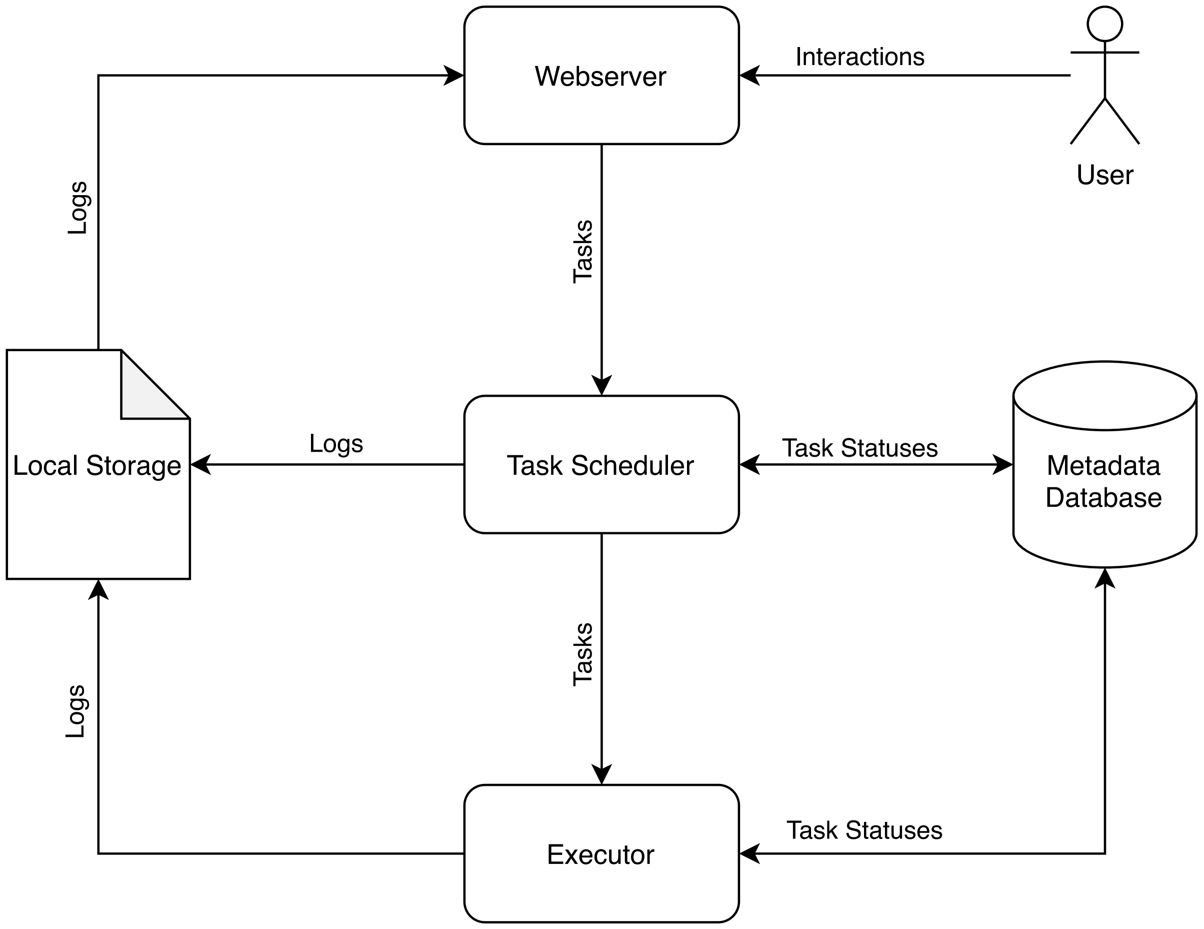


Figure 4: The components of Apache Airflow

is currently running our Gitlab Runner, and can be used by the FE team for running parts of the knowledge graph that are not suitable for the cluster. The Elasticsearch VM is currently running a Dockerized version of Elasticsearch for both the content teams and the FE team to access for indexing and searching the data, respectively. These two VMs both have Docker installed on them to run their respective applications.

4.9 Reasoner Engine

The reasoner engine will work closely with Airflow. The main purpose of the reasoner is to mine a set of services to be executed as a workflow. All of the services supported by the system will be stored in a MySQL database. This database can also be used to determine already existing services. Letting the users know about the existing services in the frontend helps them build new workflows utilising already existing services. Once a workflow is determined, it will be sent to Apache Airflow for execution.

4.10 Service API

In order to provide an interface to the Reasoner Engine discussed above, we are creating a service API that will interact with the same database as the Reasoner in order to create, update, delete, and get the workloads provided by the Reasoner (and Airflow by extension). The structure of this API consists of a Dockerized Flask application to be deployed to the Kubernetes cluster where it can connect with the database and the FE environment. This API will be exposed to the FE where some of the views will provide methods to call these workflows. One example of this is the service that indexes data into Elasticsearch. All of the services provided by the content teams can be found in Appendix A.

4.11 Apache Kafka

In this project, one of the ultimate goals is to continually expand three content collections. To handle the ingestion of new documents, we currently store the files directly into Ceph Storage after being processed, which works well for now. However, considering the future usage and scalability, we investigate Apache Kafka [40] for expansion.

Apache Kafka is a distributed, partitioned, replicated commit log service [28]. It is based on the publish-subscribe mechanism where producers write messages into a buffer while consumers read messages from it. Apache Kafka's architecture includes the following components:

- Records/Messages: contain key (optional), value, and timestamp;
- Topic: A category or feed name where messages are published and stored in. A topic is a log consisting of many partitions that are distributed in servers;
- Producer: Any object that can post/publish messages to any topic;
- Broker: A controller or server who receives the request and does the particular operation. For example, it: receives messages from producers and stores them on disk keyed by unique offset or allows consumers to fetch messages by topic, partition, and offset;
- Consumer: It can pull data from/subscribe to one or more topics, that are maintained by the Broker, to consume published messages.

For the case that a user specifies a very large queue of documents to be processed, Kafka will be able to scale and throttle both levels of queue processing such that system resources are not adversely impacted. Ultimately, as the number of collections grows, indexing of documents into Elasticsearch would be the bottleneck of the ingestion system. Queuing the documents through Kafka allows for the most efficient means of prioritizing and scheduling how new documents from all collections are to be ingested. Consumers can consume produced topics in real time.

4.12 System Architecture

Figure 6 shows the structure and the components of the information system that is designed in this project as described in this report.

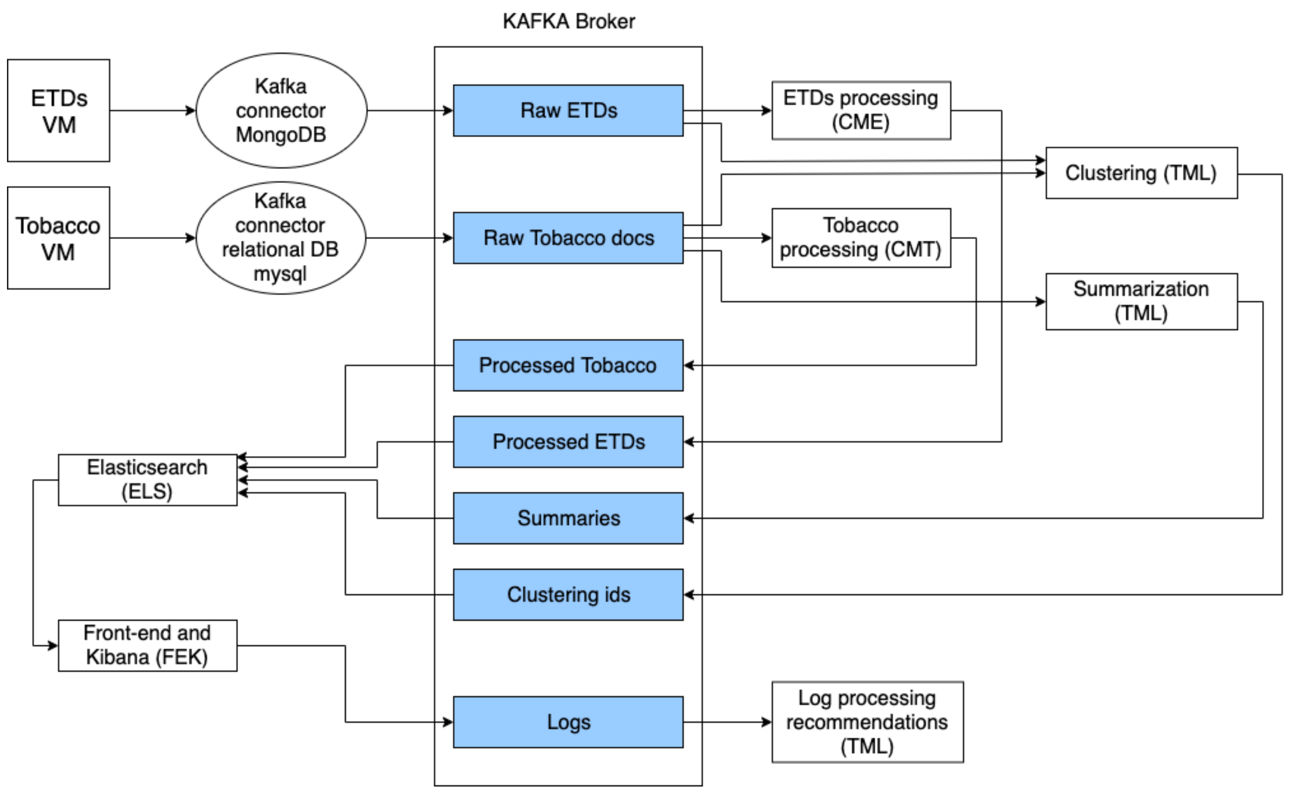


Figure 5: Kafka topics (blue) and producers and consumers 2019

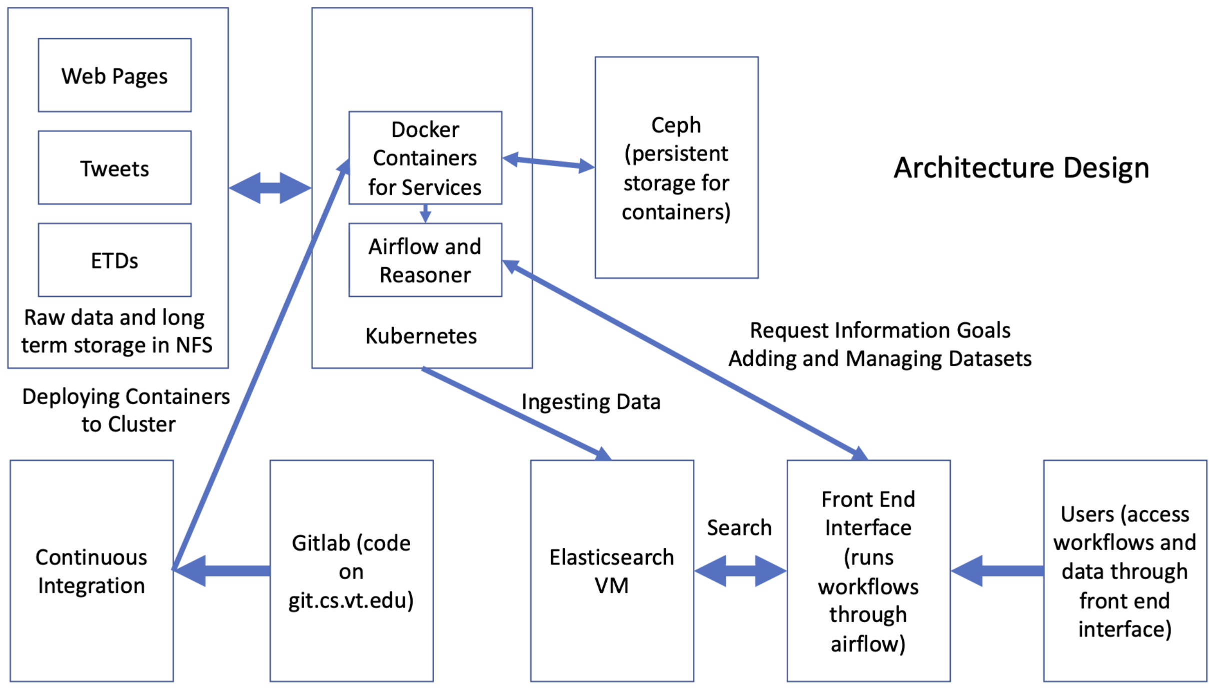

Web Pages
Tweets
ETDs
Raw data and long term storage in NFS
Docker Containers for Services
Airflow and Reasoner
Kubernetes
Ceph (persistent storage for containers)
Architecture Design
Request Information Goals
Adding and Managing Datasets
Deploying Containers to Cluster
Ingesting Data
Continuous Integration
Gitlab (code on git.cs.vt.edu)
Elasticsearch VM
Search
Front End Interface (runs workflows through airflow)
Users (access workflows and data through front end interface)

Figure 6: System diagram for information retrieval and analysis system

5 Implementation

5.1 Timeline

Table 1 shows our schedule. It contains the task description, our estimated timeline in weeks, team members responsible for accomplishing the task, and the current status. This schedule has been added to and changed over time.

Table 1: Tasks and Timeline

Task	Timeline (week)	Assignee	Status
CS Cloud project creation for each team	1	Alex	DONE
Student assignments as owners for their corresponding projects	1	Alex	DONE
Deployment of containers under each project	3	ALL	DONE
Ceph client installation on VMs (virtual machines) and mounting of CephFS for cloud storage access	3	ALL	DONE
Locally testing out Apache Airflow	3	Mohit and Yash	DONE
Setting up a container for Apache Airflow	4	Mohit	DONE
Creation of pods (containers) for the ELS and FE teams for data retrieval from Ceph	4	ALL	DONE
Testing of team container(s) in the CS container cluster	4	ALL	DONE
Aggregation of teams' containers into one testing cluster (the development namespace)	4	ALL	DONE
Creation of an Elasticsearch tutorial	4	Cherie	DONE
Creation of a Kubectl installation tutorial	4	Suraj	DONE
Creation of a tutorial for leveraging Rancher's Catalog and App to deploy ready-made containers	4	Suraj	DONE
Creation of a tutorial for leveraging Docker Hub to deploy ready-made containers	4	Suraj	DONE

Continued on next page

Table 1 – continued from previous page

Task	Timeline (week)	Assignee	Status
Creation of a tutorial to provide access to Ceph Persistent Storage from ETD and ELK VMs	4	ALL	In-Process
Setting up a database for registering services (Postgres)	5	Yash	DONE
Creation of Apache Airflow Developer Manual	5	Mohit	DONE
Creation of Postgres Developer Manual	5	Mohit	DONE
Assessment of additional container requirements for each team that came out of their initial discovery processes	5-7	ALL	DONE
Implementing and deploying the reasoner engine based on the workflows provided by all the other teams	6-8	Mohit and Yash	DONE
Creation of Flask API for Workflow Management	6-8	Alex	DONE
Integration of Elasticsearch API for Managing Datasets	6-8	Xingyu	DONE
Implementation of additional container requirements for each team's container	6-8	ALL	TBD
Prototype of integrating the reasoner engine and Airflow to determine a workflow from a set of workflows	6-8	Mohit and Yash	DONE
Creation of a User Manual for creating an Airflow compatible Docker container	6-8	Mohit	DONE
Creation of a User Manual for interacting with Airflow and Reasoner APIs	6-8	Mohit	DONE
Prototype of integrating the reasoner engine and Airflow to determine a workflow from a set of workflows	6-8	Mohit and Yash	DONE
Creation of a tutorial to deploy containers through a private repository	6-8	Suraj	DONE

Continued on next page

Table 1 – continued from previous page

Task	Timeline (week)	Assignee	Status
Creation of a User Manual to deploy Kafka and decide whether to use it	6-8	Hsinhan	Done
Container testing, evaluation, and integration into the CS cloud Kubernetes cluster	6-8	ALL	Partially Done
Development of a system for automatic/direct inclusion of future new data into our information system (time permitting)	7-8	ALL	TBD
Evaluation study of system performance (time permitting)	9-11	ALL	TBD
Integrating the reasoner and Airflow APIs, enable dynamic generation of Airflow documents	9-12	Mohit	DONE
Creation of a User Manual for CI/CD	9-11	Suraj	DONE
Helping content teams to integrate CI/CD platform	9-12	All	DONE
Setting up unit and integration tests	9-12	Xingyu	DONE
Manual Service Registration	13-15	Mohit	DONE

5.2 Milestones and Deliverables

Our milestones over time are shown in Table 2. We will provide deliverables as listed in Table 3.

Table 2: Milestones

Task #	Completion Date	Milestone
1	09/3	Setup of namespaces and team projects in CS testing cluster, with students each added to their group project
2	10/23	Setup of Docker container models with Ceph and NFS mounted
3	10/23	Connect Virtual Machines to Ceph
4	09/12	Prepare documentation/tutorial on installing kubectl and connecting to the CS cloud cluster
5	09/12	Complete setting up Airflow locally and tested it out by running a few workflows

Continued on next page

Table 2 – continued from previous page

Task #	Completion Date	Milestone
6	09/17	Create Docker container for Airflow on CS container cluster
7	09/24	Create Docker container for Postgres on CS container cluster
8	11/16	Document and prepare important tutorials of the process being followed for future reference
9	11/16	Prepare tutorial on deploying containers from Rancher catalogs
10	11/3	Deploy initial development versions of containers requested by other team
11	10/23	Prepare tutorial on committing changes to a new container image
12	10/23	Prepare tutorial on building a Docker image from a Docker file
13	10/23	Prepare tutorial on deploying containers to git.cs.vt.edu
14	10/28	Prepare tutorial on how to change the Elasticsearch configurations
15	11/1	Deploy Postgresql container and the Flask application to get the first version of system running
16	11/3	Deploy Airflow container and make it accessible from the frontend
17	11/16	Research on CI/CD
18	11/16	Prepare tutorial and give a demo on how to achieve CI/CD
19	12/8	Manual Service Registration

Table 3: Deliverables

Task #	Completion Date	Deliverables
1	09/5	Project setup with initial baseline containers for the other teams
2	09/17	Interim Report 1
3	09/17	Tutorials for how to use various containers
4	09/17	Setting up containers for Apache Airflow
5	10/25	Developer manual for setting up Apache Airflow
6	10/1	Developer manual for setting up Postgres
7	10/8	Interim Report 2
8	10/12	Docker container files showing example services
9	10/19	Container for the Reasoner Engine to generate workflows with information goal as input
10	10/23	Container for running Airflow API

Continued on next page

Table 3 – continued from previous page

Task #	**Completion Date**	**Deliverables**
11	10/23	Developer Manual for building Airflow compatible Docker containers
12	10/23	Developer Manual for running Airflow API
13	10/29	Interim Report 3
14	11/10	Complete set up of Airflow to programatically author, monitor and schedule the workflows generated by the reasoner engine.
15	11/16	Demo on CI/CD using GitLab
16	11/30	Load testing
17	11/30	TWT Team Service Registration
18	12/6	ETD Team Service Registration
19	12/8	WP Team Service Registration
20	12/9	Final Project Report

6 User Manual

6.1 Rancher UI: Deploying Containers and Accessing Persistent Storage

With this introductory guide, users learn how to navigate the Rancher cluster, deploy containers from images either from Docker Hub or from Rancher Catalogs, and execute processes in containers.

1. To view your respective group project(s) hosted at CS Cloud, log in to the website and browse under the *Global* menu to *testing*.

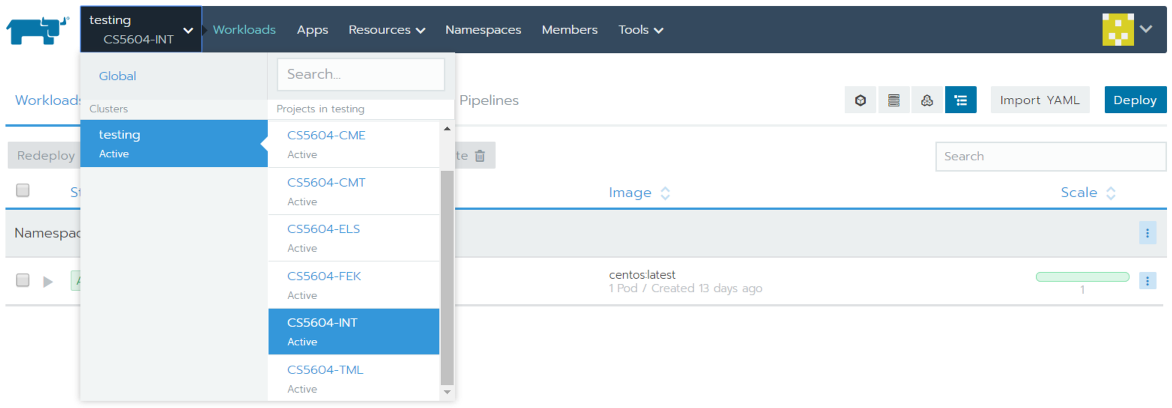


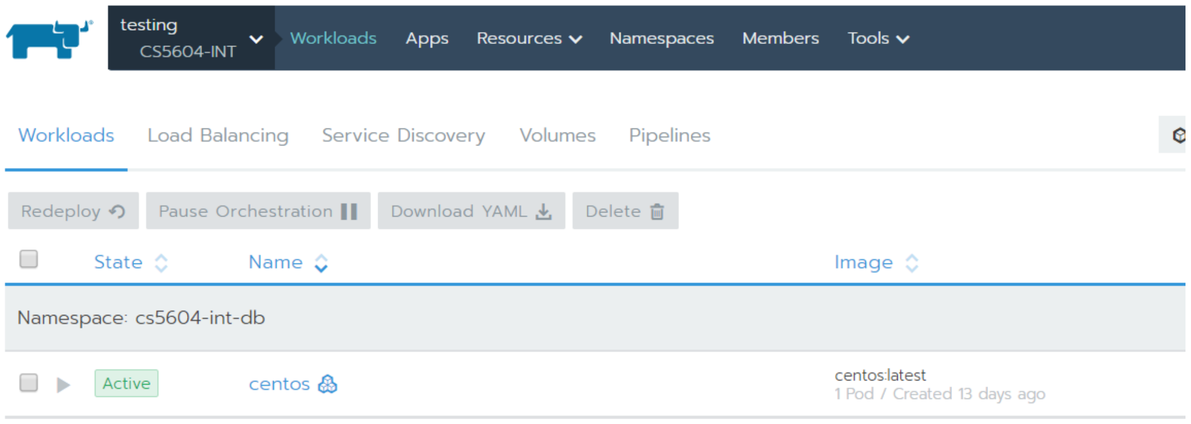
Figure 7: CS cloud cluster

2. A submenu appears under *testing* with a listing of all projects in which you have membership or ownership privileges. Figure 7 shows an example of a user with membership in all five of the projects, with the *CS5604-INT* project selected.

3. Clicking a group project will load a view of the namespaces and pods within each namespace in that project and the Docker image(s) used to deploy the pod(s) (see Figure 8).

4. Click the "..." button on the right side of the container you want to use, and select "Execute Shell" from the pop-up menu to run a shell in the container (see Figure 9). Note that all executions in this shell are not persistent in the container. The container will restart/reset and all non-committed changes will be lost. Storing data in the Ceph File System as a persistent volume is a way to save execution results.

5. In the container's shell, a user can execute commands such as running programs including running Elasticsearch and MySQL commands. Installing required packages (e.g., via `pip`) through the shell is not recommended, because the container execution runtime is ephemeral. To persist installed packages, add installation commands to the Dockerfile or commit changes via Docker CLI.



State	Name	Image
Namespace: cs5604-int-db		
Active	centos	centos:latest 1 Pod / Created 13 days ago

Figure 8: Pods and namespaces

6.2 Elasticsearch

This section is used to help users interface with Elasticsearch. Figure 13 shows the workflow and schema of the Elasticsearch configuration.

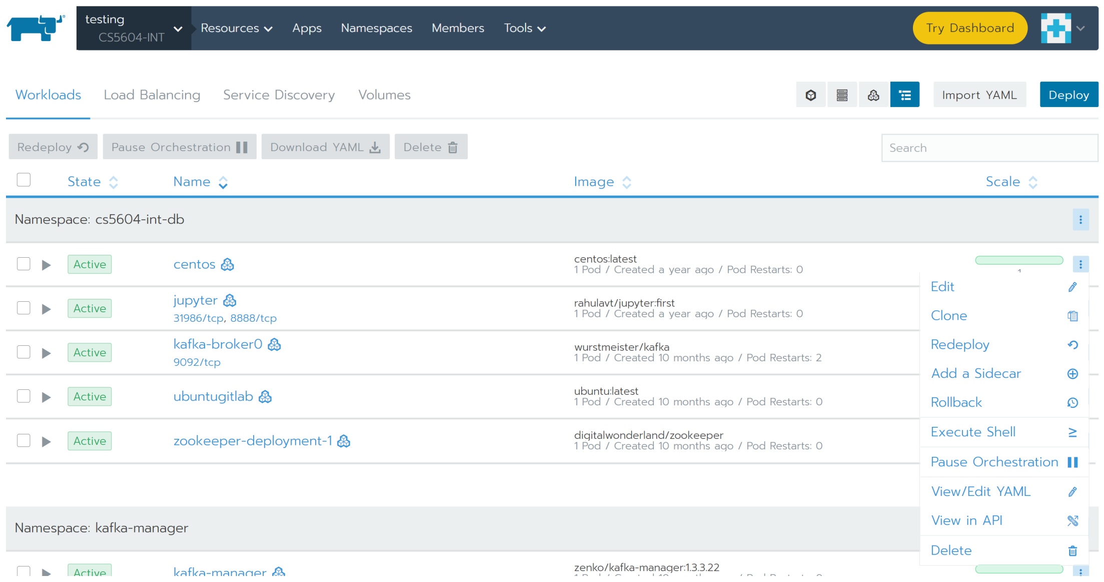
1. There is a virtual machine (VM) dedicated to Elasticsearch on the cs.vt.edu network, with sufficient local storage to hold various indexes for each of the 3 content collections.

2. The VM is configured for Elasticsearch to automatically ingest data into the VM.

3. The VM is running ElasticSearch v7.9.2 in Docker containers configured in a multi-node cluster.

4. Users will be able to connect to ElasticSearch running on the VM, to send it data.

5. The front-end API will be able to connect to Elasticsearch to send queries and return result lists.



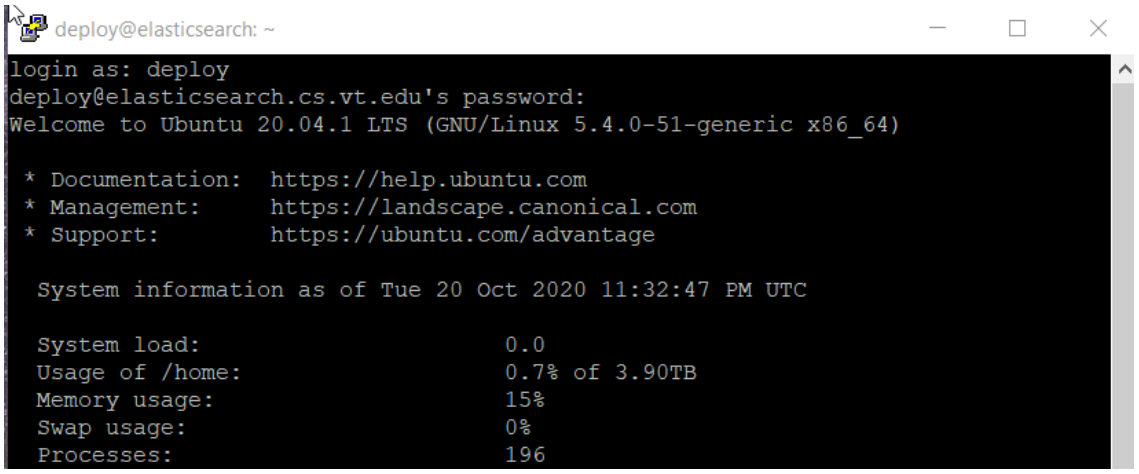
State	Name	Image	Scale
Namespace: cs5604-int-db			
Active	centos	centos:latest 1 Pod / Created a year ago / Pod Restarts: 0	
Active	jupyter 31986/tcp, 8888/tcp	rahulavt/jupyter:first 1 Pod / Created a year ago / Pod Restarts: 0	
Active	kafka-broker0 9092/tcp	wurstmeister/kafka 1 Pod / Created 10 months ago / Pod Restarts: 2	
Active	ubuntugitlab	ubuntu:latest 1 Pod / Created 10 months ago / Pod Restarts: 0	
Active	zookeeper-deployment-1	digitalwonderland/zookeeper 1 Pod / Created 10 months ago / Pod Restarts: 0	
Namespace: kafka-manager			
Active	kafka-manager	zenko/kafka-manager:1.3.3.22	

Figure 9: Container running entrance

```
{
  "name" : "es01",
  "cluster_name" : "es-docker-cluster",
  "cluster_uuid" : "aS-b64JHTZmNnLSjqF8Mdw",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

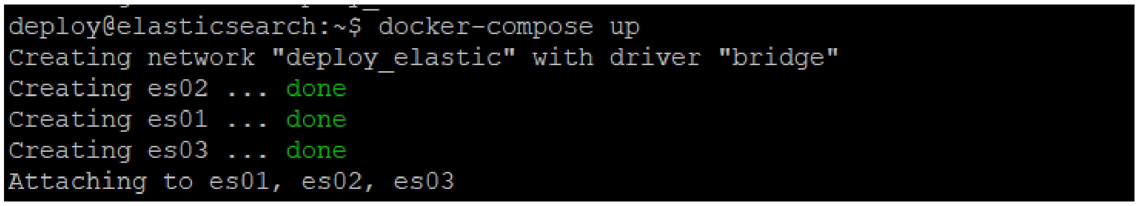
Figure 10: Elasticsearch VM Confirmational Information

6. Data will be ingested using workflows running in the Kubernetes Pods running on the cluster in Docker containers that are created by CI and Airflow. A service API for interacting with Elasticsearch will be running in containers on the cluster.



```
login as: deploy
deploy@elasticsearch.cs.vt.edu's password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-51-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue 20 Oct 2020 11:32:47 PM UTC

  System load:                      0.0
  Usage of /home:                   0.7% of 3.90TB
  Memory usage:                     15%
  Swap usage:                       0%
  Processes:                        196
```

Figure 11: Elasticsearch VM running Ubuntu



```
deploy@elasticsearch:~$ docker-compose up
Creating network "deploy_elastic" with driver "bridge"
Creating es02 ... done
Creating es01 ... done
Creating es03 ... done
Attaching to es01, es02, es03
```

Figure 12: ES v7.9.2 in a 3-node cluster in a Docker Container

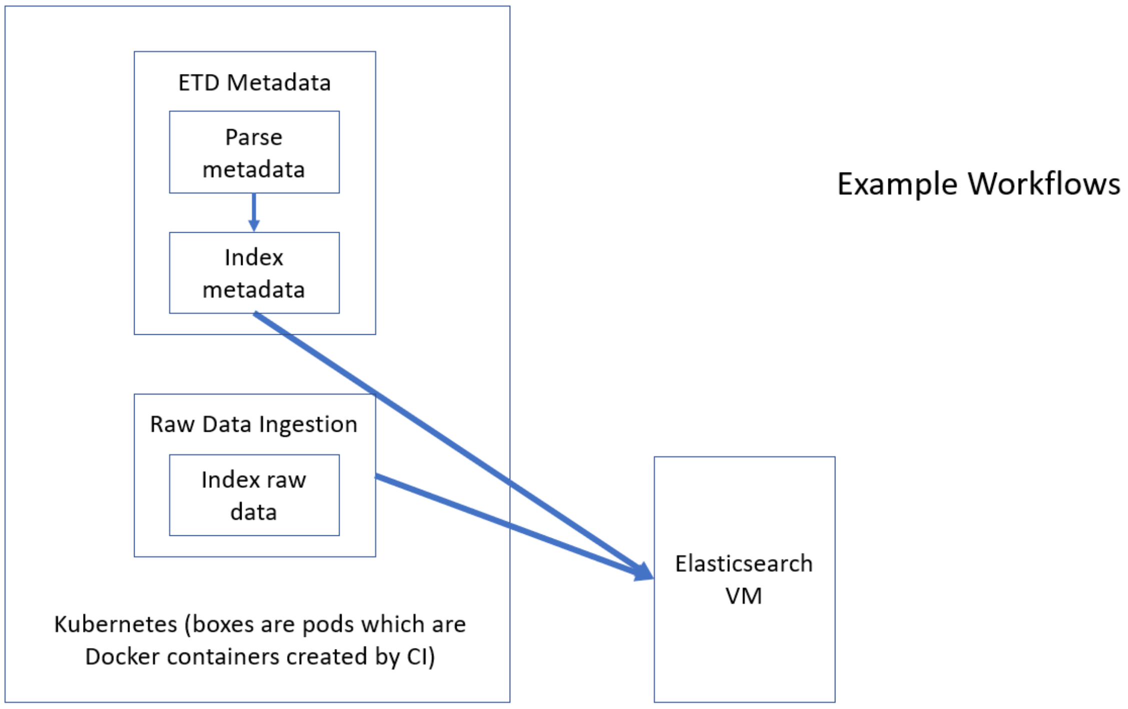
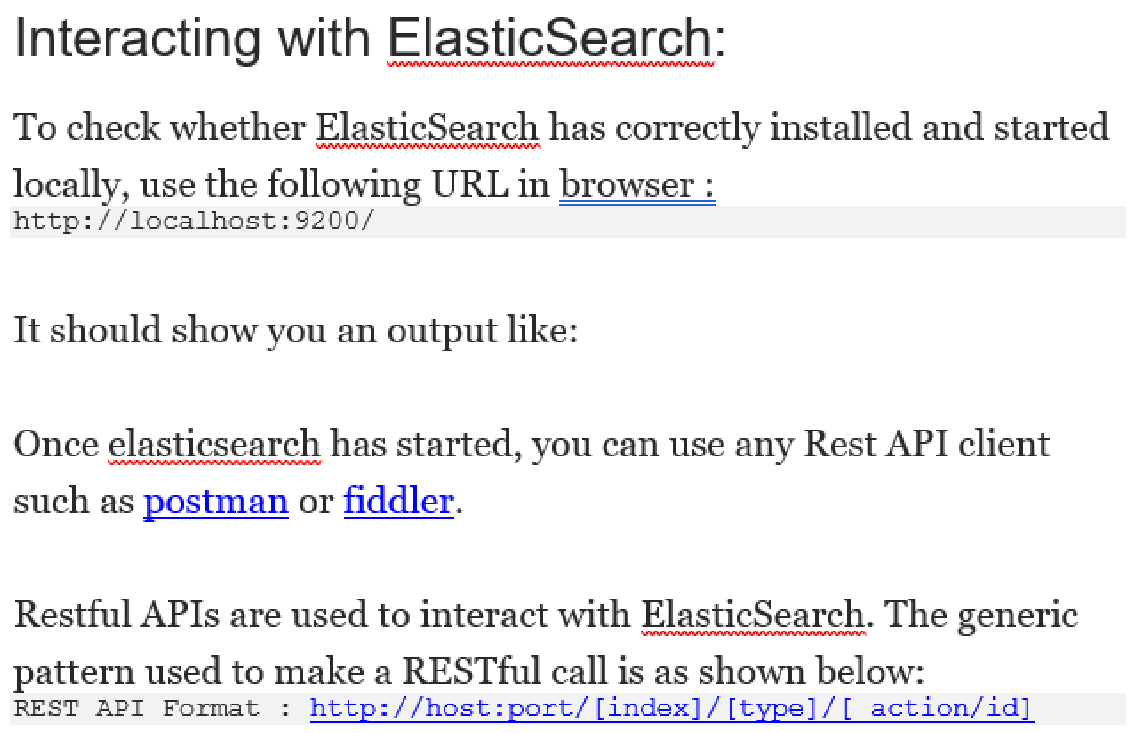


Figure 13: Example Ingestion Schema of Elasticsearch VM



Interacting with ElasticSearch:

To check whether ElasticSearch has correctly installed and started locally, use the following URL in browser :

```
http://localhost:9200/
```

It should show you an output like:

Once elasticsearch has started, you can use any Rest API client such as postman or fiddler.

Restful APIs are used to interact with ElasticSearch. The generic pattern used to make a RESTful call is as shown below:

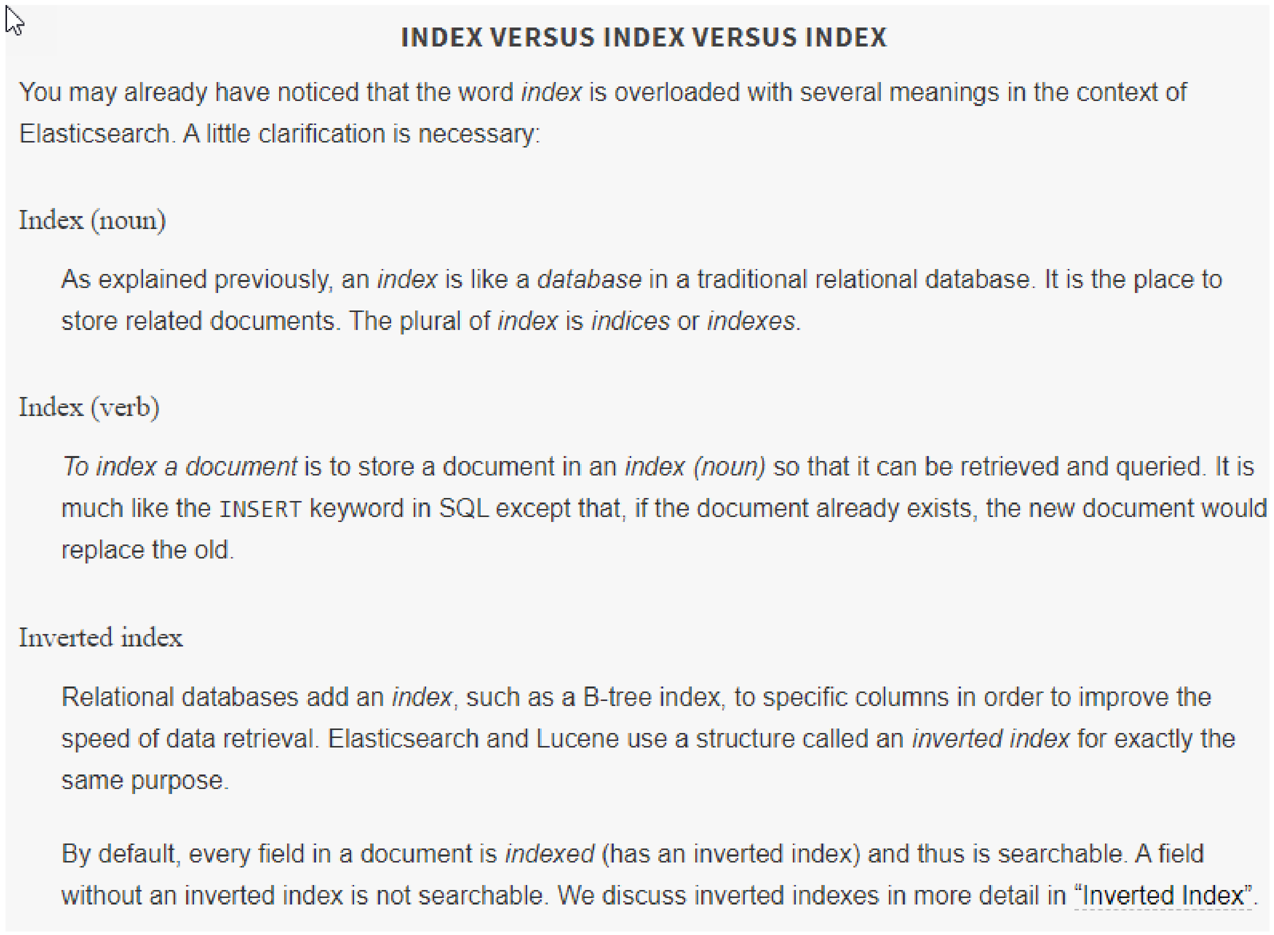
```
REST API Format : http://host:port/[index]/[type]/[ action/id]
```

Figure 14: Ways to Interact with Elasticsearch

7. Elasticsearch uses defaults that are intended to provide good full text search, highlighting, aggregations, and indexing that should all just work without the user having to configure anything.

8. Once data has been indexed in Elasticsearch, users can search it by sending requests through the search endpoint. This will be routed through the user interface developed by the FE team.

9. Sharded indexes will be confirmed once created.


INDEX VERSUS INDEX VERSUS INDEX

You may already have noticed that the word *index* is overloaded with several meanings in the context of Elasticsearch. A little clarification is necessary:

Index (noun)

As explained previously, an *index* is like a *database* in a traditional relational database. It is the place to store related documents. The plural of *index* is *indices* or *indexes*.

Index (verb)

To index a document is to store a document in an *index (noun)* so that it can be retrieved and queried. It is much like the `INSERT` keyword in SQL except that, if the document already exists, the new document would replace the old.

Inverted index

Relational databases add an *index*, such as a B-tree index, to specific columns in order to improve the speed of data retrieval. Elasticsearch and Lucene use a structure called an *inverted index* for exactly the same purpose.

By default, every field in a document is *indexed* (has an inverted index) and thus is searchable. A field without an inverted index is not searchable. We discuss inverted indexes in more detail in "Inverted Index".

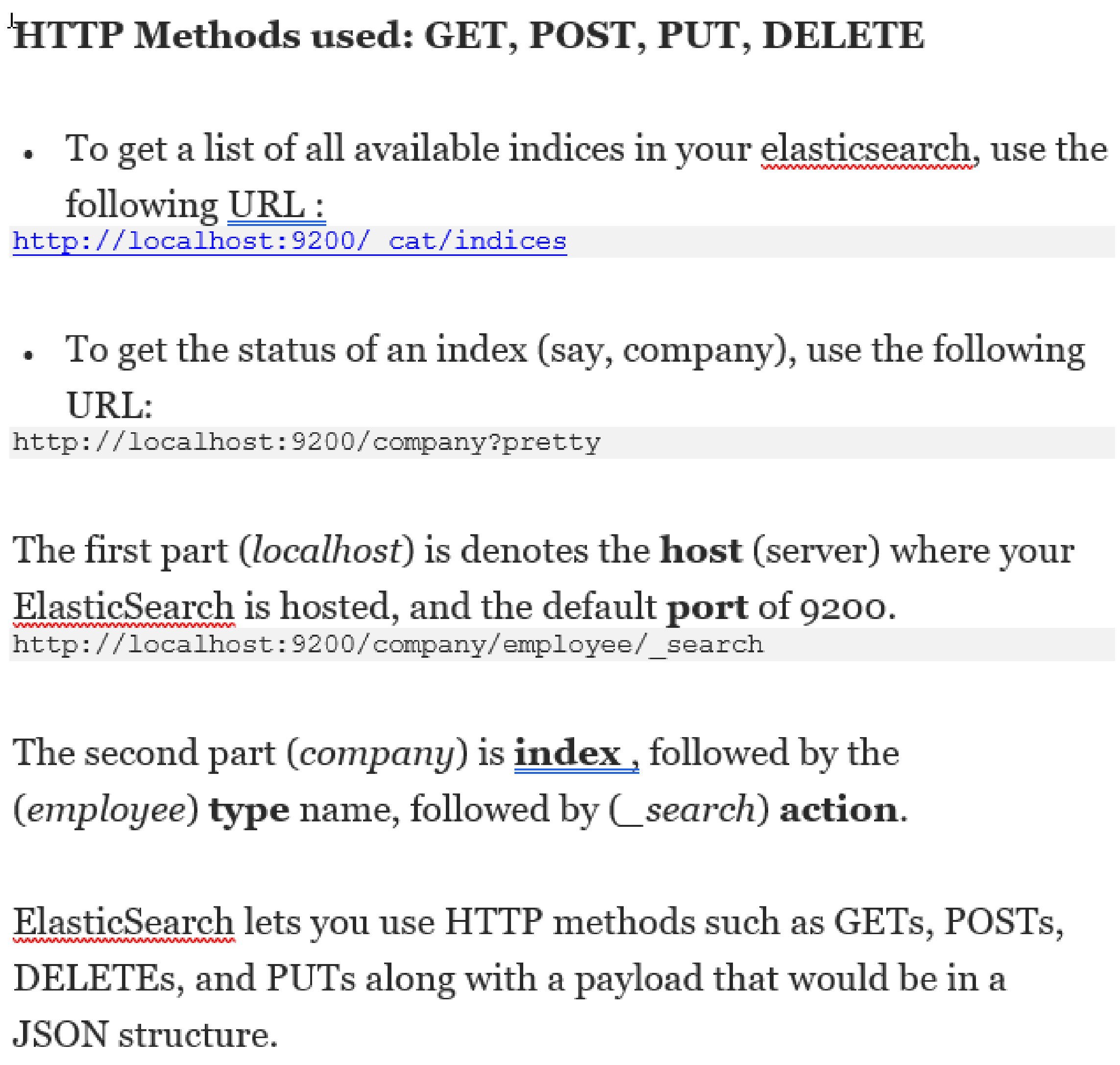
Figure 15: Indexing with Elasticsearch

Teams may also interact with Elasticsearch programmatically using the example Python file located in the source code [36].

6.3 Service Registry REST API

In order to interact with the service and goal tables that Airflow pulls from, the Front End team will interact with the service API. The service API is exposed at `http://[2001:468:c80:6102:1:7015:81b3:1760]:5000/services/` and the goal API is exposed at `http://[2001:468:c80:6102:1:7015:81b3:1760]:5000/goals/`. Both of these endpoints function similarly so the below section will refer only to the service endpoint. This endpoint provides a full CRUD API where a user can send the following HTTP requests:

- GET - /services/ or /services/<id>/
- POST - /services/
- PUT - /services/<id>/


HTTP Methods used: GET, POST, PUT, DELETE

- To get a list of all available indices in your elasticsearch, use the following URL :

```
http://localhost:9200/_cat/indices
```

- To get the status of an index (say, company), use the following URL:

```
http://localhost:9200/company?pretty
```

The first part (*localhost*) is denotes the **host** (server) where your ElasticSearch is hosted, and the default **port** of 9200.

```
http://localhost:9200/company/employee/_search
```

The second part (*company*) is **index** , followed by the (*employee*) **type** name, followed by (*_search*) **action**.

ElasticSearch lets you use HTTP methods such as GETs, POSTs, DELETEs, and PUTs along with a payload that would be in a JSON structure.

Figure 16: Indexing on Elasticsearch

- DELETE - /services/<id>/

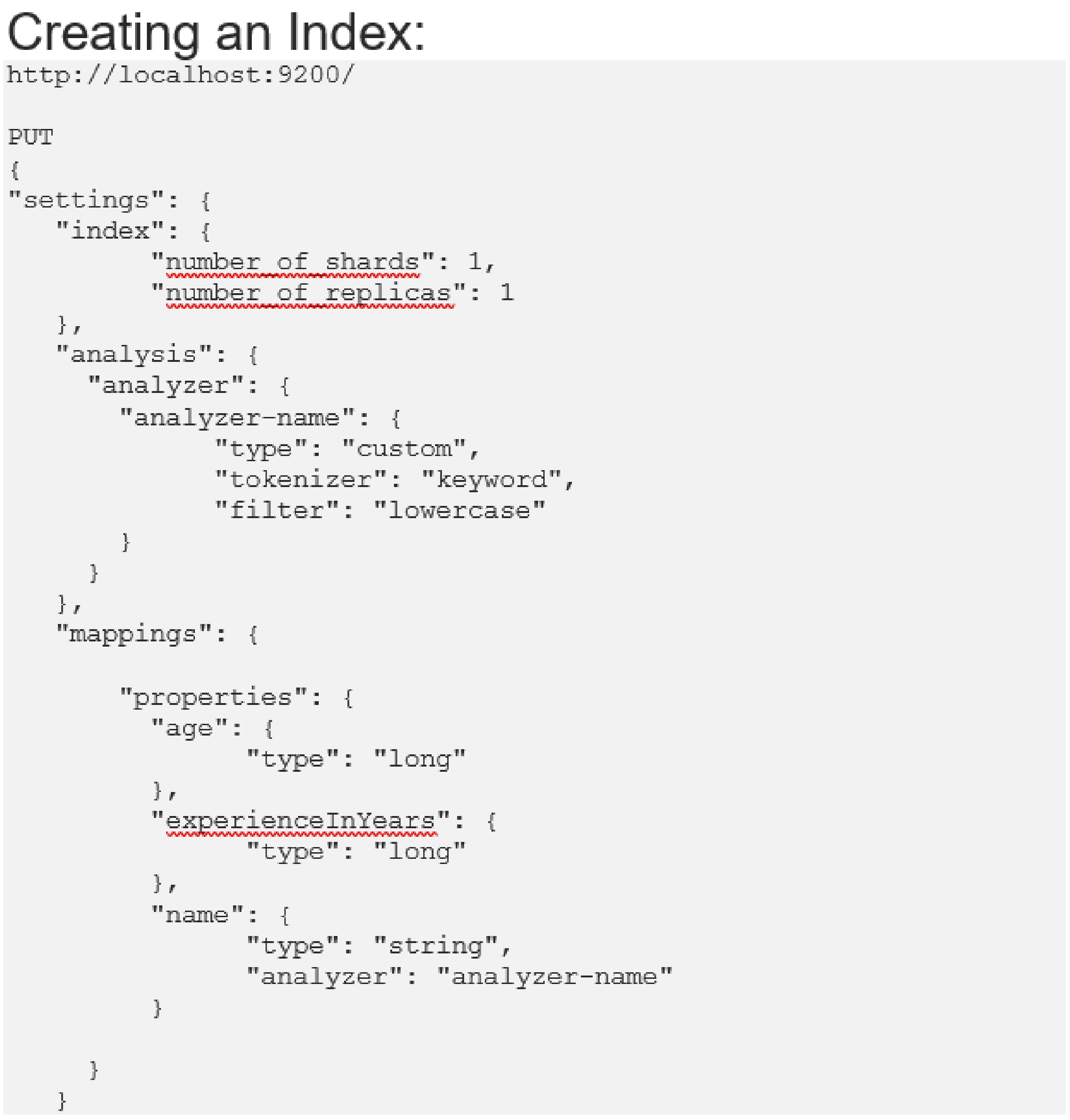
For the POST and PUT requests, the data should be passed as JSON encoded data in the following format:

6.4 Interacting with workflow related REST APIs

The frontend will have to generate and trigger workflows using REST API calls (see Fig. 21). The Reasoner API is accessible at `http://reasoner.cs5604-int-test.svc.cluster.local:5000` and the Airflow API is accessible at `http://airflow.cs5604-int-test.svc.cluster.local:5000`.

6.4.1 Reasoner API

This API is used to mine workflows using the reasoner table (see Section 7.13) and dynamically generate an Airflow compatible document that would represent the mined workflow. The following


Creating an Index:

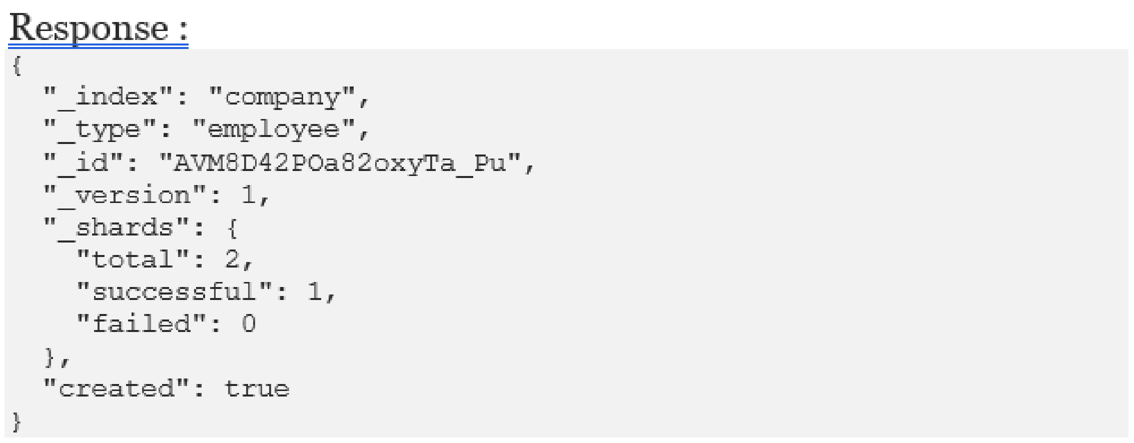
```
http://localhost:9200/

PUT
{
"settings": {
   "index": {
         "number_of_shards": 1,
         "number_of_replicas": 1
   },
   "analysis": {
     "analyzer": {
       "analyzer-name": {
             "type": "custom",
             "tokenizer": "keyword",
             "filter": "lowercase"
       }
     }
   },
   "mappings": {

       "properties": {
         "age": {
               "type": "long"
         },
         "experienceInYears": {
               "type": "long"
         },
         "name": {
               "type": "string",
               "analyzer": "analyzer-name"
         }

     }
   }
```

Figure 17: Index Mapping

are the endpoints accessible.

- `/generateGrammar` - This endpoint supports the `GET` method and is used to modify the existing production rules. Production rules are generated by reading values from the reasoner table. Therefore, every time any update is done on the table, this endpoint needs to be called. The generated grammar is stored as a file within the container running the API. This API call will be done as a part of the service registry API and will not have to be handled by the frontend.

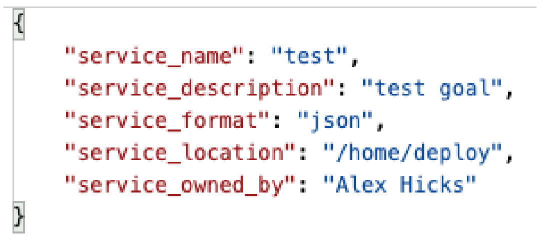
- `/generateWorklow/<requestedGoal>?dry=True` - This endpoint supports the `GET` method and is used to generate the workflow based on the production rules generated using the previous endpoint. Along with this, it also dynamically generates an Airflow document



Response :

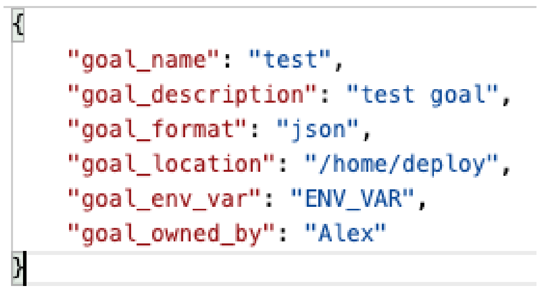
```
{
  "_index": "company",
  "_type": "employee",
  "_id": "AVM8D42POa82oxyTa_Pu",
  "_version": 1,
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

Figure 18: Index Shard Response



```
{
    "service_name": "test",
    "service_description": "test goal",
    "service_format": "json",
    "service_location": "/home/deploy",
    "service_owned_by": "Alex Hicks"
}
```

Figure 19: Posting to the Service API



```
{
    "goal_name": "test",
    "goal_description": "test goal",
    "goal_format": "json",
    "goal_location": "/home/deploy",
    "goal_env_var": "ENV_VAR",
    "goal_owned_by": "Alex"
}
```

Figure 20: Posting to the Goal API

for the mined workflow using Python Jinja templates. This document is then uploaded to the Airflow API (`/workflow/upload` endpoint) implicitly. The second parameter states the `goal_id` (see Section 7.13) needed by the end user. The endpoint returns a JSON response as shown in Figure 22. The endpoint also has an optional parameter `dry` which can be either True or False (default). If it is True, the dynamically generated Airflow document does not get uploaded to the Airflow API.

6.4.2 Airflow API

This API is used to run workflows or run a particular service in a workflow (useful when user interaction is required). The following endpoints are available in the API.

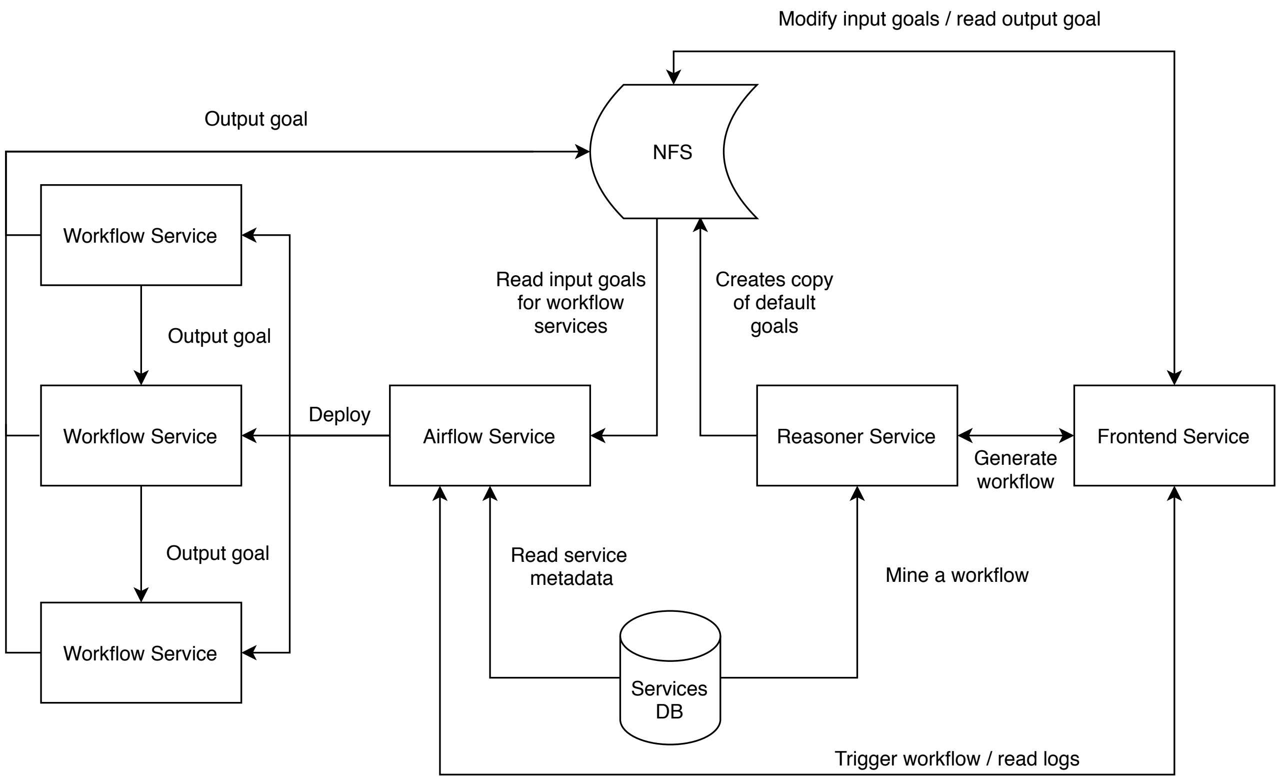
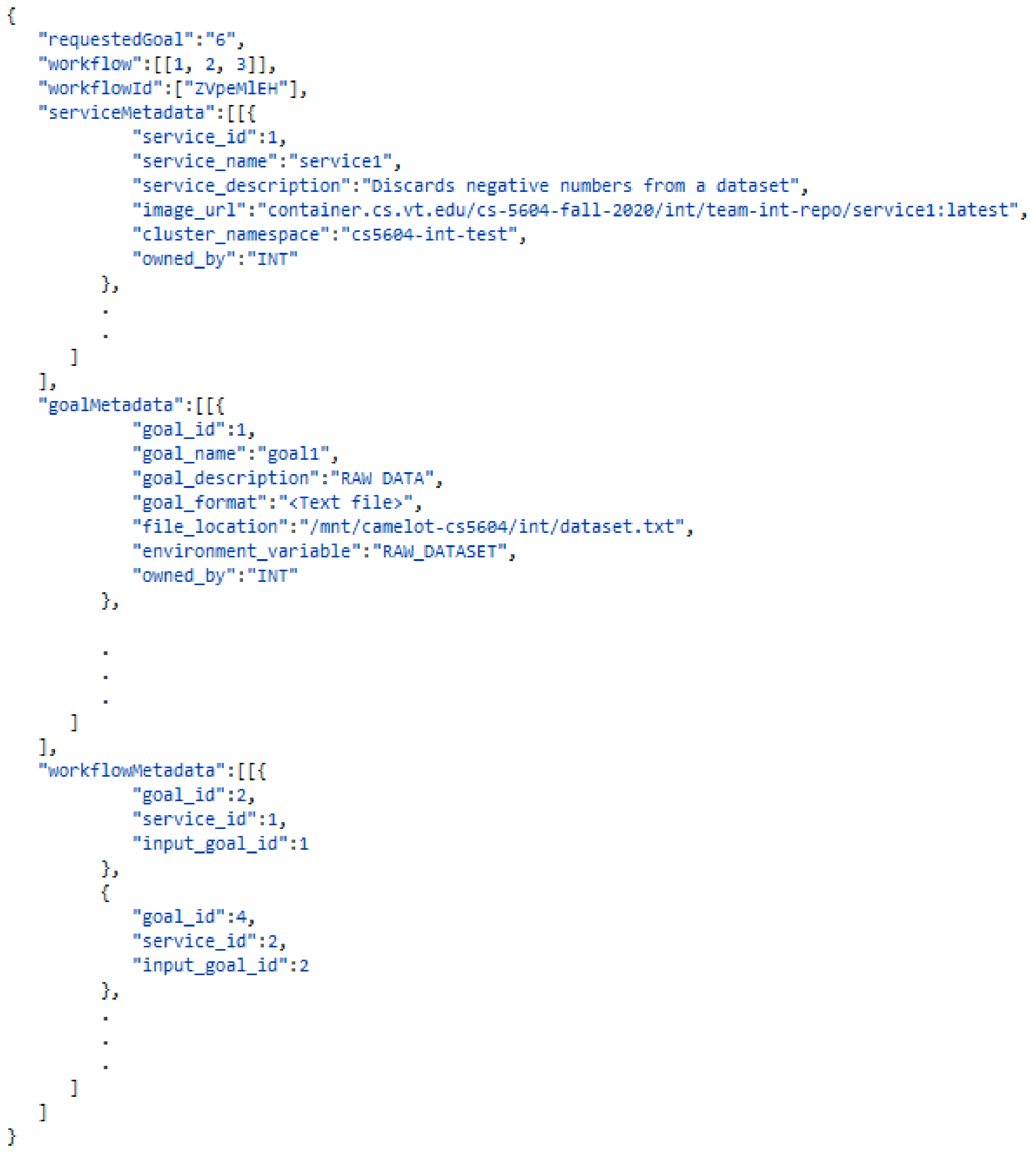


Figure 21: Frontend interaction with Airflow

- `/workflow/upload` - This endpoint expects a `POST` request with the `files` parameter to be set to the dynamically generated Airflow compatible document obtained by calling the reasoner API. No explicit API call needs to be made by the frontend to upload the document. The endpoint is implicitly called in the `/generateWorkflow` route of the reasoner API.

- `/workflow/run` - This endpoint is used to run a workflow as it is. If the end user does not wish to modify any input goals during a workflow execution, this endpoint can be used to run the workflow from the first service to the last. During the execution, all default values will be used (stored in the `file_location` field of the Goal Metadata table) in services. The `POSTed` data to this endpoint should contain a `JSON` dictionary (refer to 23 for example Python code). Once the data is posted, a `key` and a `result_url` is obtained in the response which can be used to see if the execution of the workflow is completed or not. If the workflow completes execution or fails unexpectedly, the logs can be found by sending a `GET` to the `result_url`.

- `/service/run` - This endpoint is used to run a particular service within a given workflow. This endpoint is helpful to use when services in the workflow require user interaction. Using this endpoint, the frontend can run a service in the workflow, wait for the result, present it to the user, obtain user input, and then execute the next service in the workflow using this user input. The request/response semantics for this endpoint are similar to the `/workflow/run` endpoint.

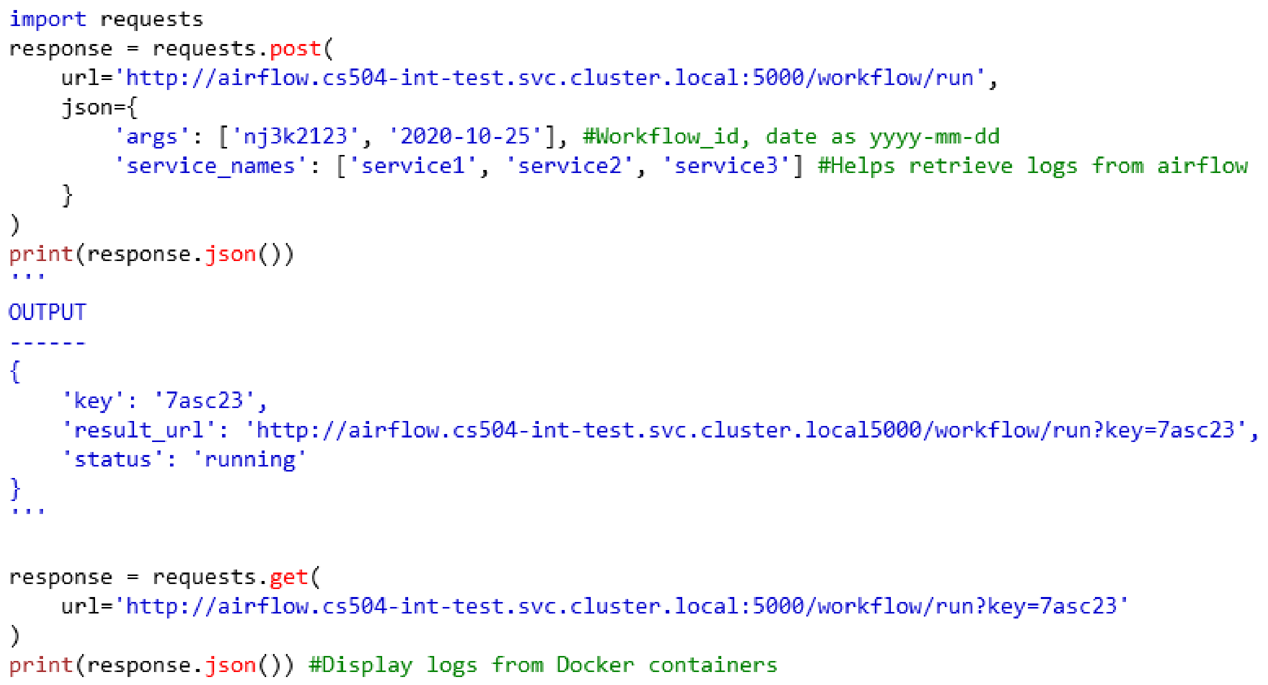


```
{
   "requestedGoal":"6",
   "workflow":[[1, 2, 3]],
   "workflowId":["ZVpeMlEH"],
   "serviceMetadata":[[{
            "service_id":1,
            "service_name":"service1",
            "service_description":"Discards negative numbers from a dataset",
            "image_url":"container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/service1:latest",
            "cluster_namespace":"cs5604-int-test",
            "owned_by":"INT"
         },
         .
         .
      ]
   ],
   "goalMetadata":[[{
            "goal_id":1,
            "goal_name":"goal1",
            "goal_description":"RAW DATA",
            "goal_format":"<Text file>",
            "file_location":"/mnt/camelot-cs5604/int/dataset.txt",
            "environment_variable":"RAW_DATASET",
            "owned_by":"INT"
         },

         .
         .
         .
      ]
   ],
   "workflowMetadata":[[{
            "goal_id":2,
            "service_id":1,
            "input_goal_id":1
         },
         {
            "goal_id":4,
            "service_id":2,
            "input_goal_id":2
         },
         .
         .
         .
      ]
   ]
}
```

Figure 22: Response of the /generateWorkflow API endpoint

6.4.3 Modifying goals using user inputs

The input goals for a service can be found at
`/mnt/ceph/workflow-id/service-id/input-goal-id/file.ext` and the output for a service can be found at `/mnt/ceph/workflow-id/service-id/output-goal-id/file.ext`. These locations



```
import requests
response = requests.post(
    url='http://airflow.cs504-int-test.svc.cluster.local:5000/workflow/run',
    json={
        'args': ['nj3k2123', '2020-10-25'], #Workflow_id, date as yyyy-mm-dd
        'service_names': ['service1', 'service2', 'service3'] #Helps retrieve logs from airflow
    }
)
print(response.json())
'''
OUTPUT
------
{
    'key': '7asc23',
    'result_url': 'http://airflow.cs504-int-test.svc.cluster.local5000/workflow/run?key=7asc23',
    'status': 'running'
}
'''

response = requests.get(
    url='http://airflow.cs504-int-test.svc.cluster.local:5000/workflow/run?key=7asc23'
)
print(response.json()) #Display logs from Docker containers
```

Figure 23: Airflow API usage example

can be accessed by the frontend application to facilitate user input. These locations need not point to a file; it could also be a directory. To identify the goals used by individual services, the frontend can use the `servicesMetadata` obtained from the `/generateWorkflow` route of the reasoner API. Obtaining information related to each goal/service can be done using the Service Registry API.

6.5 Containerizing a Service for Airflow

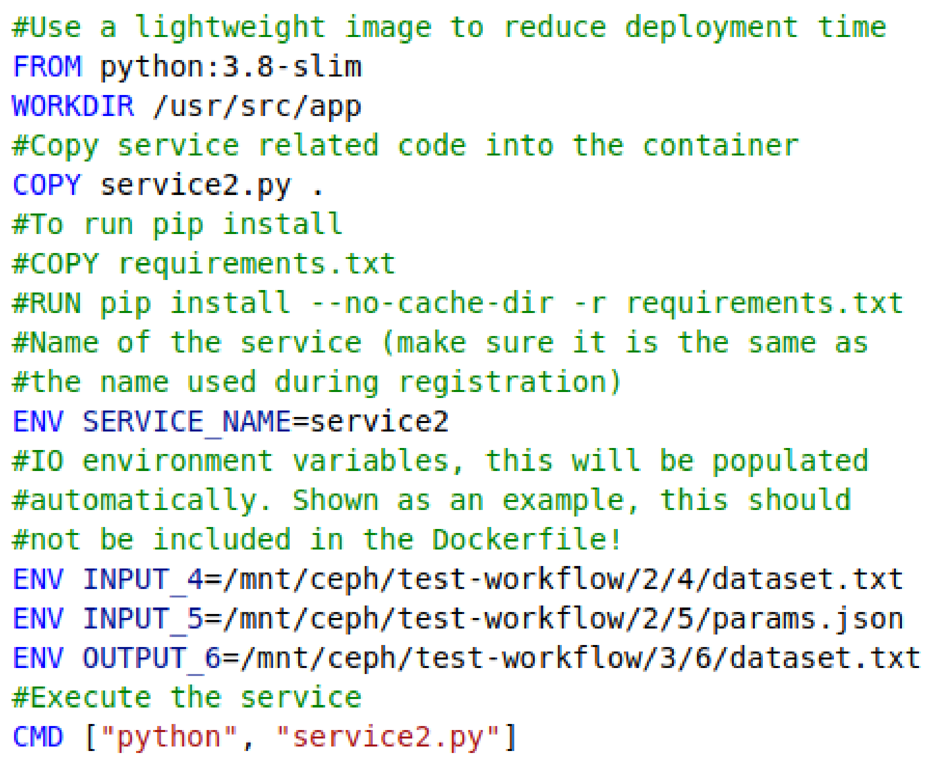
All services are expected to run within Docker containers deployed through Airflow. The developers of each service must test the containers locally before registering it into the system.

6.5.1 Creating a Dockerfile

While creating the Dockerfile for a service, the developer must keep in mind to use a lightweight base image. This ensures quick deployment and reduces waiting time for the end user. It is advisable to use official Docker images as the base image. A sample Dockerfile can be seen in Figure 24.

6.5.2 Pushing the Service to a Container Registry

Once the Dockerfile is created and tested locally, the developed service image needs to be pushed to a container registry. Airflow pulls these images from the container registry and deploys the containers on a Kubernetes cluster. The image URL needs to be stored in the Service Metadata Column (see Section 7.13). To push the image to Virginia Tech's container registry (`container.cs.vt.edu`) the following commands can be run within the directory that contains the Dockerfile.



```
#Use a lightweight image to reduce deployment time
FROM python:3.8-slim
WORKDIR /usr/src/app
#Copy service related code into the container
COPY service2.py .
#To run pip install
#COPY requirements.txt
#RUN pip install --no-cache-dir -r requirements.txt
#Name of the service (make sure it is the same as
#the name used during registration)
ENV SERVICE_NAME=service2
#IO environment variables, this will be populated
#automatically. Shown as an example, this should
#not be included in the Dockerfile!
ENV INPUT_4=/mnt/ceph/test-workflow/2/4/dataset.txt
ENV INPUT_5=/mnt/ceph/test-workflow/2/5/params.json
ENV OUTPUT_6=/mnt/ceph/test-workflow/3/6/dataset.txt
#Execute the service
CMD ["python", "service2.py"]
```

Figure 24: An example Dockerfile for a service

```
docker login container.cs.vt.edu
docker build -t container.cs.vt.edu/cs5604-fall-2020/team-int-repo/img:latest .
docker push container.cs.vt.edu/cs5604-fall-2020/team-int-repo/img:latest
```

The login command prompts a username and password, in which you must enter your VT CS login credentials. Make sure to always include `:latest` in the image name. This ensures that Airflow always pulls the latest image from the registry.

6.5.3 Using Input/Output Environment Variables

The input filenames and the output filename will be automatically available as environment variables named in the `goal_metadata` table. The reading/writing into the files or folders pointed to by the environment variable needs to be handled by the developers of the service. Note that the Dockerfile should not declare these environment variables as it gets automatically generated and is shown in Figure 24 as an example.

6.6 CI/CD with Gitlab

In this section, we will demonstrate step-by-step how to implement a simple CI/CD with the Gitlab.

1. The first step is to configure a Gitlab runner. A Gitlab runner is an application that is used to run the jobs in the pipeline. All the services would be executed and tested on this application. While Gitlab offers some runners for free, it is better to configure your own runner so that

there is no wait/down time in order to test the pipeline. The step-wise process to configure a runner is explained in the Developer Manual Section 7.11.

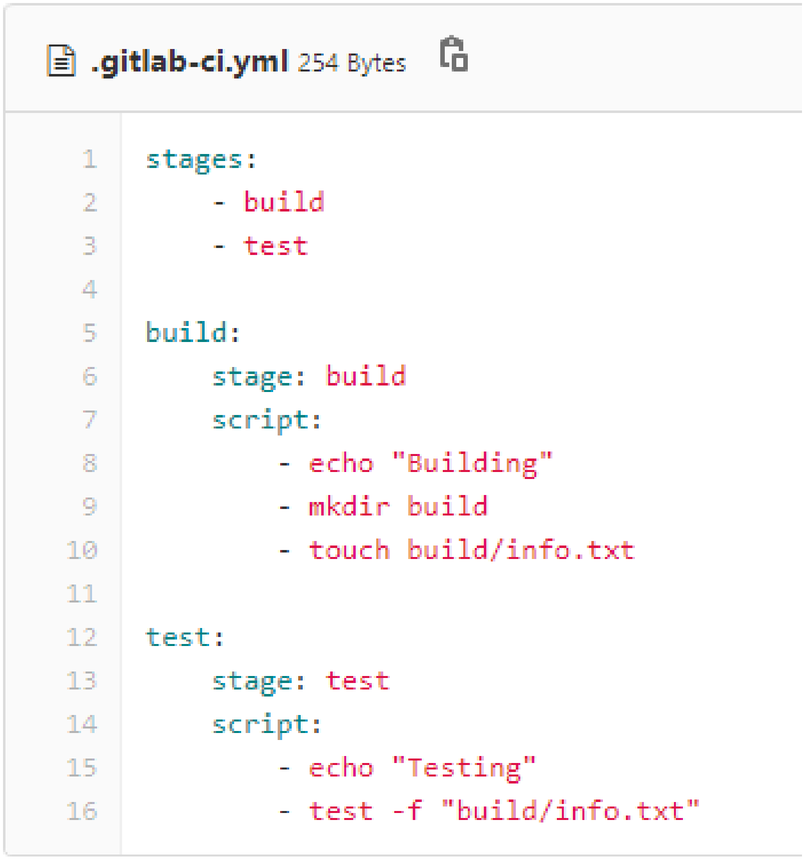
2. Most CI/CD tools expect a configuration file to exist in the project. Similarly, Gitlab CI/CD requires a .gitlab-ci.yml file to exist in the project root.

3. The .gitlab-ci.yml file would contain all of the configuration details and tasks that one wants to perform such as unit testing, integration testing, etc. There are usually different stages such as "build" and "test". This file also includes the script for the tasks that need to be performed. There are many more configurations that can be added as illustrated in `https://docs.gitlab.com/ee/ci/yaml/`. Here we present two examples of a typical .gitlab-ci.yml file as shown in Figures 25 and 26.

4. Figure 25 shows a simple configuration where we generate a text file in the "build" stage which is our main objective and then test it in the "test" stage if the file was successfully created. Figure 26 shows a more complex version that includes building and running a Docker container. Similar to Figure 25, in the build stage, we build an image and then run it. Here the objective of this service (say Service1) is to create a .json file. Then in the "test" stage, we use another Docker image to test if the .json generated from Service1 is a valid .json or not. These examples are just a simple illustration but one can configure them to achieve more complex testing for their pipeline.

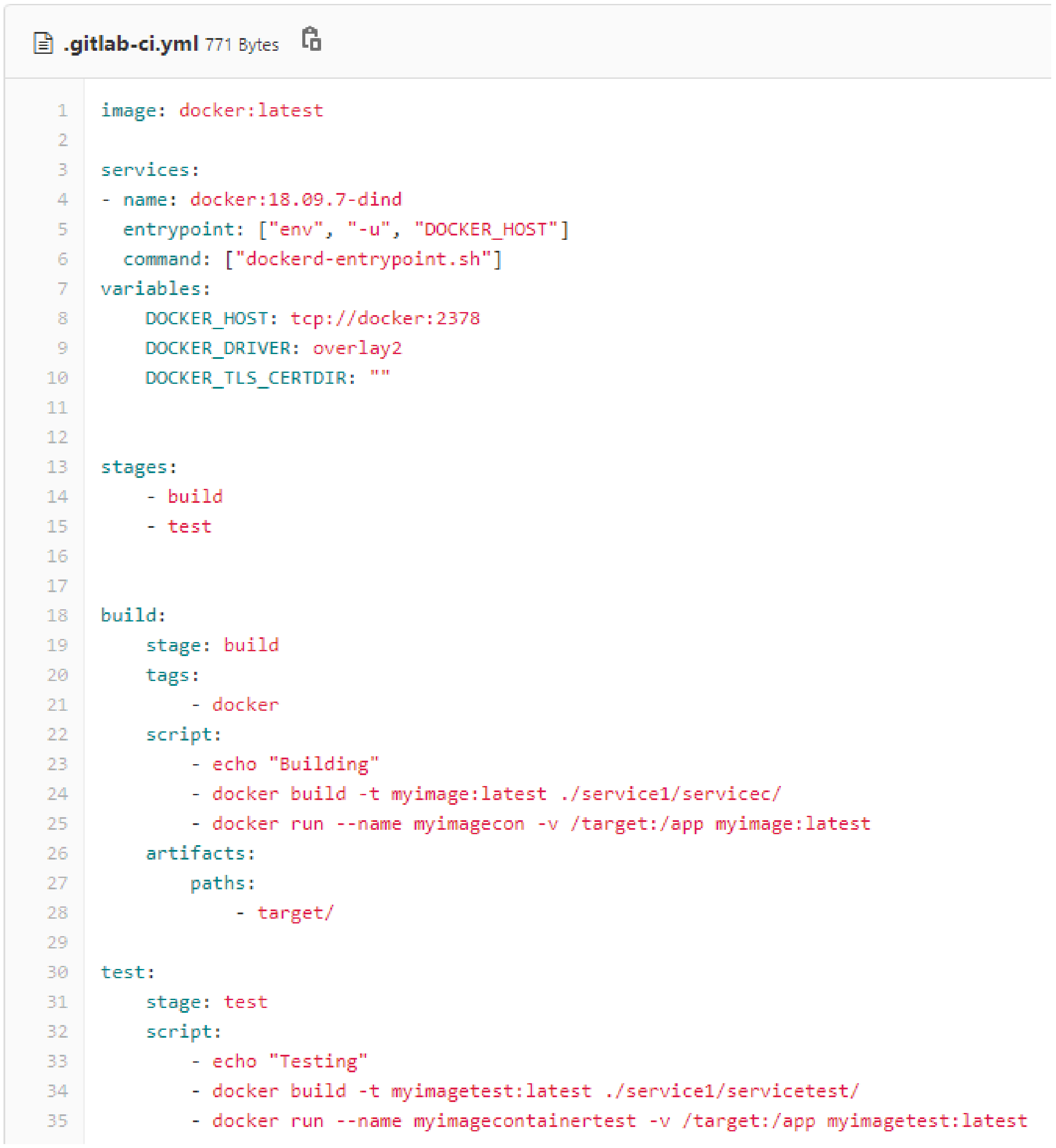
5. Once the repository is ready, you do a git commit. The moment you push a change, .gitlab-ci.yml file would get triggered and start executing the tasks as defined in various stages. This can be interactively seen on the Gitlab»CI/CD»Pipelines (see Figure 27). As shown in the figure, one could see various stages the pipeline contains. One can also click on these stages and see the terminal where all of the task are getting executed. This could come in handy for debugging in case there is an error (as shown in Figure 28).

6. Lastly, the Docker images can be pushed into the container registry for deployment.



```yaml
stages:
    - build
    - test

build:
    stage: build
    script:
        - echo "Building"
        - mkdir build
        - touch build/info.txt

test:
    stage: test
    script:
        - echo "Testing"
        - test -f "build/info.txt"
```

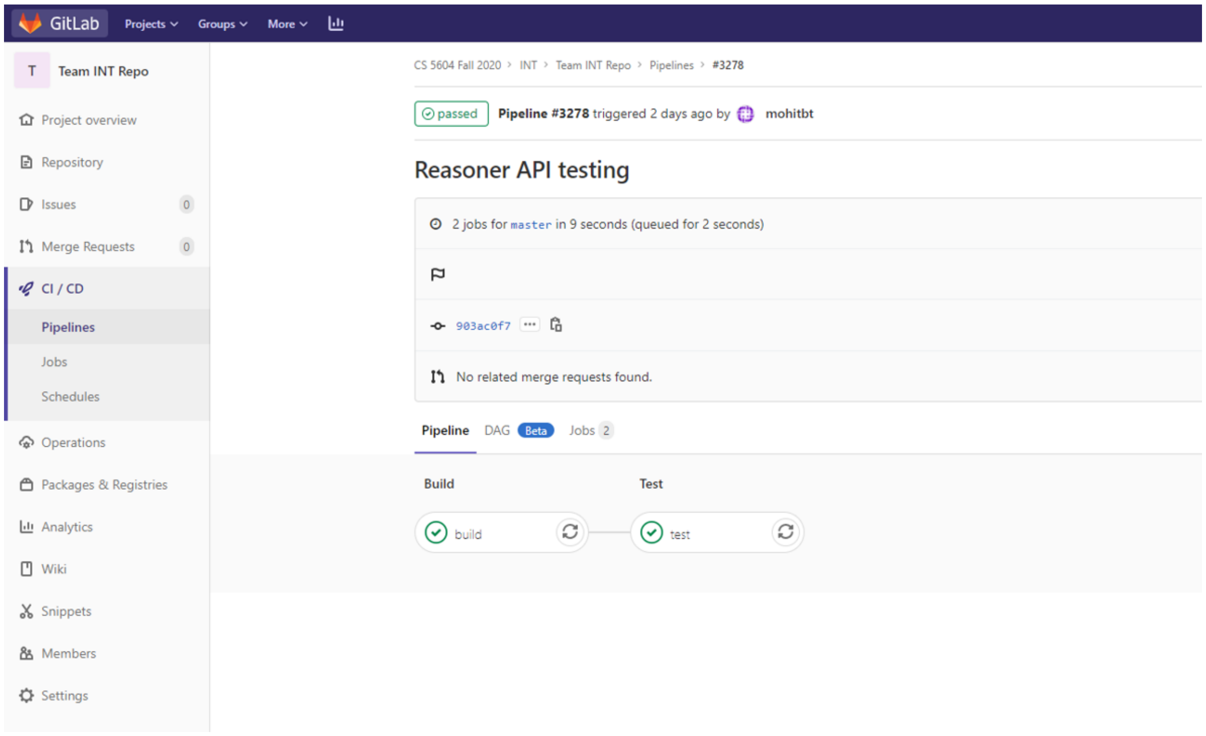
Figure 25: Example 1: .gitlab-ci.yml file



.gitlab-ci.yml 771 Bytes

```yaml
image: docker:latest

services:
- name: docker:18.09.7-dind
  entrypoint: ["env", "-u", "DOCKER_HOST"]
  command: ["dockerd-entrypoint.sh"]
variables:
    DOCKER_HOST: tcp://docker:2378
    DOCKER_DRIVER: overlay2
    DOCKER_TLS_CERTDIR: ""


stages:
    - build
    - test


build:
    stage: build
    tags:
        - docker
    script:
        - echo "Building"
        - docker build -t myimage:latest ./service1/servicec/
        - docker run --name myimagecon -v /target:/app myimage:latest
    artifacts:
        paths:
            - target/

test:
    stage: test
    script:
        - echo "Testing"
        - docker build -t myimagetest:latest ./service1/servicetest/
        - docker run --name myimagecontainertest -v /target:/app myimagetest:latest
```

Figure 26: Example 2: .gitlab-ci.yml file



passed **Pipeline #3278** triggered 2 days ago by mohitbt

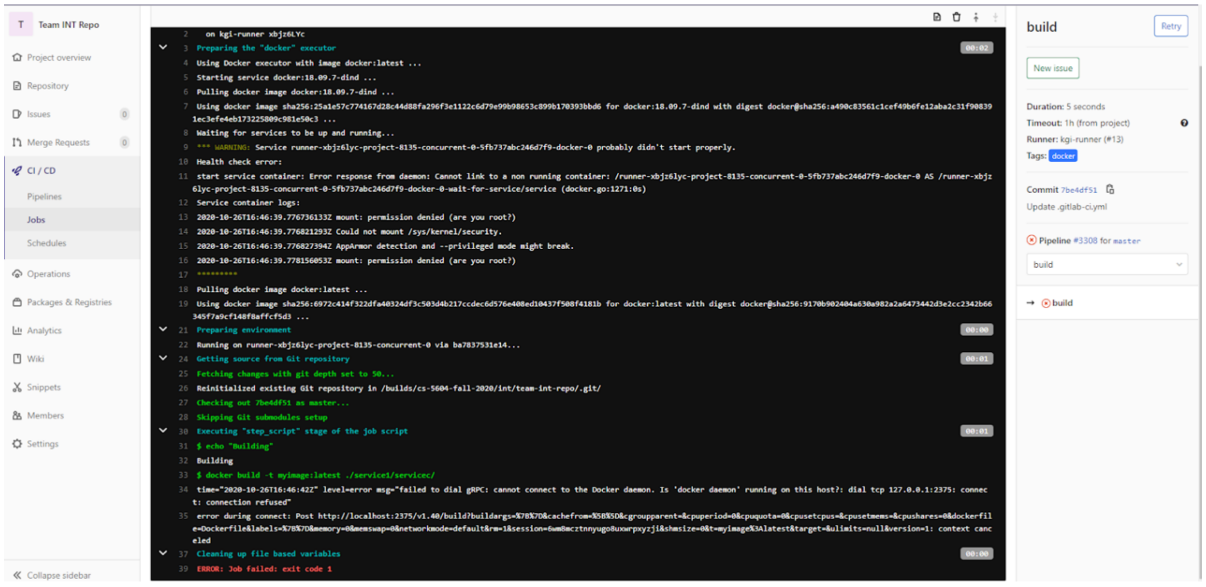
Reasoner API testing

2 jobs for master in 9 seconds (queued for 2 seconds)

903ac0f7

No related merge requests found.

Figure 27: Gitlab CI/CD stages



```
  on kgi-runner xbjz6LYc
Preparing the "docker" executor                                         00:02
Using Docker executor with image docker:latest ...
Starting service docker:18.09.7-dind ...
Pulling docker image docker:18.09.7-dind ...
Using docker image sha256:25a1e57c774167d28c44d88fa296f3e1122c6d79e99b98653c899b170393bbd6 for docker:18.09.7-dind with digest docker@sha256:a490c83561c1cef49b6fe12aba2c31f908391ec3efe4eb173225809c981e50c3 ...
Waiting for services to be up and running...
*** WARNING: Service runner-xbjz6lyc-project-8135-concurrent-0-5fb737abc246d7f9-docker-0 probably didn't start properly.
Health check error:
start service container: Error response from daemon: Cannot link to a non running container: /runner-xbjz6lyc-project-8135-concurrent-0-5fb737abc246d7f9-docker-0 AS /runner-xbjz6lyc-project-8135-concurrent-0-5fb737abc246d7f9-docker-0-wait-for-service/service (docker.go:1271:0s)
Service container logs:
2020-10-26T16:46:39.776736133Z mount: permission denied (are you root?)
2020-10-26T16:46:39.776821293Z Could not mount /sys/kernel/security.
2020-10-26T16:46:39.776827394Z AppArmor detection and --privileged mode might break.
2020-10-26T16:46:39.778156053Z mount: permission denied (are you root?)
*********
Pulling docker image docker:latest ...
Using docker image sha256:6972c414f322dfa40324df3c503d4b217ccdec6d576e408ed10437f508f4181b for docker:latest with digest docker@sha256:9170b902404a630a982a2a6473442d3e2cc2342b66345f7a9cf148f8affcf5d3 ...
Preparing environment                                                   00:00
Running on runner-xbjz6lyc-project-8135-concurrent-0 via ba7837531e14...
Getting source from Git repository                                      00:01
Fetching changes with git depth set to 50...
Reinitialized existing Git repository in /builds/cs-5604-fall-2020/int/team-int-repo/.git/
Checking out 7be4df51 as master...
Skipping Git submodules setup
Executing "step_script" stage of the job script                         00:01
$ echo "Building"
Building
$ docker build -t myimage:latest ./service1/servicec/
time="2020-10-26T16:46:42Z" level=error msg="failed to dial gRPC: cannot connect to the Docker daemon. Is 'docker daemon' running on this host?: dial tcp 127.0.0.1:2375: connect: connection refused"
error during connect: Post http://localhost:2375/v1.40/build?buildargs=%7B%7D&cachefrom=%5B%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpushares=0&dockerfile=Dockerfile&labels=%7B%7D&memory=0&memswap=0&networkmode=default&rm=1&session=6wm8mcztnnyugo8uxwrpxyzji&shmsize=0&t=myimage%3Alatest&target=&ulimits=null&version=1: context canceled
Cleaning up file based variables                                        00:00
ERROR: Job failed: exit code 1
```

Duration: 5 seconds
Timeout: 1h (from project)
Runner: kgi-runner (#13)
Tags: docker

Commit 7be4df51
Update .gitlab-ci.yml

Pipeline #3308 for master

build

→ build

Figure 28: Terminal view of the tasks executing in the build stage as seen through the Gitlab CI/CD interface

7 Developer Manual

7.1 Providing Access to Ceph Persistent Storage from KGI and ELS VMs

In order to be able to transfer large processed datasets from the virtual machines, that store the ingested datasets, into Ceph storage, a Ceph client must be installed on the VMs. Then, Ceph is mounted onto the VMs, facilitating direct access. The login information and secret key are examples (they're not real). This is a reference that explains the process. No action is required from any team.

1. Using the terminal on your machine, connect to the VM (a CentOS machine); must have root access:

   ```
   ssh user@etdvm
   ```

2. Enter the password:

   ```
   Password
   ```

   The following steps are executed in the VM:

3. Install the Ceph tools and Ceph client on the VM:

   ```
   sudo yum -y install centos-release-ceph-nautilus
   sudo yum -y install ceph-common
   ```

4. To have permissions to connect to and access the CS cluster, you will have to first:

   Store the cluster's secret key into a file (we named the file here: cephfs.secret)

   ```
   echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > cephfs.secret
   ```

   Assign read and write permissions for cephfs.secret

   ```
   chmod 600 cephfs.secret
   ```

5. Create a directory that we'll use to mount to the Ceph File System

   ```
   mkdir /mnt/ceph
   ```

6. Edit /etc/fstab file (which is the OS's file system table) to add Ceph File System info

   ```
   vi /etc/fstab
   ```

   Add the following line right under the last line:

   ```
   101.102.1.10:/courses/cs5604 /mnt/ceph ceph name=cs5604,
   secretfile=/root/cephfs.secret,_netdev,noatime 0 0
   ```

7. Mount all filesystems mentioned in fstab as indicated

   ```
   mount -a
   ```

8. To check all mounted filesystems

   ```
   df -h
   ```

9. To check content of Ceph Storage (after adding content using Track changes is on 48 Kubectl)

   ```
   ls /mnt/ceph -l
   ```

10. To exit the VM and go back to your machine

    ```
    exit
    ```

7.2 Kubectl Installation and Introduction

In order for teams to be able to access the container cluster including Ceph storage (and avoiding Rancher limitations such as the 1 hour shell limit), kubectl needs to be installed on their machines. Using Rancher (CS cloud), we have set up containers for all of the other teams to mount data to Ceph storage.

To demonstrate, we have deployed a CentOS (named fe-centos) container on FE team's namespace (cs5604-fe-db). The following steps show how to install and work with Kubectl.

1. Install Kubectl:

   For Linux: (Download the latest release, make the kubectl binary executable, move the binary into your PATH)

   `https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-linux`

   For MacOS:

   `https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-macos`

   For Windows:

   `https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-windows`
   Alternate tip for Windows User: Kubectl can also be installed on the Windows subsystem for Linux.

2. kubectl looks for a file named config in the $HOME/.kube directory

   a. Create a directory that is named ".kube".
      ```
      mkdir  /.kube
      ```
   b. Create an empty file and call it "config".
      ```
      vi config
      ```
   c. Go to cloud.cs.vt.edu. Under the testing cluster, click on the Kubeconfig File (see Figure 29).
   d. You'll see the following page; click on "Copy to Clipboard" (see Figure 30).
   e. Return back to your computer's terminal, paste what you copied into the config file we created in step (b), and then save and close the file.

3. Move the config file we just created into the .kube directory.

   ```
   mv config  /.kube/
   ```

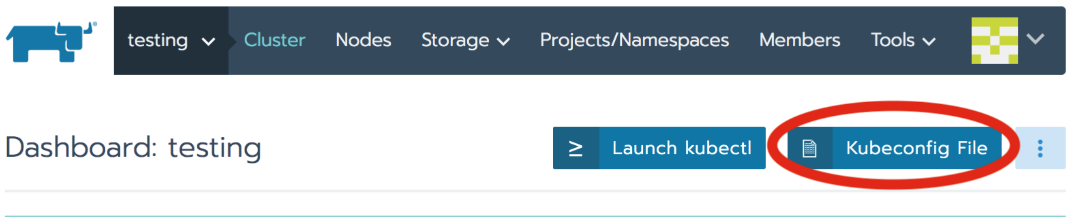


Figure 29: Kubectl configuration

4. Display the pods that are within the namespace cs5604-team-db. For example, for the FE team, the namespace is `cs5604-fe-db`. One can only run/get pods or deployments associated with their team's namespaces.

   `kubectl get pods -namespace cs5604-fe-db` (see Figure 31 for the output)

5. Now you can use kubectl to administer the deployments in your respective namespaces.

7.3 Deploying Containers from Rancher Catalogs

In this section, we will describe how to leverage Rancher's Catalogs and Apps to deploy containers seamlessly. To demonstrate this feature, we deploy a MySQL container as an example. With that said, Rancher hosts a wide variety of other applications including Elasticsearch, Kibana, Kubeflow, etc.

Catalogs are GitHub repositories or Helm Chart repositories filled with applications that are ready-made for deployment. Applications are bundled in objects called Helm charts [41].

1. Login to `cloud.cs.vt.edu` and navigate to the namespace where the application/container has to be deployed (see Figure 32).

2. Under the Apps tab click on launch (see Figure 33).

3. Search for a ready-made container that you want to deploy (see Figure 34).

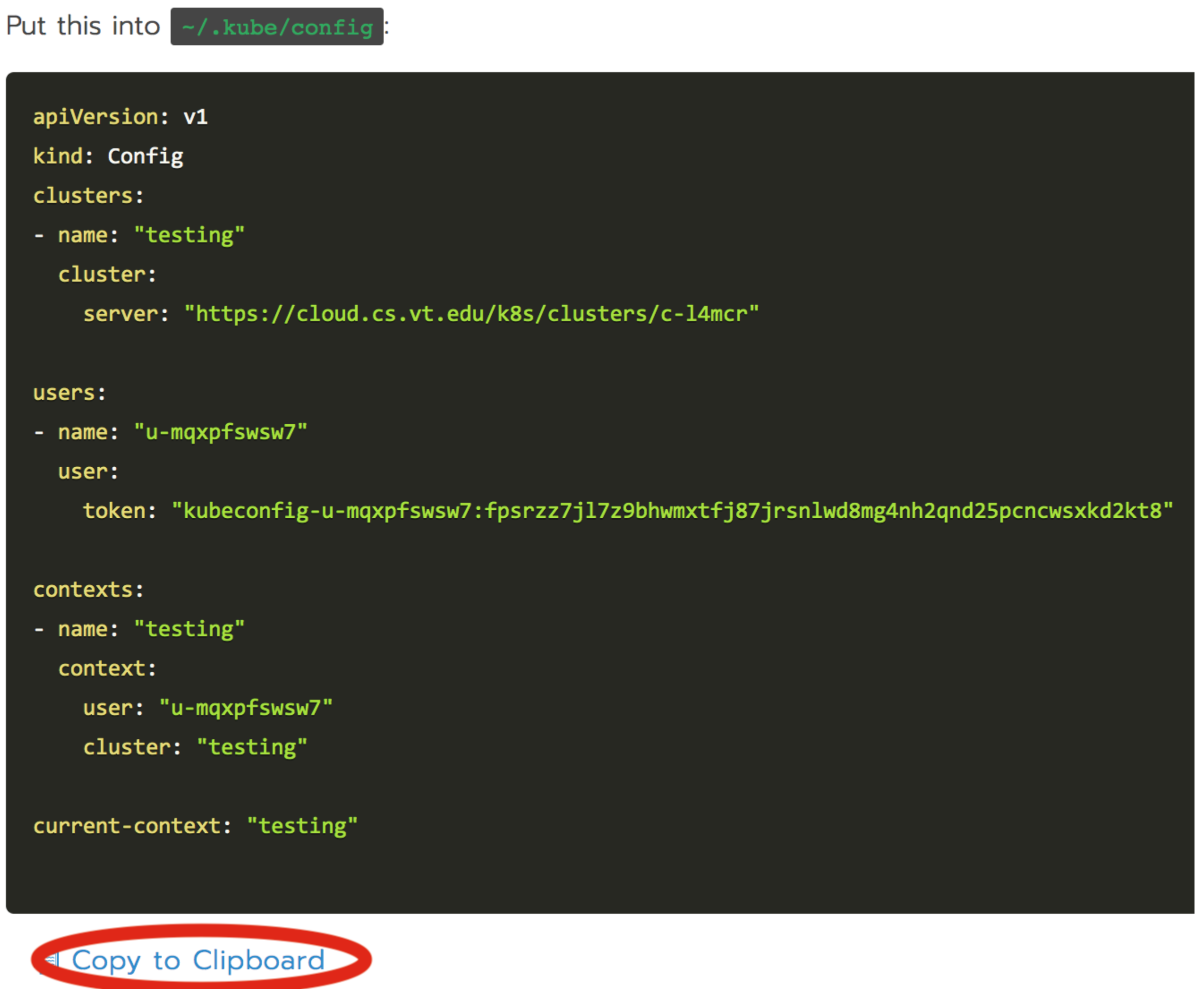
4. Configure your App with required parameters like namespace, environment variables, volume, etc. (see Figure 35).

5. Execute the shell of the newly deployed App and type the command to test if it works properly or not (see Figure 36).

7.4 Deploying Containers from Docker Hub

In this section, we will describe how to create a new container from Docker Hub on Rancher. To demonstrate this feature, we deploy a Python container as an example. With that said, we can find almost all of the containers we need in `hub.docker.com`. We can specify the version for any container.

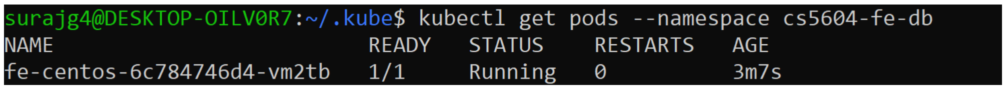
1. Open `hub.docker.com`, search for the container that you want to deploy, confirm the full name of the container and the version tag, for example, *python:latest* (see Figure 37).


Put this into `~/.kube/config`:

```yaml
apiVersion: v1
kind: Config
clusters:
- name: "testing"
  cluster:
    server: "https://cloud.cs.vt.edu/k8s/clusters/c-l4mcr"

users:
- name: "u-mqxpfswsw7"
  user:
    token: "kubeconfig-u-mqxpfswsw7:fpsrzz7jl7z9bhwmxtfj87jrsnlwd8mg4nh2qnd25pcncwsxkd2kt8"

contexts:
- name: "testing"
  context:
    user: "u-mqxpfswsw7"
    cluster: "testing"

current-context: "testing"
```

Figure 30: Kubeconfig File

2. Login to `cloud.cs.vt.edu` and navigate to the namespace where the application/container has to be deployed (see Figure 32).
3. Under the Workloads tab click on Deploy.
4. Configure your Container with required parameters like namespace, environment variables, volume, etc. (see Figure 35). It should be noted that the full name of the container and the version tag should be accurately typed into the **Docker Image** text-box (see Figure 38). Click **Launch** to finish your deployment.
5. Execute the shell of the newly deployed App and type the command to test if it works properly or not.



```
surajg4@DESKTOP-OILV0R7:~/.kube$ kubectl get pods --namespace cs5604-fe-db
NAME                     READY   STATUS    RESTARTS   AGE
fe-centos-6c784746d4-vm2tb   1/1     Running   0          3m7s
```

Figure 31: List of pods in the namespace `cs5604-fe-db`

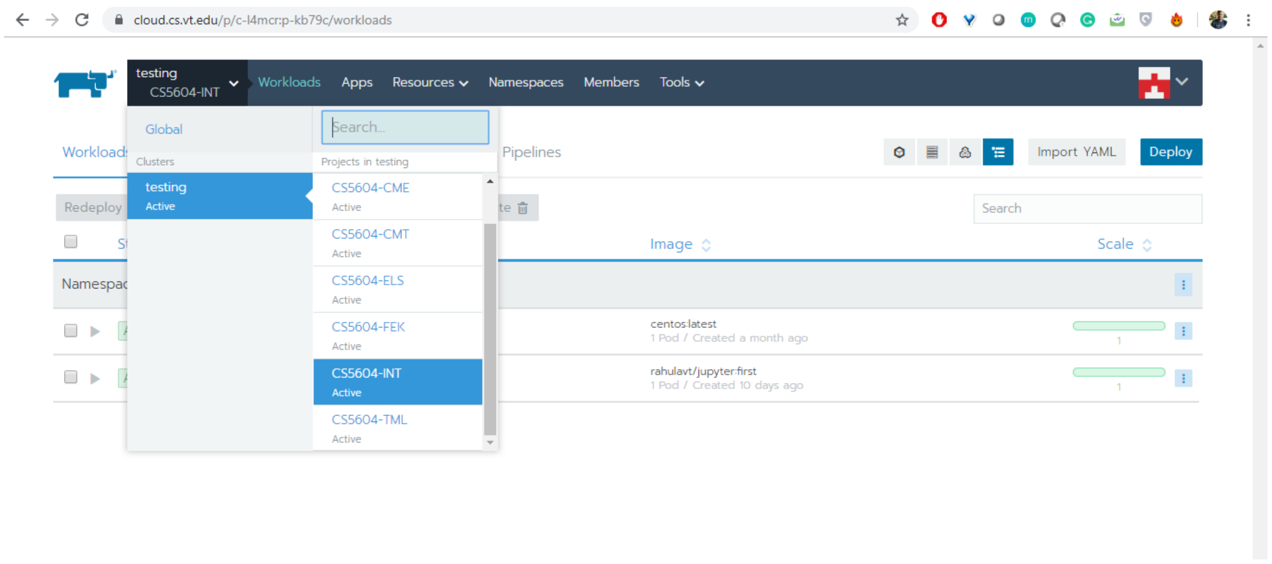


Figure 32: Navigating to CS5604-INT namespace

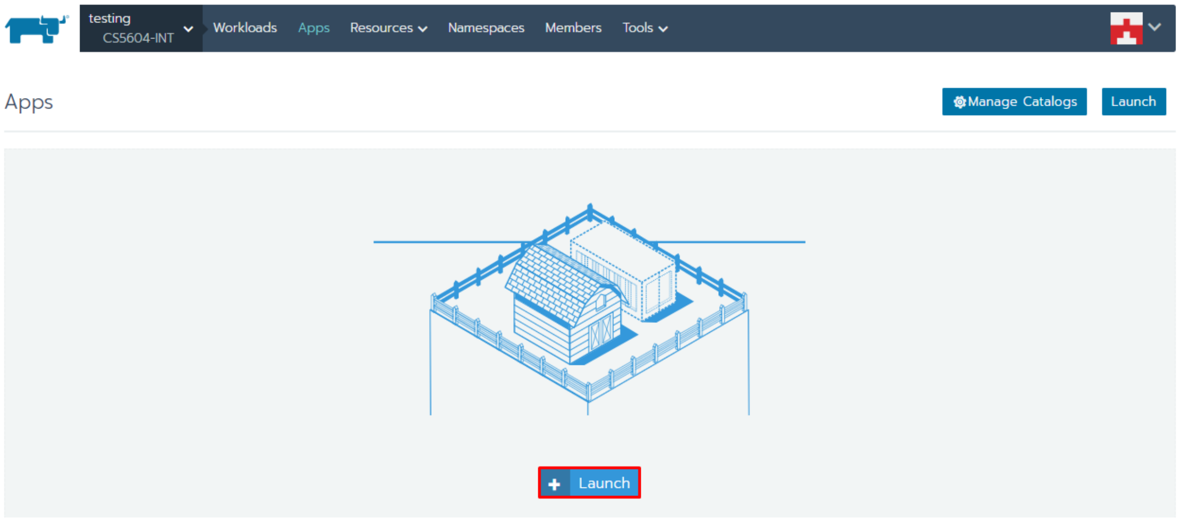


Figure 33: Launching a Rancher App

7.5 Configuring and Deploying Containers through Gitlab Container Registry

In this section, we will describe how to configure and deploy containers from Gitlab Container Registry on Rancher. For this, we will show how to generate deploy tokens on Gitlab, apply them on Rancher, and then deploy the container from the Gitlab container registry.

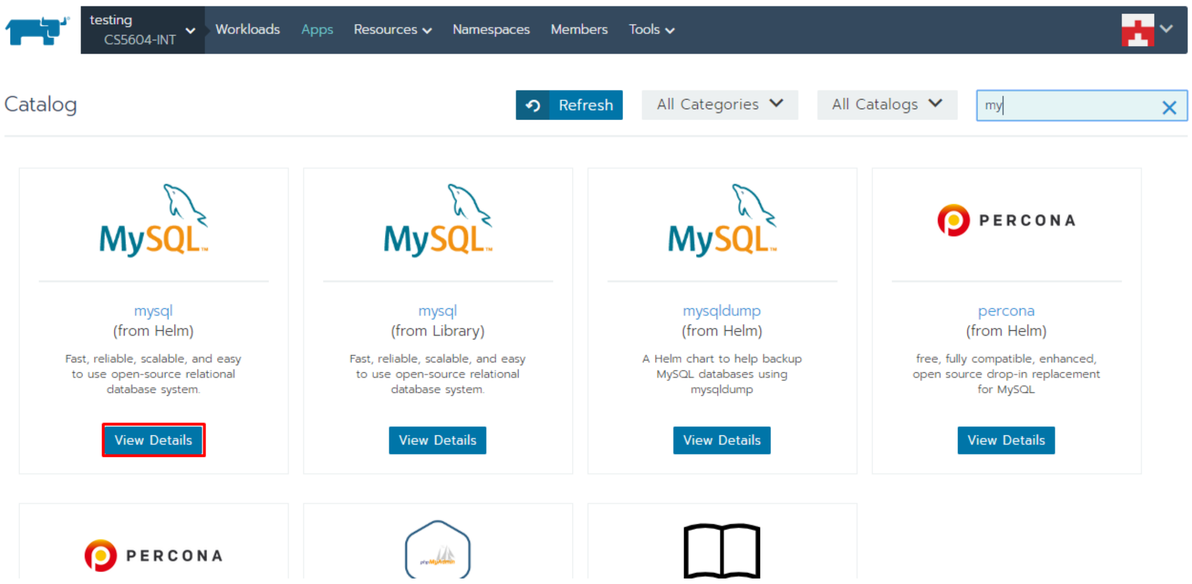
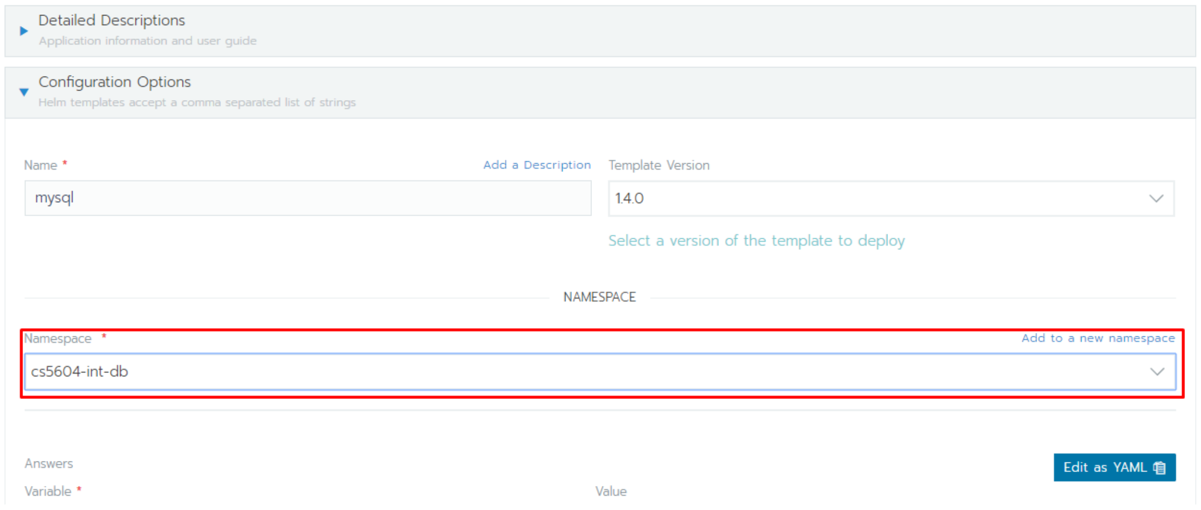

testing
CS5604-INT
Workloads
Apps
Resources
Namespaces
Members
Tools
Catalog
Refresh
All Categories
All Catalogs
my
mysql
(from Helm)
Fast, reliable, scalable, and easy to use open-source relational database system.
View Details
mysql
(from Library)
Fast, reliable, scalable, and easy to use open-source relational database system.
View Details
mysqldump
(from Helm)
A Helm chart to help backup MySQL databases using mysqldump
View Details
percona
(from Helm)
free, fully compatible, enhanced, open source drop-in replacement for MySQL
View Details

Figure 34: Launching MySQL App


Detailed Descriptions
Application information and user guide
Configuration Options
Helm templates accept a comma separated list of strings
Add a Description
Select a version of the template to deploy
NAMESPACE
Add to a new namespace
Answers
Edit as YAML
Variable *
Value

Name *: mysql

Template Version: 1.4.0

Namespace *: cs5604-int-db

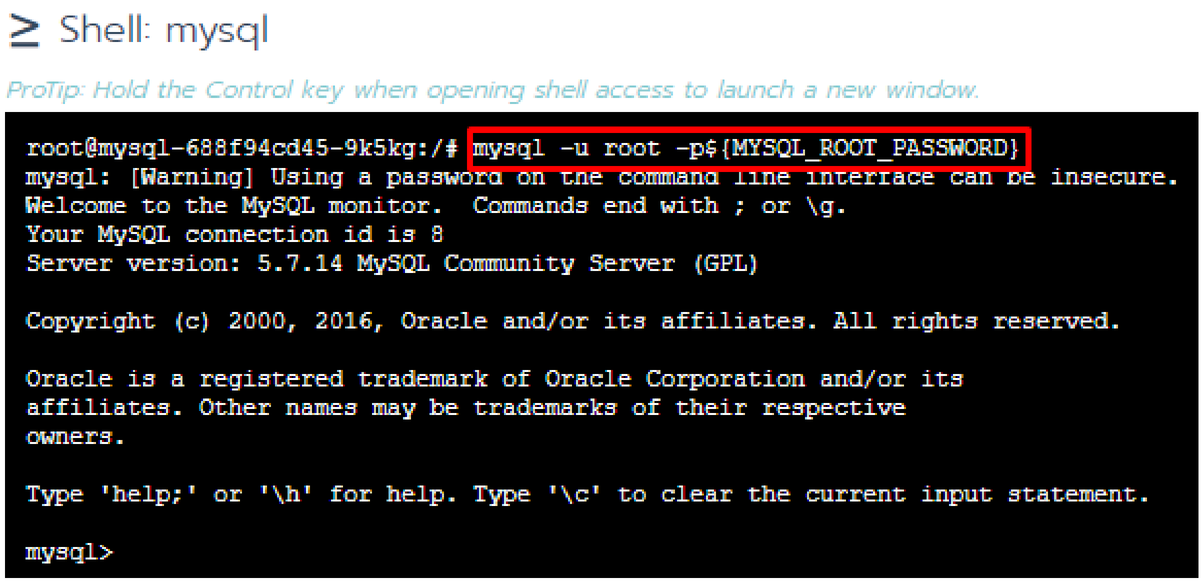
Figure 35: Configuring the App

1. The figure shows the Docker images present in the container registry present for the specific Gitlab repository (see Figure 39).

2. This point shows the step wise process for generating a deploy token on Gitlab. A deploy token is needed to be applied on Rancher so that any deployment through Gitlab is authenticated.

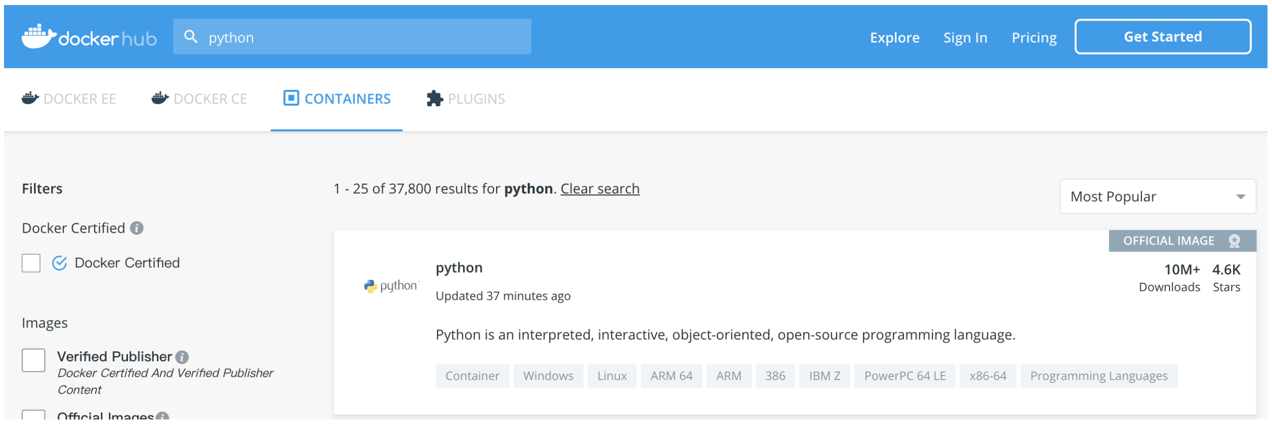
   [a.] Go to "Settings>Repository", expand "Deploy Tokens", and fill in the details such as Name, Expires at (can leave it blank to indicate "Never"), and the scope as required. Click on "Create deploy token" (see Figure 40).

   [b.] Once a deploy token is created, the Username and Password would get generated.


≥ Shell: mysql
ProTip: Hold the Control key when opening shell access to launch a new window.

```
root@mysql-688f94cd45-9k5kg:/# mysql -u root -p${MYSQL_ROOT_PASSWORD}
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Figure 36: Testing if MySQL App has been launched successfully



1 - 25 of 37,800 results for **python**. Clear search

python
Updated 37 minutes ago
10M+ Downloads 4.6K Stars

Python is an interpreted, interactive, object-oriented, open-source programming language.

Figure 37: Search for container on Docker Hub

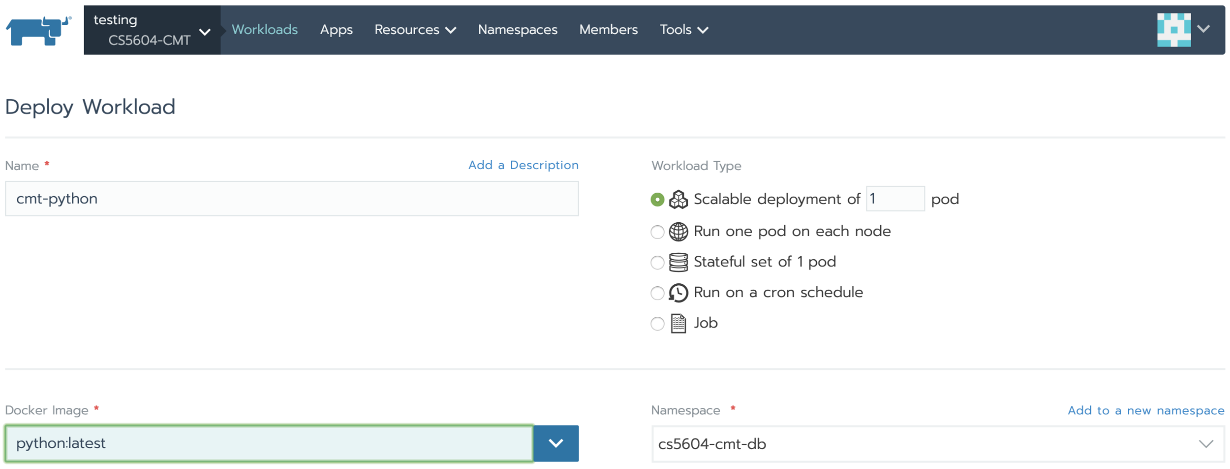
Save this Username and Password. This is generated only once during the creation of a deploy token, hence save this username and password. We would need this later (see Figure 41).

3. Login to cloud.cs.vt.edu, navigate to the respective namespace and "Resources>Registry Credentials". Click on "Add Registry" (see Figure 42).

4. Fill in the details for the deploy token (see Figure 43). Username and Password are the same that were generated in the previous step.

5. For the deployment follow initial steps as explained in Section 7.4 until step 3. Fill in the details as shown in Figure 44. Note that the Docker image here is the name of the image present in the Gitlab registry with its tag. Click Launch.

6. The launched container ("centostest") is now available under the namespace (see Figure 45).



Deploy Workload

Name * cmt-python

Add a Description

Workload Type

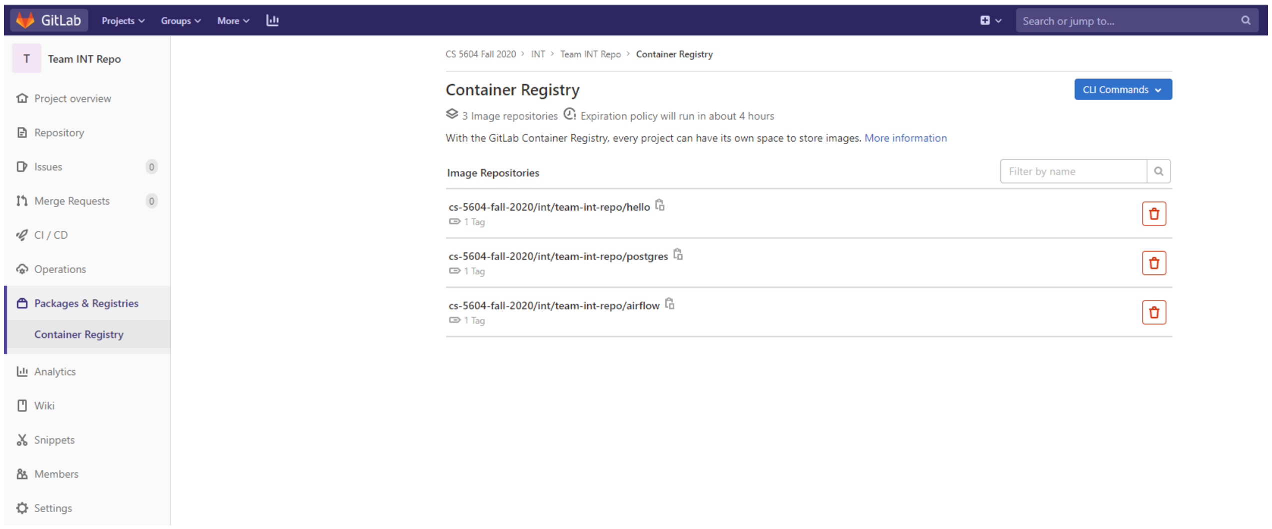
- [✓ Scalable deployment of 1 pod]
- Run one pod on each node
- Stateful set of 1 pod
- Run on a cron schedule
- Job

Docker Image * python:latest

Namespace * cs5604-cmt-db

Add to a new namespace

Figure 38: Search container on Docker Hub



CS 5604 Fall 2020 › INT › Team INT Repo › Container Registry

Container Registry

3 Image repositories Expiration policy will run in about 4 hours

With the GitLab Container Registry, every project can have its own space to store images. More information

Image Repositories

Filter by name

cs-5604-fall-2020/int/team-int-repo/hello
1 Tag

cs-5604-fall-2020/int/team-int-repo/postgres
1 Tag

cs-5604-fall-2020/int/team-int-repo/airflow
1 Tag

Figure 39: Gitlab Container Registry

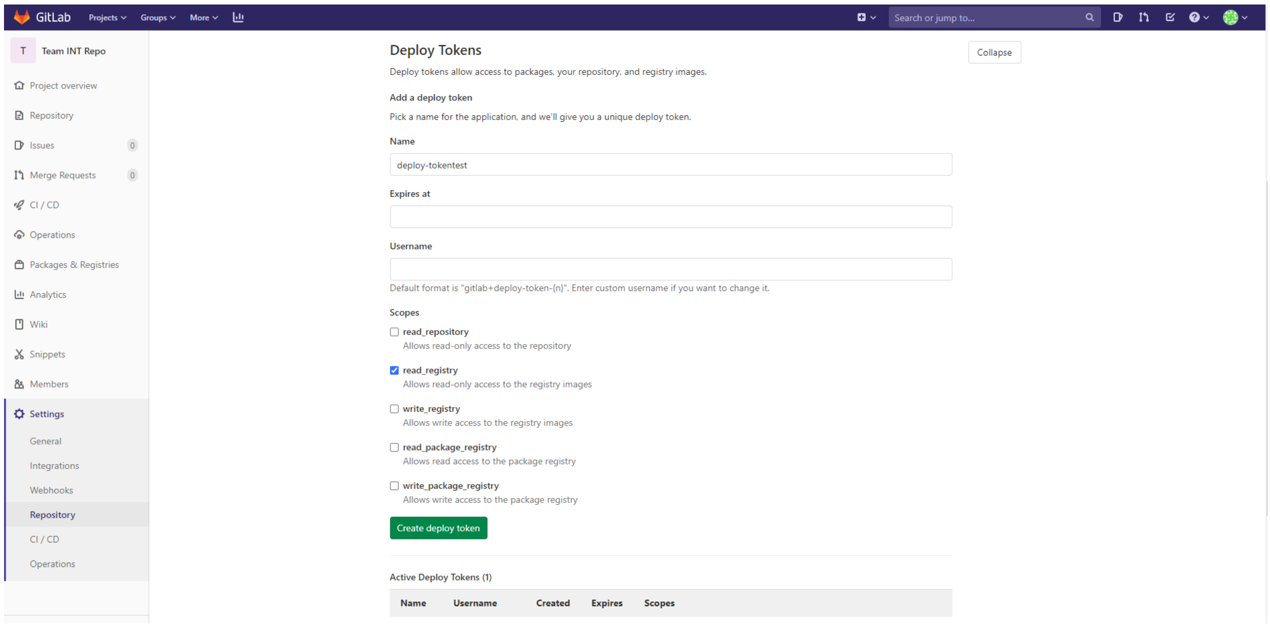
7.6 Mounting Ceph storage on containers

In this section, we will describe how to add the Ceph storage while creating a new container from Docker Hub on Rancher. To demonstrate, we deploy a CentOS container as an example. Find the CentOS container on Docker Hub. Note its details. The initial configuration is similar to steps 1-4 in Section 7.4 until the 'Launch'. Now, before launching, in order to add the Ceph storage, we will perform the following steps:

1. Scroll down and click on Select Volumes-Add Volume (see Figure 46). In the Add Volume drop down - choose 'Add an ephemeral volume' (see Figure 47).

2. In the ephemeral volume portal - In the 'Source' drop down, select "Ceph Filesystem" (see Figure 48) and define the remaining configurations as shown in Figure 49.

3. In case Step 2 doesn't show 'Ceph Filesystem' in the drop down (this is possible as Rancher



Deploy Tokens

Deploy tokens allow access to packages, your repository, and registry images.

Add a deploy token

Pick a name for the application, and we'll give you a unique deploy token.

Name

deploy-tokentest

Expires at

Username

Default format is "gitlab+deploy-token-{n}". Enter custom username if you want to change it.

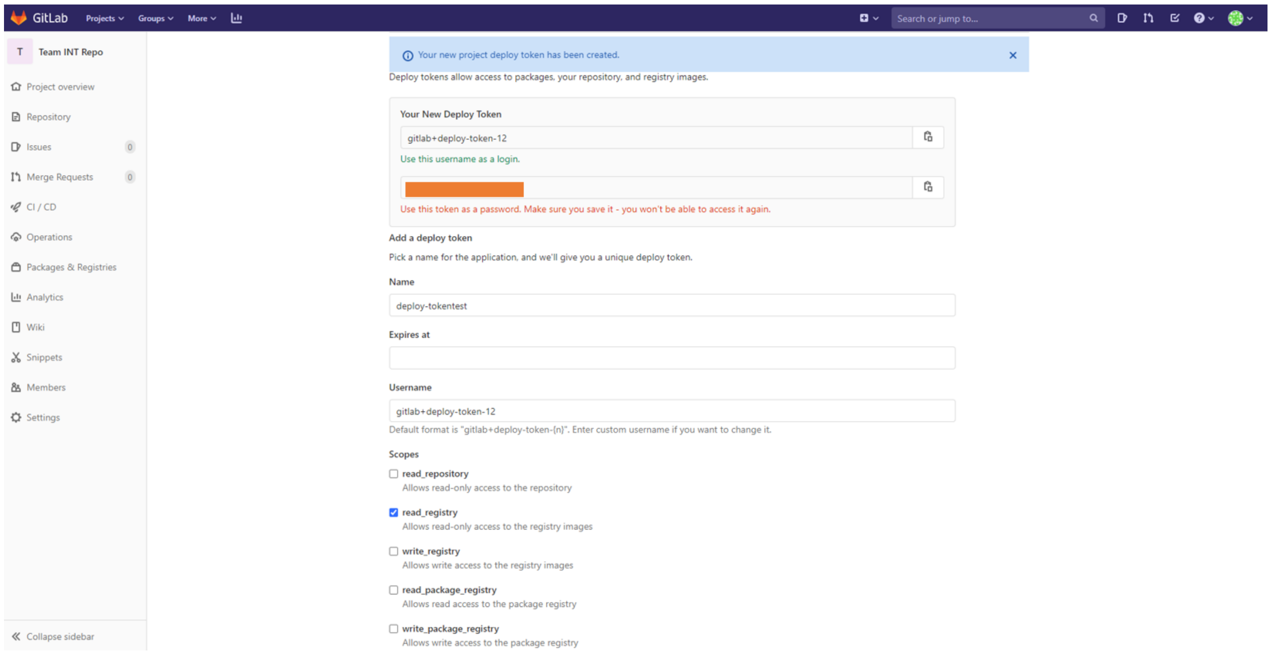
Scopes

- [ ] read_repository
  Allows read-only access to the repository
- [x] read_registry
  Allows read-only access to the registry images
- [ ] write_registry
  Allows write access to the registry images
- [ ] read_package_registry
  Allows read access to the package registry
- [ ] write_package_registry
  Allows write access to the package registry

Active Deploy Tokens (1)

Name	Username	Created	Expires	Scopes

Figure 40: Filling deploy token details



Your new project deploy token has been created.

Deploy tokens allow access to packages, your repository, and registry images.

Your New Deploy Token

gitlab+deploy-token-12

Use this username as a login.

[redacted]

Use this token as a password. Make sure you save it - you won't be able to access it again.

Add a deploy token

Pick a name for the application, and we'll give you a unique deploy token.

Name

deploy-tokentest

Expires at

Username

gitlab+deploy-token-12

Default format is "gitlab+deploy-token-{n}". Enter custom username if you want to change it.

Scopes

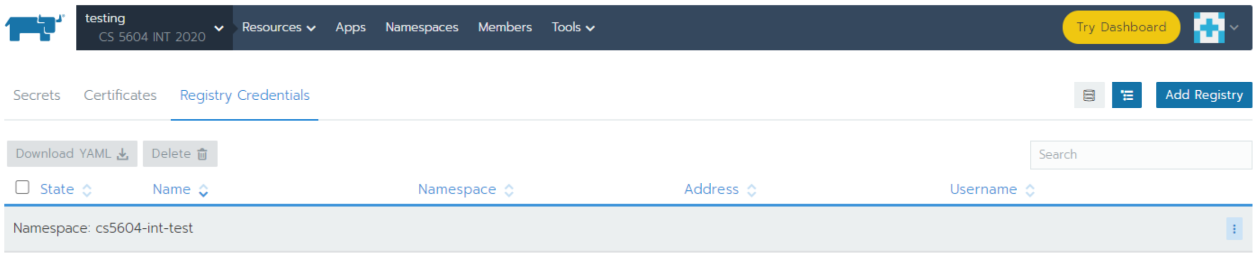
- [ ] read_repository
  Allows read-only access to the repository
- [x] read_registry
  Allows read-only access to the registry images
- [ ] write_registry
  Allows write access to the registry images
- [ ] read_package_registry
  Allows read access to the package registry
- [ ] write_package_registry
  Allows write access to the package registry

Figure 41: Deploy Token Username and Password

may remove it from its interface), then one can also do this by directly modifying the YAML file. For this, navigate to the respective namespace where the pod has been deployed. Click the "..." button on the right side of the container and select 'View/Edit YAM' (see Figure 50).

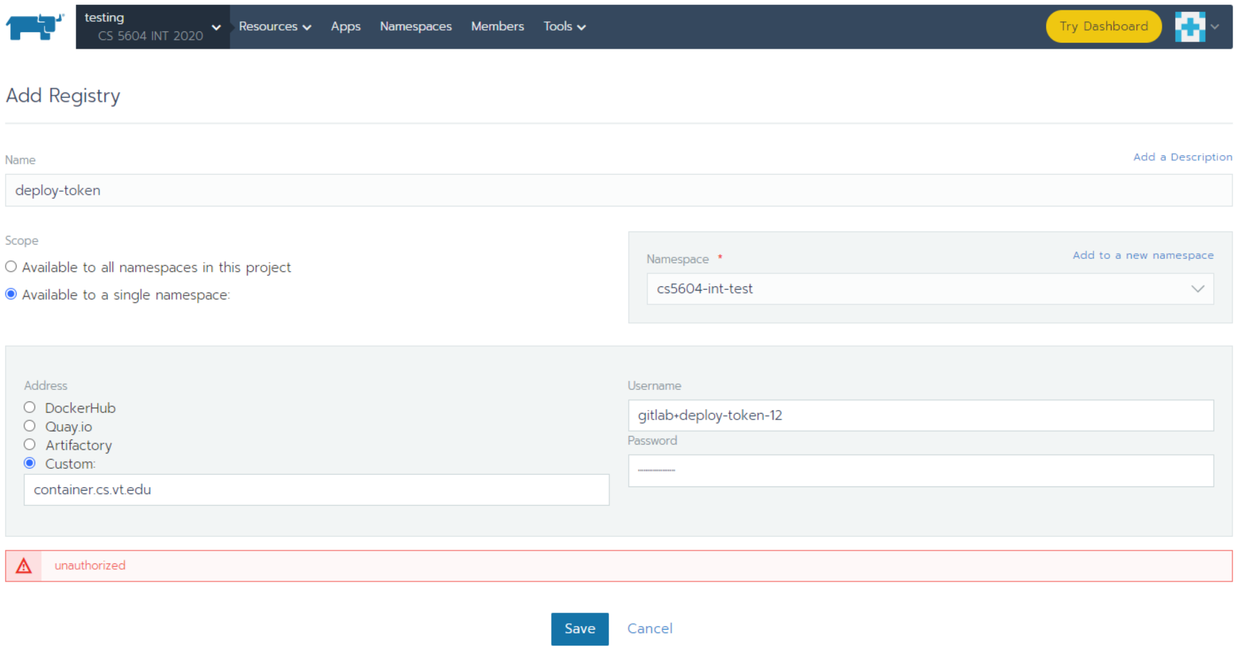
4. One can modify the YAML as follows: This will mount Ceph to the container.

```
...
```



Namespace: cs5604-int-test

Figure 42: Registry Credentials on Rancher



Add Registry

Name: deploy-token

Scope
- Available to all namespaces in this project
- [✓ Available to a single namespace:]

Namespace *: cs5604-int-test

Address
- DockerHub
- Quay.io
- Artifactory
- [✓ Custom:] container.cs.vt.edu

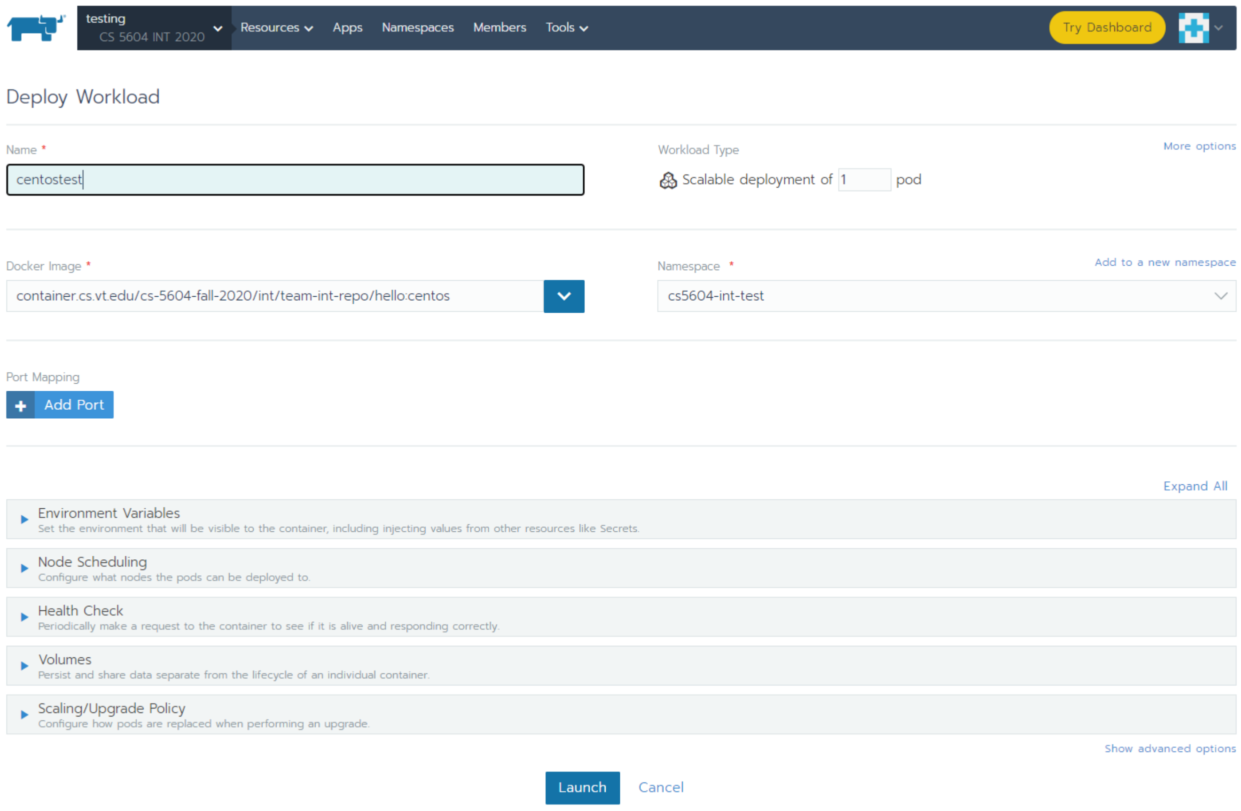
Username: gitlab+deploy-token-12

Password: ············

unauthorized

Figure 43: Adding deploy token details

```
    spec:
      containers:
      - image: centos:latest
...
          volumeMounts:
          - mountPath: /mnt/ceph
            name: vol1
...
        volumes:
        - cephfs:
            monitors:
            - 128.173.41.10:6789
            - 128.173.41.11:6789
            - 128.173.41.12:6789
            path: /courses/cs5604
```



Deploy Workload

Name * centostest

Workload Type: Scalable deployment of 1 pod

Docker Image * container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/hello:centos

Namespace * cs5604-int-test

Port Mapping

Environment Variables
Set the environment that will be visible to the container, including injecting values from other resources like Secrets.

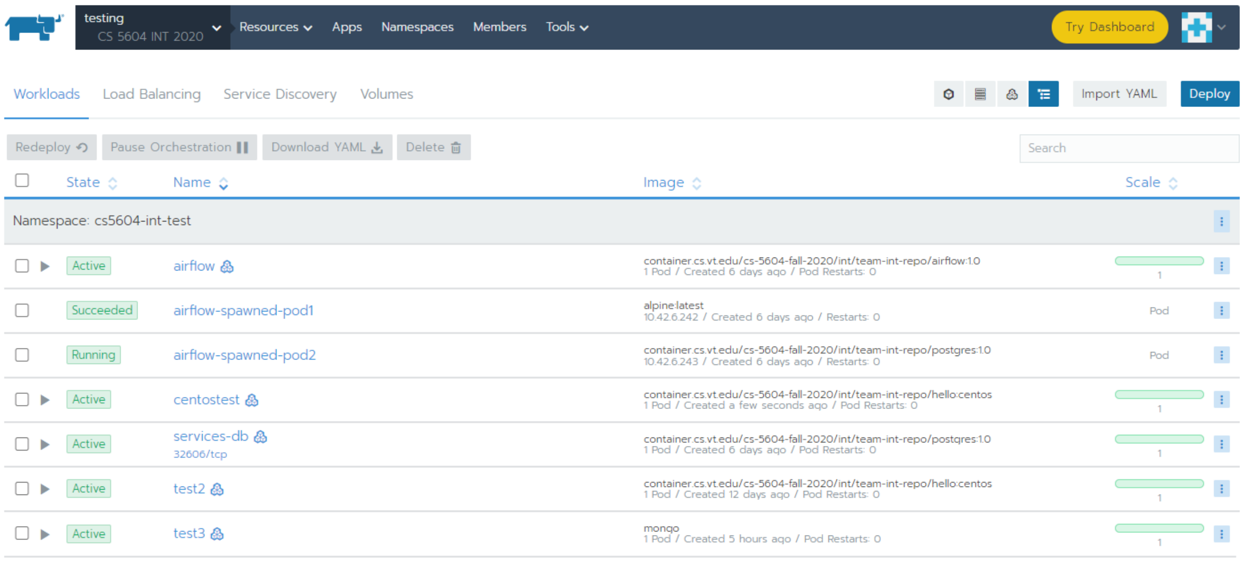
Node Scheduling
Configure what nodes the pods can be deployed to.

Health Check
Periodically make a request to the container to see if it is alive and responding correctly.

Volumes
Persist and share data separate from the lifecycle of an individual container.

Scaling/Upgrade Policy
Configure how pods are replaced when performing an upgrade.

Figure 44: Deploying container from Gitlab Container Registry

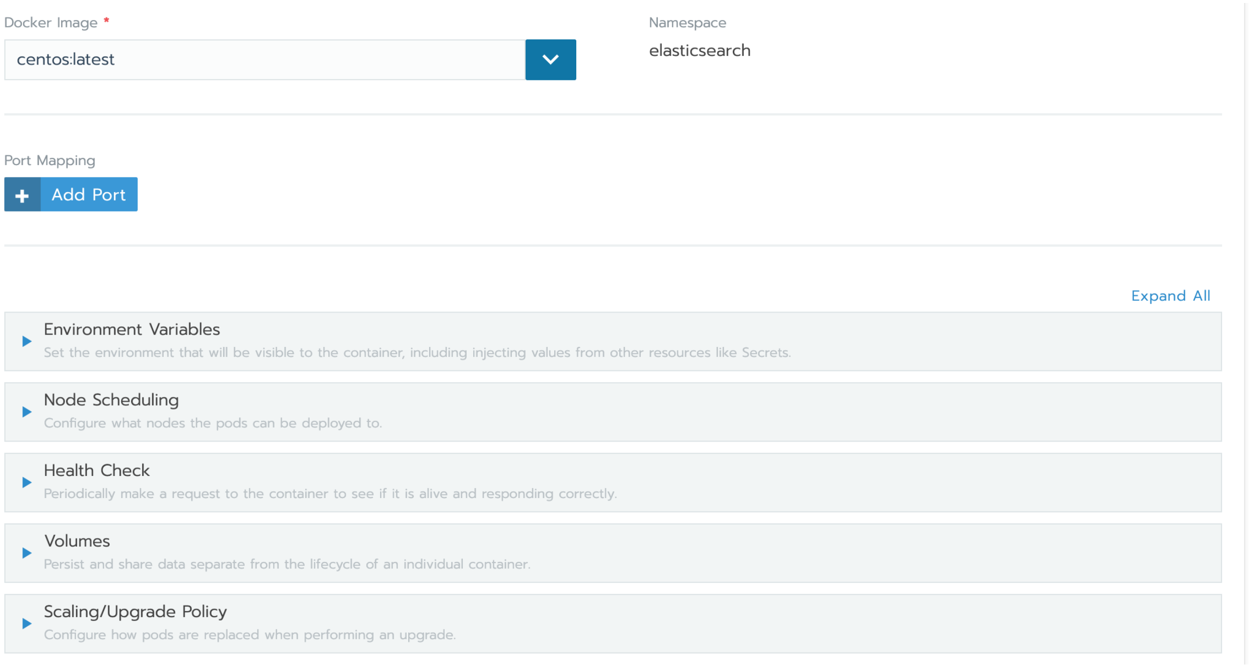


Namespace: cs5604-int-test

State	Name	Image	Scale
Active	airflow	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/airflow:1.0 1 Pod / Created 6 days ago / Pod Restarts: 0	1
Succeeded	airflow-spawned-pod1	alpine:latest 10.42.6.242 / Created 6 days ago / Restarts: 0	Pod
Running	airflow-spawned-pod2	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/postgres:1.0 10.42.6.243 / Created 6 days ago / Restarts: 0	Pod
Active	centostest	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/hello:centos 1 Pod / Created a few seconds ago / Pod Restarts: 0	1
Active	services-db 32606/tcp	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/postgres:1.0 1 Pod / Created 6 days ago / Pod Restarts: 0	1
Active	test2	container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/hello:centos 1 Pod / Created 12 days ago / Pod Restarts: 0	1
Active	test3	mongo 1 Pod / Created 5 hours ago / Pod Restarts: 0	1

Figure 45: Deployed Container "centostest" present under the namespace

```
secretRef:
  name: ceph-cs5604
```


Docker Image *

centos:latest

Namespace

elasticsearch

Port Mapping

+ Add Port

Expand All

Environment Variables
Set the environment that will be visible to the container, including injecting values from other resources like Secrets.

Node Scheduling
Configure what nodes the pods can be deployed to.

Health Check
Periodically make a request to the container to see if it is alive and responding correctly.

Volumes
Persist and share data separate from the lifecycle of an individual container.

Scaling/Upgrade Policy
Configure how pods are replaced when performing an upgrade.

Figure 46: Configure Docker image details on deploy portal

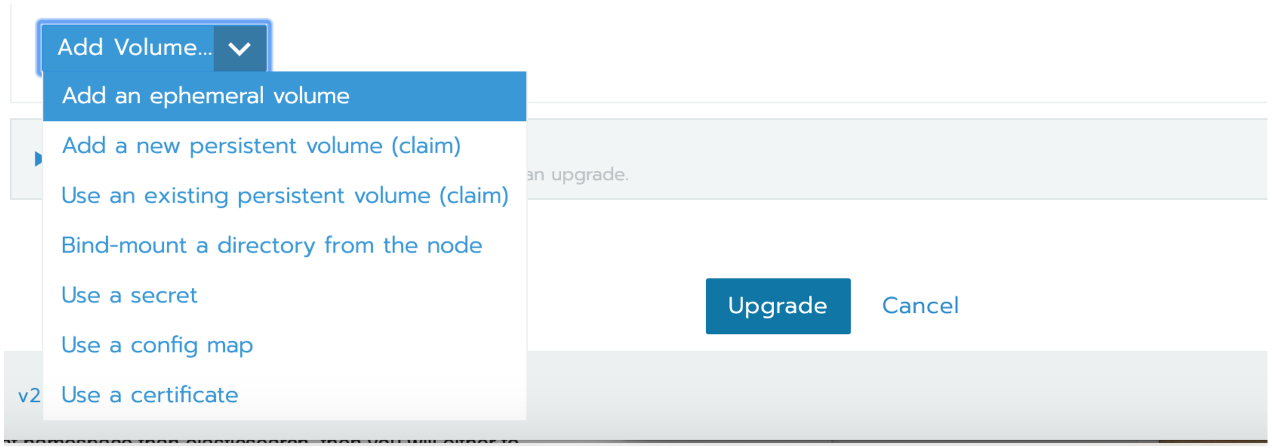

Add Volume...
Add an ephemeral volume
Add a new persistent volume (claim)
Use an existing persistent volume (claim)
Bind-mount a directory from the node
Use a secret
Use a config map
Use a certificate
an upgrade.
Upgrade
Cancel
v2

Figure 47: Select Volumes and Add Volume to mount storage on container

```
          user: cs5604
        name: vol1

"mountPath" is where it will be mounted inside the container.
"monitors" are the IPs and ports for our Ceph monitor servers.
"path" is the CephFS subpath to mount; leave this as "/courses/cs5604".
"secretRef" is the *name* of the secret that contains the security key.
"user" is the CephFS user to authorize; leave it as "cs5604".
```

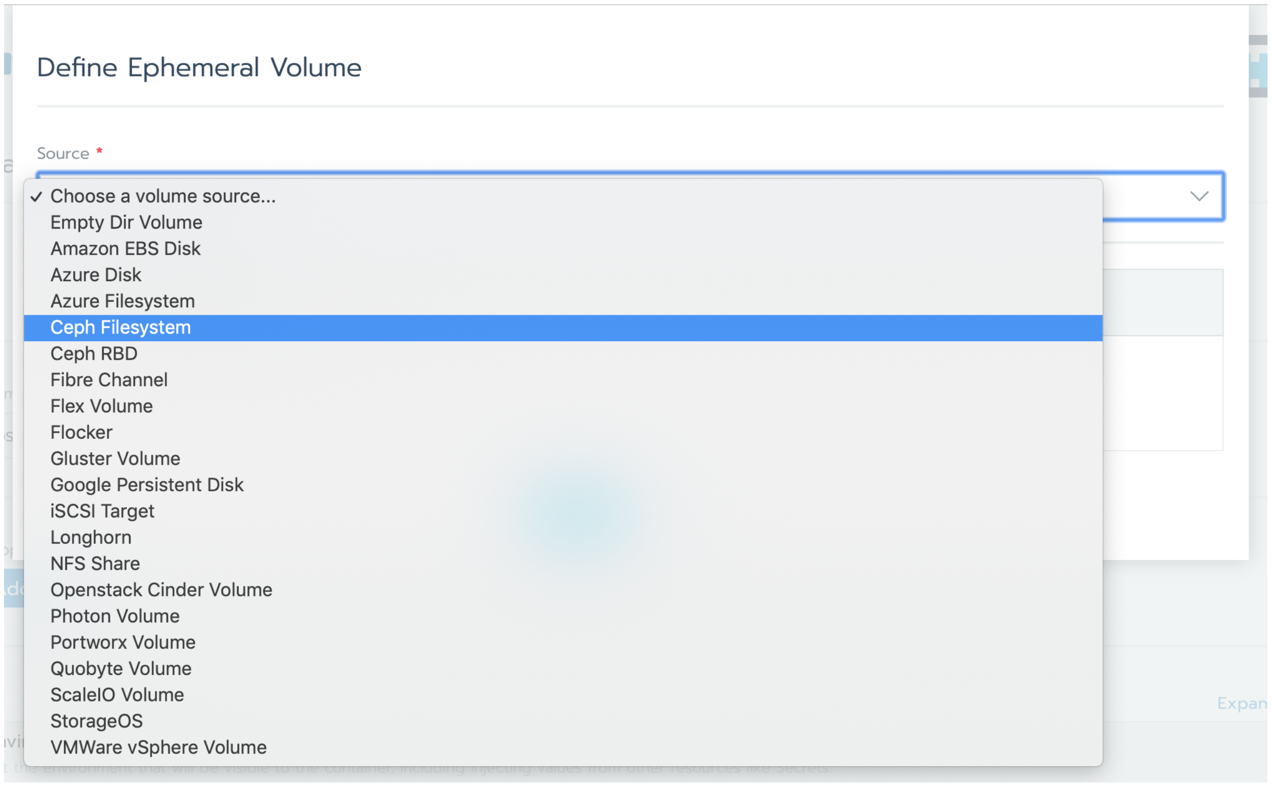


Figure 48: Select 'Ceph Filesystem' as the Source on Ephemeral Volume portal'

5. Now execute the shell and one should be able to access the Ceph storage at /mnt/ceph

6. In this case, the pods are not running and Ceph is not getting mounted after editing the YAML file. Then there's a possibility that the Ceph storage is not authenticated. For this, one needs to add the authentication key in "Resources>Secrets>Secrets" (see Figure 51).

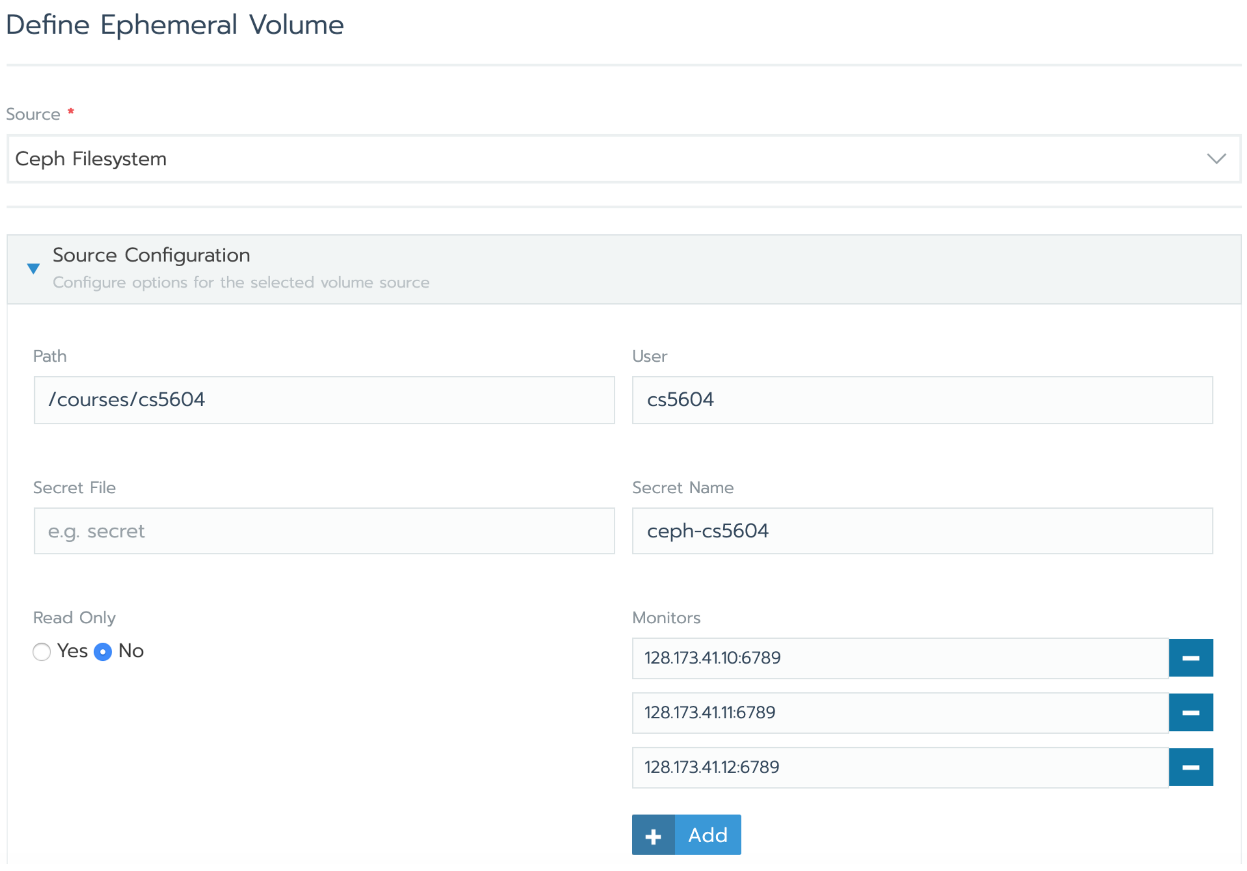
7.7 Running Airflow on the Kubernetes cluster

7.7.1 Setting up the container

Airflow was set up on a Kubernetes pod using the official Docker image [23]. Airflow requires a lot of configuration, two main ones being - a URL to the database that would store metadata and the type of executor that would be used (more about this in Section 4.7). The configuration could be done in the following two ways.

- Through code - Python has a great library called `ConfigParser`; a small script could be made to set desired values to fields in the configuration file. This script would have to be invoked every time the container is instantiated.

- Through environment (`ENV`) variables - This method was chosen as Airflow inherently supports such a mechanism and setting up `ENV` variables is extremely easy through the `Dockerfile`. This method is also suggested in the official documentation [25].

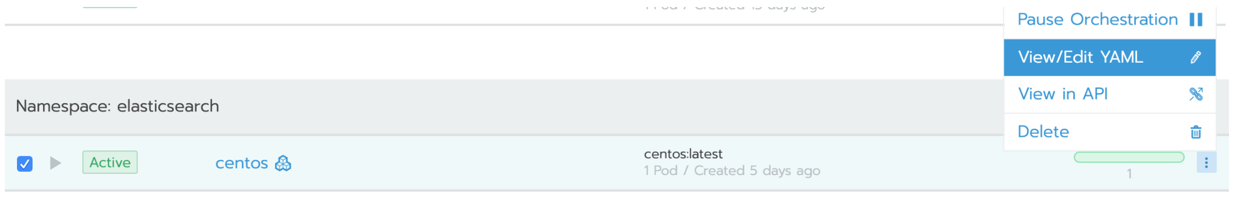

Define Ephemeral Volume

Source *: Ceph Filesystem

Source Configuration
Configure options for the selected volume source

Path	/courses/cs5604
User	cs5604
Secret File	e.g. secret
Secret Name	ceph-cs5604
Read Only	Yes [✓ No]
Monitors	128.173.41.10:6789
	128.173.41.11:6789
	128.173.41.12:6789

Figure 49: Enter Configuration details for the Ceph Filesystem on Ephemeral Volume portal


Namespace: elasticsearch

Active centos centos:latest 1 Pod / Created 5 days ago

Figure 50: Snapshot of a pod under a namespace and "View/Edit YAML" button

Once the `Dockerfile` (accessible at our Gitlab repository [54]) was set up with the needed `ENV` variables, we built an HTTP API for running Airflow commands. To do so, we used a Python library called `FlaskShell2HTTP`. The library can be used to make web API endpoints for shell commands. We set up 3 endpoints - one to upload a dynamically generated Airflow document, one to run a workflow, and an endpoint to run a service from a given workflow. (Refer to Section 6.4 for usage instructions.)

Finally, an image was built and pushed to Virginia Tech's image registry using the following.

```
docker build -t
container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/
airflow:latest .
```

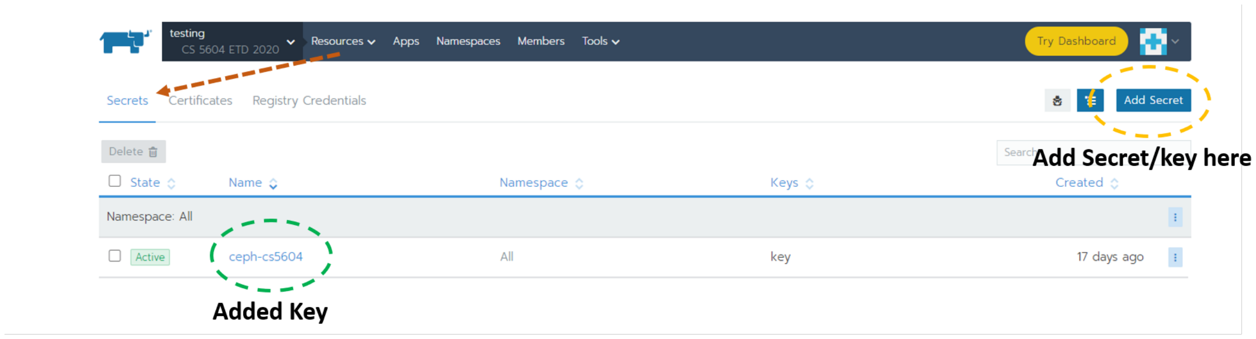


Figure 51: Adding Ceph Secret on Rancher

```
docker push
container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/
airflow:latest
```

After the image was registered, it was deployed on `cloud.cs.vt.edu` through the Rancher UI; see Section 7.5. Note that once the API is deployed, the following command needs to be run through the 'Execute Shell' option on Rancher:

```
airflow db init
```

The command needs to be run only once and is used to connect the Airflow instance with the Postgres database.

7.7.2 Issues faced

To test the deployment, a simple workflow was developed consisting of two tasks and stored in `test_dag.py` (accessible at our Gitlab repository [54]). This file was copied into the container's dags folder (default location `/opt/airflow/dags`). The main aim of this task was to see if Airflow could indeed spin up new pods in the cluster to complete each task in the workflow. This step was very troublesome as the `default` user in any `namespace` in `cloud.cs.vt.edu` does not have enough permissions to do so. To solve this issue, the `kubeconfig` was copied onto `/opt/airflow/` using the `COPY` command on Docker. Within the `test_dag.py` file pods were deployed using the `KubernetesPodOperator`; this operator takes an argument `config_file` which should point to the `kubeconfig` in the container (in our case, `/opt/airflow/kubeconfig`). Do not forget to set the `in_cluster` argument of the operator to `False`, otherwise it reads the default configuration file from the cluster.

Another issue we faced was to define a shell script for spinning up the Airflow API and to initialize the connection between Airflow and Postgres. We hoped to call the shell script during the initialization of the Docker container, however, both the commands were not running for unknown reasons. Hence we decided to run the API through the Dockerfile and then run the `airflow db init` command manually.

Finally, Airflow (v2.0.0) was having trouble running tasks with more than 3-4 services and we have raised an issue on their GitHub [56]. The parser they are using to read logs from Kubernetes seems to be causing them issues. There has been no response from their side, but we have created a small patch to use locally in our project.

7.8 Running Postgres on the Kubernetes cluster

7.8.1 Setting up the container

Postgres was set up on a Kubernetes pod as well using the official Docker image [24]. Postgres is being used as our database for the services available as well as holding the configuration and data for Airflow. A `Dockerfile` (accessible at our Gitlab repository [55]) was created that would pull the official image and run a few commands to automatically set up three databases (one for Airflow, one for the service registry, and one for the frontend) within the pod. These commands were executed using the `RUN` command supported in Docker. Users were also created for each database to ensure that each service only has access to the appropriate database. An example command used for setting up a database and creating a user is as follows.

```
psql -command "create database airflow;"
psql -command "create user airflow with encrypted password 'xxxx';"
```

Finally, an image was built and pushed to Virginia Tech's image registry using the following commands.

```
docker build -t
container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/
postgres:latest .

docker push
container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/
postgres:latest
```

After the image was registered, it was deployed on `cloud.cs.vt.edu` through the Rancher UI. Refer to Section 7.5 for more details.

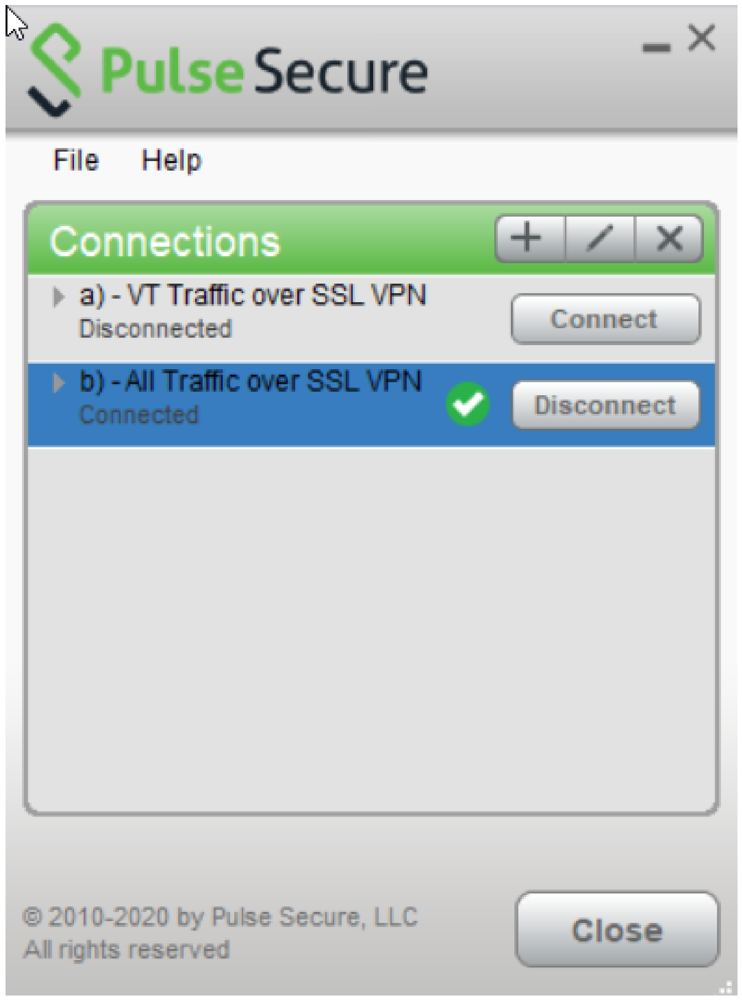
7.9 Accessing Elasticsearch through the Elasticsearch VM

In order to be able to connect to the Multi-Node Elasticsearch service running in Docker containers on the Elasticsearch VM, a developer has to be installed on the Elasticsearch.cs.vt.edu VM either directly from the VT network or by VPN and through SSH, through Putty.

The login information is an example. The actual connection process has been provided directly to the teams.

1. Using the terminal on your machine, connect to the VM (an Ubuntu machine).

2. The following steps are executed in the VM:



a) - VT Traffic over SSL VPN
Disconnected

b) - All Traffic over SSL VPN
Connected

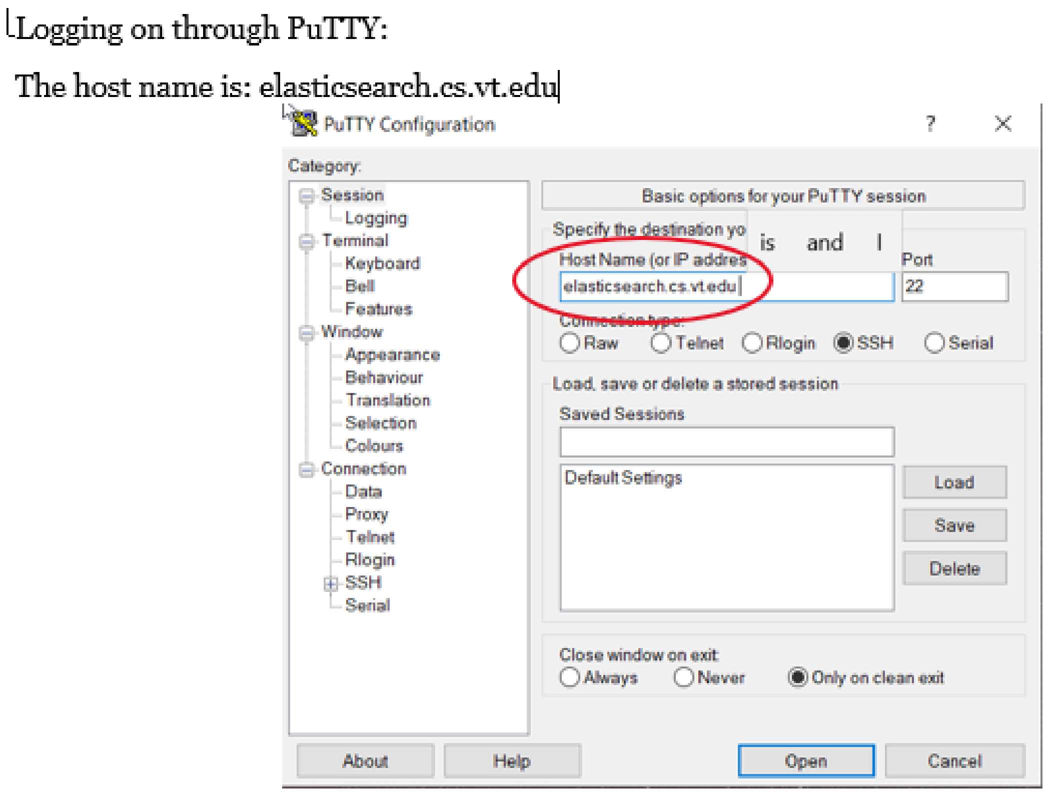
© 2010-2020 by Pulse Secure, LLC
All rights reserved

Figure 52: VPN Connection

3. Curl to the VM local host or access through a web browser. The addresses have previously been provided to the teams.
4. Once connected to the Elasticsearch VM, connect to the Elasticsearch Nodes by curling to the cat/indices?v (previously provided)
5. Begin uploading documents and indexing according to the individual team plan.

7.9.1 Issues faced

To test the deployment, a simple shell was executed on the running Postgres pod. Using this, we checked if the databases were set up correctly within Postgres. After this was confirmed, we tried to set up a connection between the Airflow pod and the Postgres pod. We faced problems here because we were hard coding the IP address of the Postgres pod into the Airflow configuration. This was an issue because every time we spin up the Postgres pod, it would obtain a new IP address which would in turn requires us to modify the Airflow deployment. We managed to fix this by accessing pods using service names (see Section 7.10). After this was done, we ran the `airflow`



Logging on through PuTTY:

The host name is: elasticsearch.cs.vt.edu

PuTTY Configuration

Category:

Basic options for your PuTTY session

Specify the destination yo is and I

Host Name (or IP addres: elasticsearch.cs.vt.edu

Port: 22

Connection type: ( ) Raw ( ) Telnet ( ) Rlogin (● SSH) ( ) Serial

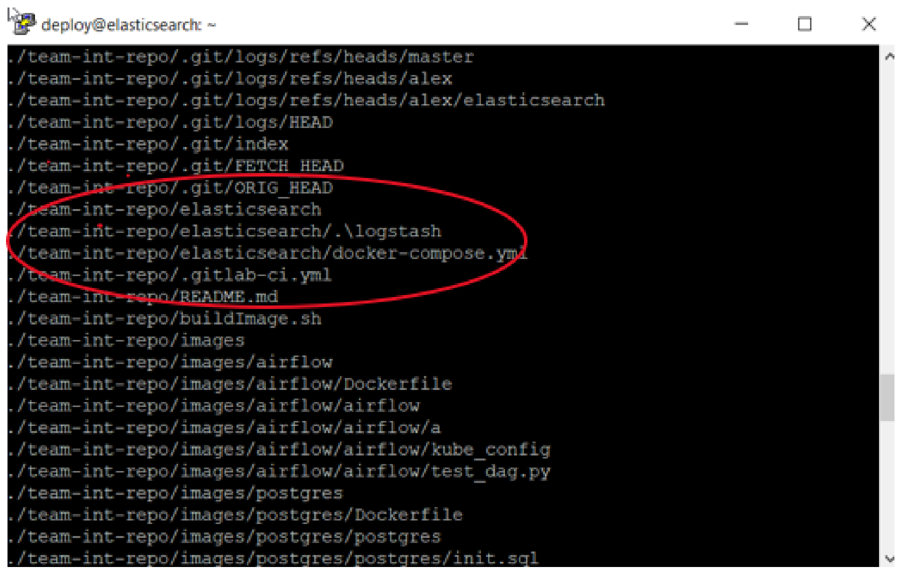
Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: ( ) Always ( ) Never (● Only on clean exit)

Figure 53: SSH through Putty

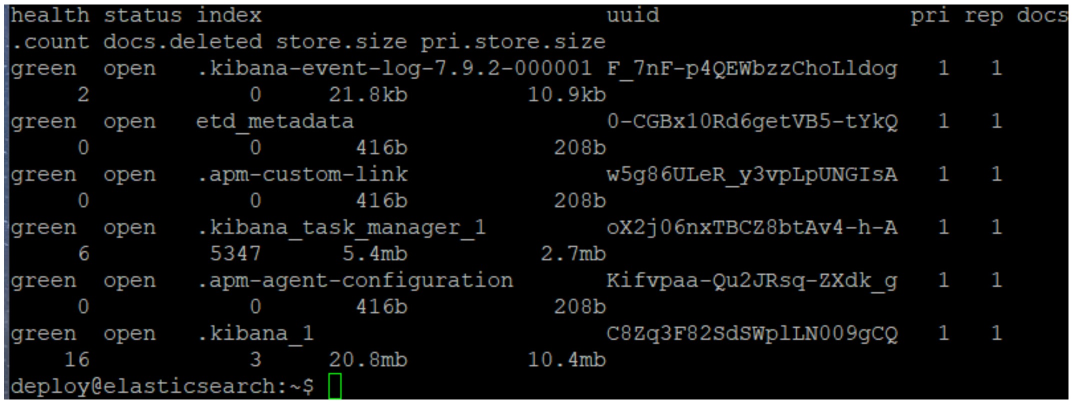


deploy@elasticsearch: ~

```
./team-int-repo/.git/logs/refs/heads/master
./team-int-repo/.git/logs/refs/heads/alex
./team-int-repo/.git/logs/refs/heads/alex/elasticsearch
./team-int-repo/.git/logs/HEAD
./team-int-repo/.git/index
./team-int-repo/.git/FETCH_HEAD
./team-int-repo/.git/ORIG_HEAD
./team-int-repo/elasticsearch
./team-int-repo/elasticsearch/.\logstash
./team-int-repo/elasticsearch/docker-compose.yml
./team-int-repo/.gitlab-ci.yml
./team-int-repo/README.md
./team-int-repo/buildImage.sh
./team-int-repo/images
./team-int-repo/images/airflow
./team-int-repo/images/airflow/Dockerfile
./team-int-repo/images/airflow/airflow
./team-int-repo/images/airflow/airflow/a
./team-int-repo/images/airflow/airflow/kube_config
./team-int-repo/images/airflow/airflow/test_dag.py
./team-int-repo/images/postgres
./team-int-repo/images/postgres/Dockerfile
./team-int-repo/images/postgres/postgres
./team-int-repo/images/postgres/postgres/init.sql
```

Figure 54: VM Directory for Elasticsearch

`db init` command from the Airflow pod. This command generated about 20 tables successfully on the `airflow` database in the Postgres pod.



```
health status index                             uuid                   pri rep docs
.count docs.deleted store.size pri.store.size
green  open   .kibana-event-log-7.9.2-000001 F_7nF-p4QEWbzzChoLldog   1   1
     2            0      21.8kb        10.9kb
green  open   etd_metadata                    0-CGBx10Rd6getVB5-tYkQ   1   1
     0            0        416b          208b
green  open   .apm-custom-link                w5g86ULeR_y3vpLpUNGIsA   1   1
     0            0        416b          208b
green  open   .kibana_task_manager_1          oX2j06nxTBCZ8btAv4-h-A   1   1
     6         5347      5.4mb          2.7mb
green  open   .apm-agent-configuration        Kifvpaa-Qu2JRsq-ZXdk_g   1   1
     0            0        416b          208b
green  open   .kibana_1                       C8Zq3F82SdSWplLN009gCQ   1   1
    16            3      20.8mb        10.4mb
deploy@elasticsearch:~$
```

Figure 55: VM Directory for Elasticsearch

7.10 Enabling pod to pod communication within a namespace

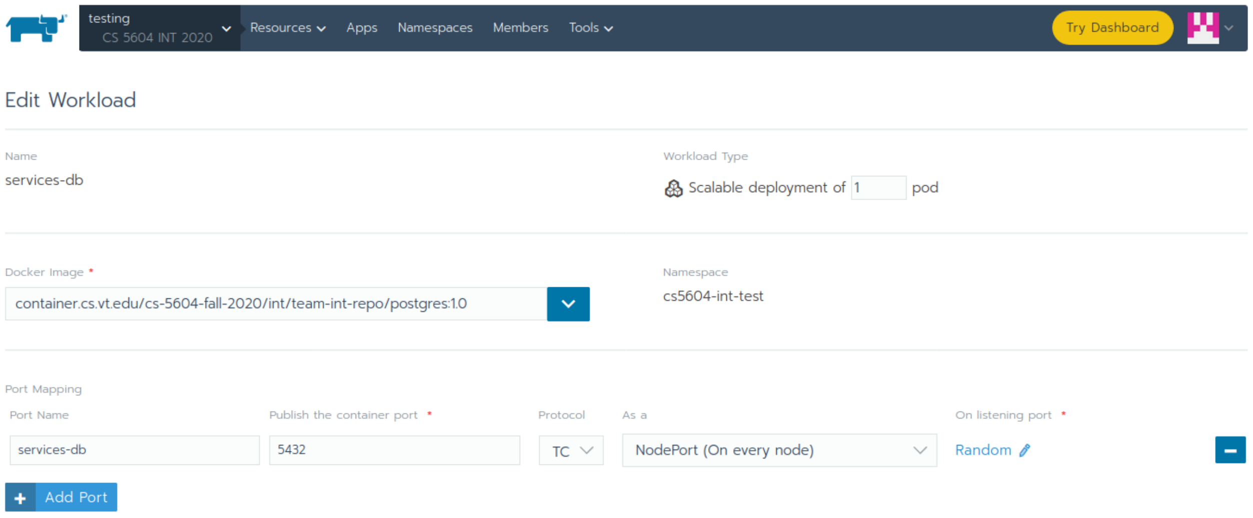
Once the pods are deployed, depending on the task at hand, it may be important for them to communicate between each other. For example, let's say one pod is running a database and another pod is running a web server that wishes to connect to the database. One could hard code the IP address of the pod running the database on the web server but this solution is not optimal as pods are allocated IP addresses dynamically. This means that if the database pod is redeployed for some reason, one has to change the IP address in the web server as well. To solve this issue, we create a `service-name` for the database pod. `service-name` is a `namespace` specific 'domain name' of sorts given to a pod using which any pod in a `namespace` can identify another pod within that same `namespace`. To do so, perform the following steps (see Figure 56).

- While deploying the pod to Rancher click on the `Ports` section. If the pod is already deployed, it can be accessed from the `Edit` page (found by clicking the three dots on the extreme right).
- Set the value of `port name` to the desired `service-name`. This value will be subsequently used to access this pod.
- Set the value of `Publish the container port` as the port number on which the database listens. For example, in the case of Postgres, it runs on port `5432`.
- Leave the other values as is.

Once done, it can be accessed using `service-name.namespace.svc.local`, in our case `services-db.cs5604-int-test.svc.cluster.local`

7.11 Installing Gitlab-Runner

In this section we describe how to install the Gitlab-Runner on a virtual machine in order to create a Continuous Integration and Continuous Development (CI/CD) environment. Since we are using Docker to run the Gitlab Runner, the only installation overhead is that Docker is installed on the specific VM, and those instructions will depend on your environment [30, 31]. Once Docker has been installed, the user will need to modify the config.toml at lines 9 and 10. Once the config file has been updated, the user should run



Edit Workload

Name: services-db

Workload Type: Scalable deployment of 1 pod

Docker Image *: container.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/postgres:1.0

Namespace: cs5604-int-test

Port Mapping

Port Name	Publish the container port *	Protocol	As a	On listening port *
services-db	5432	TC	NodePort (On every node)	Random

Figure 56: Creating a service name for pod to pod communication

```
docker cp config.toml gitlab_gitlab-runner_1:/etc/gitlab-runner
```

in order to update the configuration file within the container.

The user will need to change the URL to that of their Gitlab server on line 9. Now, the user needs to get a token value for line 10, so they should navigate to the project level CI/CD configuration as shown in Figure 57. From there, they should follow the instructions for setting up a group Runner using the example from Figure 58 where the URL will be the URL of their Gitlab instance (as discussed above) and the token (blacked out here) will be included on line 10 of their config.toml for Gitlab.

Once all this configuration has been completed, the user can now start the service by running

```
docker-compose up -d
```

and verifying it worked with

```
docker-compose logs
```

7.12 Installing Elasticsearch

In this section we describe how to install Elasticsearch on a virtual machine in order to index and search the data. Since we are using Docker to run Elasticsearch, the only installation overhead on the virtual machine is Docker itself and the instructions to install that will depend on your specific environment [30, 31]. Once Docker has been installed, the user will need to run

```
docker-compose up -d
```

and verifying Elasticsearch started with

```
docker-compose logs
```

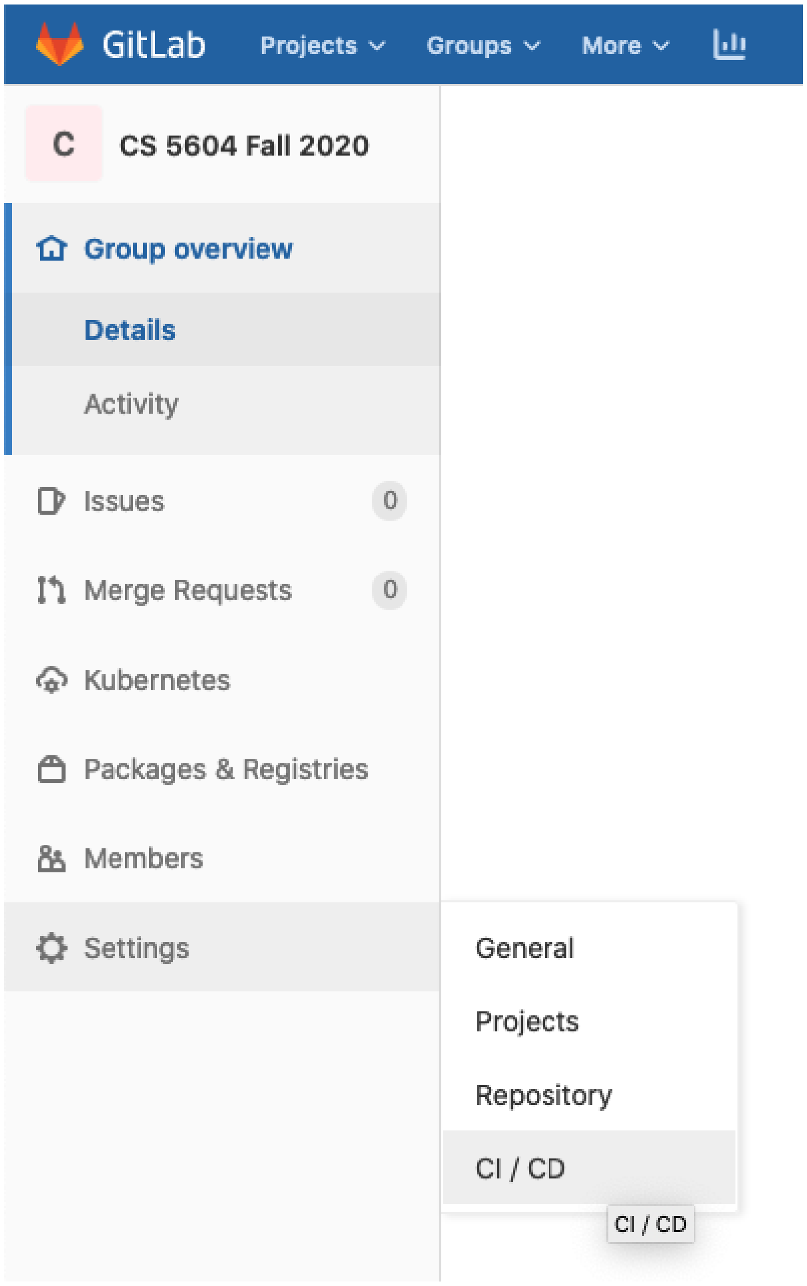
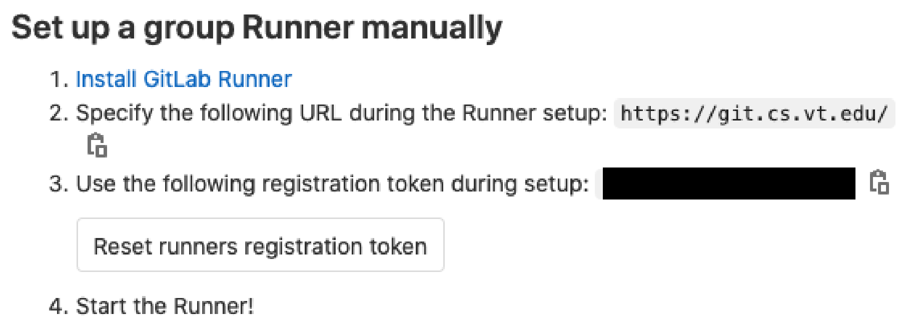


Figure 57: Connecting the CI/CD system

Once this step has been completed, the user must follow the Elasticsearch documentation in order to set up SSL and secure the instance [15, 11]. The final step is securing the virtual machine according to advice from the Virginia Tech Computer Science Techstaff. First, the user should run the following commands

```
sudo ufw limit ssh
ufw route allow proto tcp from 198.82.184.0/24 to any port 9200
sudo ufw enable (only needs to be run once)
```

Then they should follow the documentation for securing Docker to the internet. [57].


Set up a group Runner manually

1. Install GitLab Runner
2. Specify the following URL during the Runner setup: `https://git.cs.vt.edu/`
3. Use the following registration token during setup: [redacted]

4. Start the Runner!

Figure 58: Adding a Gitlab Runner to Gitlab

7.13 Service Registry Database

Figure 59 shows the schema used by the various tables jointly called the services database. The table descriptions are as follows.

A. Goal Metadata

This table stores metadata related to information goals that will be used as inputs to the services. It also includes metadata related to the output goal produced by a registered service. The following are the column descriptions.

- `goal_id` - The primary key of the table is an `int` value used to uniquely identify a goal.
- `goal_name` - A human readable `str` variable is stored in this column to represent the goal's name.
- `goal_description` - A `str` variable that describes the goal in detail, helps the end user in choosing the goal they want to use.
- `goal_format` - A `str` variable representing the format of the data represented by the goal.
- `environment_variable` - Name of the environment variable that will be available to use within the Docker containers running the services.
- `file_location` - A `str` variable that stores the path of the data (in the NFS) tied to this goal. If the data for the goal is generated by a service, an empty file with the expected file name should be stored in the NFS.

B. Service Metadata

This table stores metadata of the services that will be deployed as a part of workflows. The following are the column descriptions.

- `service_id` - The primary key of the table is an `int` value used to uniquely identify a service.
- `service_name` - A human readable `str` variable is stored in this column to represent the service's name.

- `service_description` - A `str` variable that describes the service in detail, that helps future developers understand the existing services and develop new services different from existing ones.
- `image_url` - A `str` variable representing the `url` of the image in the container registry. The service Docker image could be hosted on any publicly accessible Docker registry or Virginia Tech's local Docker registry (`http://container.cs.vt.edu`).
- `cluster_namespace` - A `str` variable that stores the kubernetes namespace in which the service will be deployed.

C. Reasoner

This table stores the goals needed for a service to run and what the goal the service produces as an output. This data is useful for the reasoner to generate workflows based on a requested information goal. Multiple entries can exist for a particular `service_id` depending on the number of goals needed / goals achievable. The table contains the following columns.

- `service_id` - Foreign key from the Service Metadata table.
- `goal_id` - Foreign key from the Goal Metadata table used to store the goal attainable using a particular service.
- `input_goal_id` - Foreign key from the Goal Metadata table used to store the input goal used by a particular service.

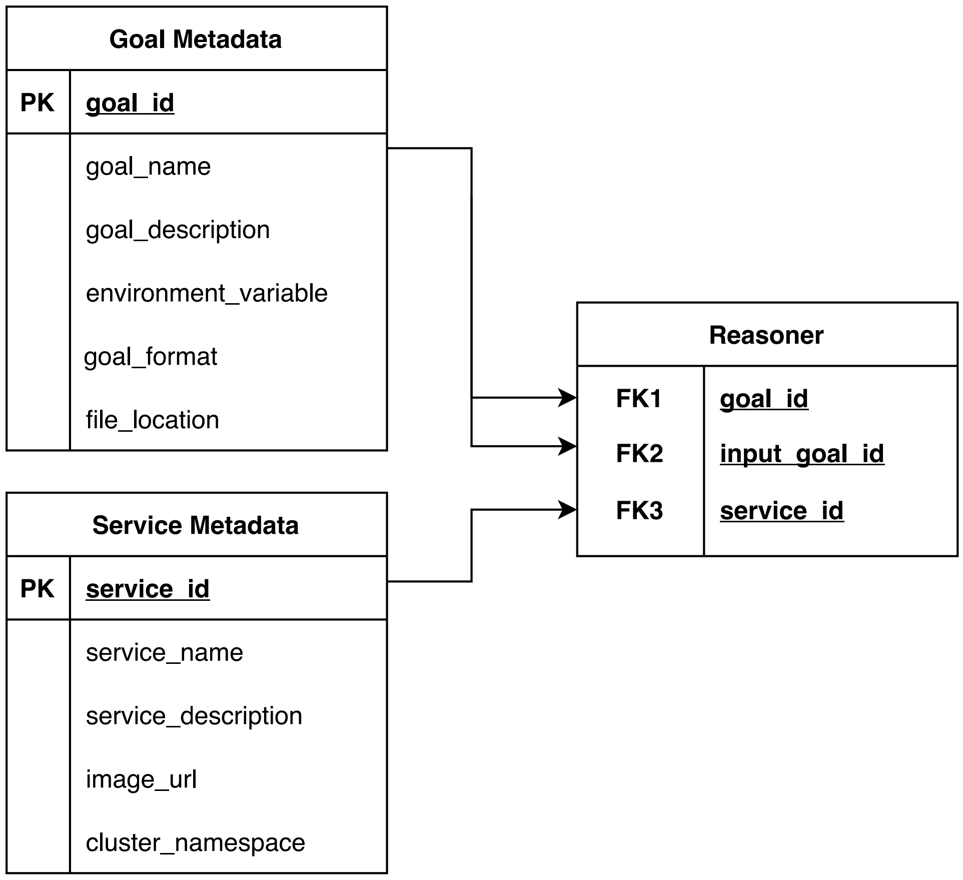


Figure 59: Schema of various tables used for registering services

References

[1] Google's Container Registry. https://cloud.google.com/container-registry/, accessed on October 8, 2020.

[2] Computer Science Cloud, 2020. https://cloud.cs.vt.edu, accessed on September 15, 2020.

[3] DockerSlim, Minify and Secure Your Docker Containers., 2020. https://dockersl.im, accessed on September 15, 2020.

[4] Podman, 2020. https://github.com/containers/podman, accessed on September 17, 2020.

[5] Zoom, 2020. https://zoom.us, accessed on September 15, 2020.

[6] 451 Research. 451 Research Says Application Containers Market Will Grow to Reach $4.3bn by 2022. https://451research.com/451-research-says-application-containers-market-will-grow-to-reach-4-3bn-by-2022, accessed on September 15, 2020.

[7] The Kubernetes Authors. Nodes - cluster architecture - Kubernetes Concepts, 2020. https://kubernetes.io/docs/concepts/architecture/nodes/, accessed on September 15, 2020.

[8] The Kubernetes Authors. Overview of kubectl, November 2020. https://kubernetes.io/docs/reference/kubectl/overview/, accessed on December 4, 2020.

[9] The Kubernetes Authors. Pod Overview - Workloads - Kubernetes Concepts, 2020. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, accessed on September 15, 2020.

[10] Michael Bose. Kubernetes vs Docker, 2019. https://www.nakivo.com/blog/docker-vs-kubernetes/, accessed on December 17, 2020.

[11] Elasticsearch B.V. and Apache Software Foundation. Encrypting communications in Elasticsearch. https://www.elastic.co/guide/en/elasticsearch/reference/6.6/configuring-tls.html, accessed on December 8, 2020.

[12] Elasticsearch B.V. and Apache Software Foundation. Kibana: Explore, visualize, discover data. https://www.elastic.co/kibana, accessed on October 8, 2020.

[13] Elasticsearch B.V. and Apache Software Foundation. Logstash: Collect, parse, transform logs. https://www.elastic.co/logstash, accessed on October 8, 2020.

[14] Elasticsearch B.V. and Apache Software Foundation. Open source search: The creators of Elasticsearch, ELK Stack and Kibana. https://www.elastic.co/, accessed on October 8, 2020.

[15] Elasticsearch B.V. and Apache Software Foundation. Setting up TLS on a cluster. https://www.elastic.co/guide/en/x-pack/current/ssl-tls.html#generating-signed-certificates, accessed on December 8, 2020.

[16] Elasticsearch B.V. and Apache Software Foundation. What is the ELK Stack? https://www.elastic.co/what-is/elk-stack, accessed on October 8, 2020.

[17] Canonical. Linux Containers, 2020. https://linuxcontainers.org/, accessed on September 15, 2020.

[18] Eric Carter. 2018 Docker Usage Report, May 29, 2018. https://sysdig.com/blog/2018-docker-usage-report/.

[19] Dave Bartoletti and Charlie Dai. The Forrester New Wave™: Enterprise Container Platform Software Suites, Q4 2018, 2020. https://www.docker.com/resources/report/the-forrester-wave-enterprise-container-platform-software-suites-2018, accessed on September 15, 2020.

[20] Docker. Best practices for writing Dockerfiles, 2020. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, accessed on September 15, 2020.

[21] Docker. Docker, 2020. https://www.docker.com/, accessed on September 15, 2020.

[22] Docker. Docker Hub, 2020. https://hub.docker.com/, accessed on September 15, 2020.

[23] Docker. Official Airflow Docker image, 2020. https://hub.docker.com/r/apache/airflow, accessed on October 8, 2020.

[24] Docker. Official Postgres Docker image, 2020. https://hub.docker.com/_/postgres, accessed on October 8, 2020.

[25] Apache Software Foundation. Airflow documentation: How to modify the Airflow configuration file, 2020. https://airflow.apache.org/docs/stable/howto/set-config.html, accessed on October 8, 2020.

[26] Apache Software Foundation. Apache Airflow, 2020. https://airflow.apache.org/docs/stable/, accessed on October 8, 2020.

[27] Apache Software Foundation. Apache Mesos, 2020. http://mesos.apache.org/, accessed on October 8, 2020,.

[28] Apache Software Foundation. Kafka Documentation, 2020. https://kafka.apache.org/intro, accessed on October 29, 2020.

[29] Gitlab. GitLab, 2020. https://about.gitlab.com/, accessed on September 15, 2020.

[30] Gitlab. GitLab Runner Docker, 2020. https://docs.gitlab.com/runner/install/docker.html, accessed on September 15, 2020.

[31] Gitlab. GitLab Runner Docs, 2020. https://docs.gitlab.com/runner/, accessed on September 15, 2020.

[32] Gitlab. Install GitLab Runner using the official GitLab repositories. 2020. https://docs.gitlab.com/runner/install/linux-repository.html, accessed on September 15, 2020.

[33] Google and Cloud Native Computing Foundation. Kubernetes. https://github.com/kubernetes/kubernetes, accessed on Dec 8, 2019.

[34] Red Hat. Building, running, and managing containers, 2020. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/building_running_and_managing_containers/index, accessed on September 17, 2020.

[35] Red Hat. CoreOS, 2020. https://coreos.com/, accessed on October 8, 2020.

[36] Alex Hicks. INT Team Elasticsearch repository on Gitlab, 2020. https://git.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/-/blob/master/scripts/elasticsearch/es.py, accessed on October 8, 2020.

[37] Discord Inc. Discord, 2020. https://discord.com, accessed on September 15, 2020.

[38] Jack Clark. Google: 'EVERYTHING at Google runs in a container', May 23, 2014. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[39] Kat Cosgrove Justin Garrison Noah Kantrowitz Bob Killen Rey Lejano Dan "POP" Papandrea Jeffrey Sica Davanum "Dims" Srinivas Jorge Castro, Duffie Cooley. Don't panic: Kubernetes and docker, 2020. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/, accessed on December 9, 2020.

[40] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. *NetDB*, pages 1–7, 2011.

[41] Rancher Labs. Catalogs and Apps, 2020. https://rancher.com/docs/rancher/v2.x/en/catalog/, accessed on September 15, 2020.

[42] Paul B Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux symposium*, volume 2, pages 45–57. Citeseer, 2007.

[43] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[44] Platform9. Deploy Kubernetes: The Ultimate Guide, 2020. https://platform9.com/docs/deploy-kubernetes-the-ultimate-guide/, accessed on September 15, 2020.

[45] Rancher Labs. Rancher. https://rancher.com, accessed on Dec 9, 2019.

[46] Rancher Labs. Rancher overview. https://rancher.com/what-is-rancher/overview/, accessed on Dec 9, 2019.

[47] Rancher Labs. What Rancher adds to Kubernetes. https://rancher.com/what-is-rancher/what-rancher-adds-to-kubernetes/, accessed on Dec 9, 2019.

[48] Red Hat, Inc., and contributors. Ceph, 2019. https://ceph.io, accessed on September 15, 2020.

[49] Red Hat, Inc., and contributors. Quay, 2020. https://quay.io/, accessed on September 15, 2020.

[50] Rami Rosen. Linux containers and the future cloud. *Linux J*, 240(4):86–95, 2014.

[51] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[52] Virginia Tech Computer Science Technical Staff. Virginia Tech Docker Registry, 2019. https://wiki.cs.vt.edu/wiki/Howto::Docker_Registry, accessed on September 16, 2020.

[53] Ally Technologies. Ally.io, 2020. https://ally.io, accessed on September 15, 2020.

[54] Mohit Thazhath. INT Team Airflow repository on Gitlab, 2020. https://git.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/-/tree/master/airflow, accessed on October 8, 2020.

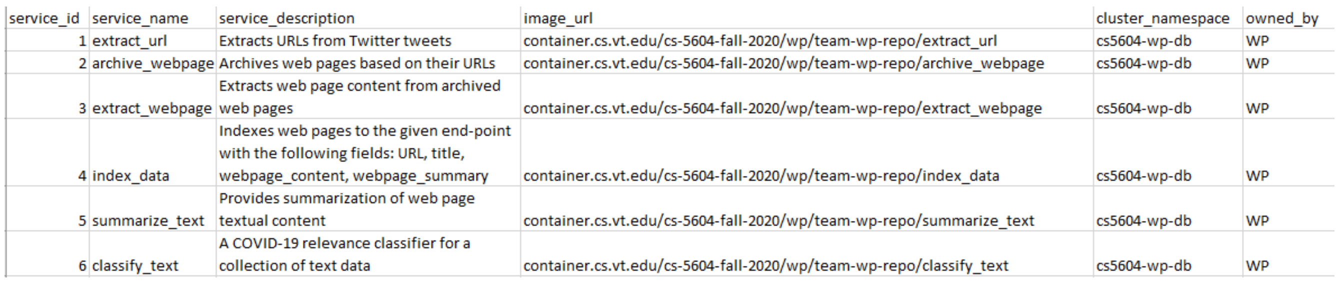
[55] Mohit Thazhath. INT Team Postgres repository on gitlab, 2020. https://git.cs.vt.edu/cs-5604-fall-2020/int/team-int-repo/-/tree/master/postgres, accessed on October 8, 2020.

[56] Mohit Thazhath. Issue raised on Apache Airflow Github repository, 2020. https://github.com/apache/airflow/issues/12728, accessed on December 2, 2020.

[57] Jack Wallen. How to fix the Docker and UFW security flaw, 2018. https://www.techrepublic.com/article/how-to-fix-the-docker-and-ufw-security-flaw/, accessed on December 8, 2020.

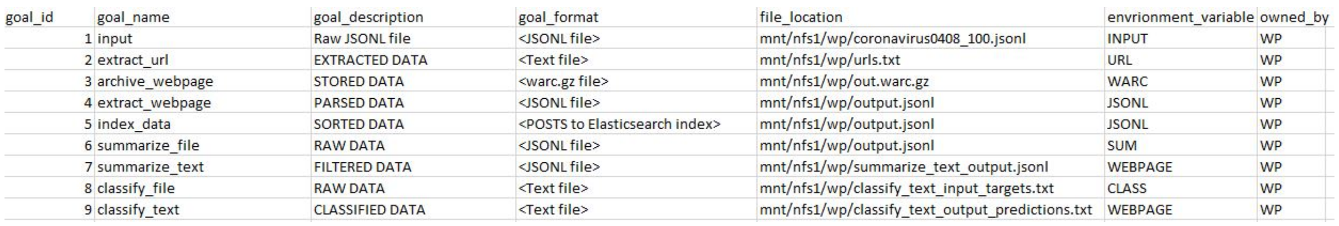
[58] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

A Manual Service Registry Artifacts



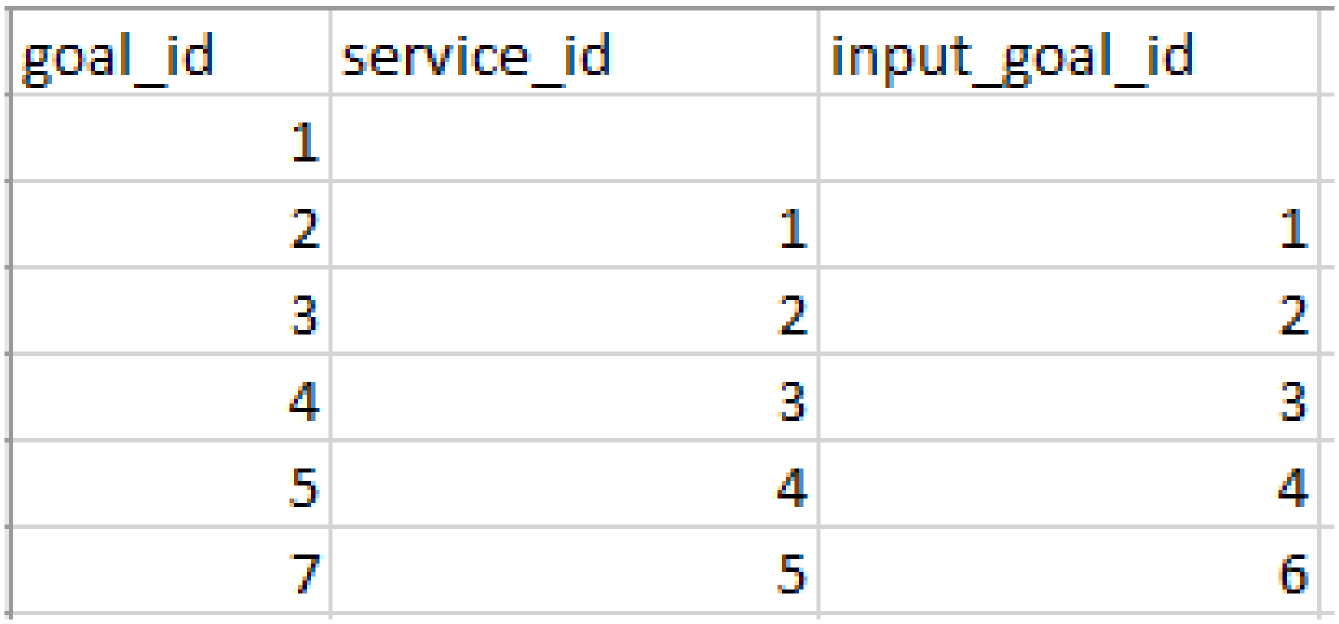
service_id	service_name	service_description	image_url	cluster_namespace	owned_by
1	extract_url	Extracts URLs from Twitter tweets	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/extract_url	cs5604-wp-db	WP
2	archive_webpage	Archives web pages based on their URLs	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/archive_webpage	cs5604-wp-db	WP
3	extract_webpage	Extracts web page content from archived web pages	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/extract_webpage	cs5604-wp-db	WP
4	index_data	Indexes web pages to the given end-point with the following fields: URL, title, webpage_content, webpage_summary	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/index_data	cs5604-wp-db	WP
5	summarize_text	Provides summarization of web page textual content	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/summarize_text	cs5604-wp-db	WP
6	classify_text	A COVID-19 relevance classifier for a collection of text data	container.cs.vt.edu/cs-5604-fall-2020/wp/team-wp-repo/classify_text	cs5604-wp-db	WP

Figure 60: WP Service Metadata



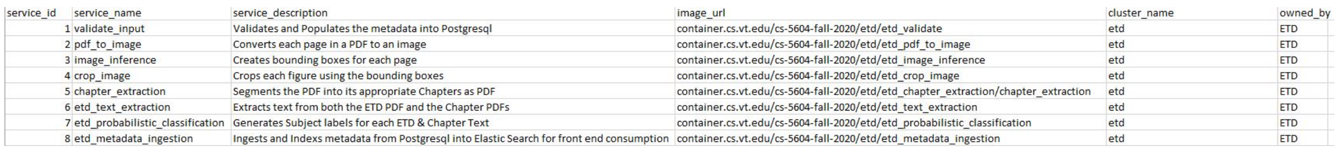
goal_id	goal_name	goal_description	goal_format	file_location	envrionment_variable	owned_by
1	input	Raw JSONL file	<JSONL file>	mnt/nfs1/wp/coronavirus0408_100.jsonl	INPUT	WP
2	extract_url	EXTRACTED DATA	<Text file>	mnt/nfs1/wp/urls.txt	URL	WP
3	archive_webpage	STORED DATA	<warc.gz file>	mnt/nfs1/wp/out.warc.gz	WARC	WP
4	extract_webpage	PARSED DATA	<JSONL file>	mnt/nfs1/wp/output.jsonl	JSONL	WP
5	index_data	SORTED DATA	<POSTS to Elasticsearch index>	mnt/nfs1/wp/output.jsonl	JSONL	WP
6	summarize_file	RAW DATA	<JSONL file>	mnt/nfs1/wp/output.jsonl	SUM	WP
7	summarize_text	FILTERED DATA	<JSONL file>	mnt/nfs1/wp/summarize_text_output.jsonl	WEBPAGE	WP
8	classify_file	RAW DATA	<Text file>	mnt/nfs1/wp/classify_text_input_targets.txt	CLASS	WP
9	classify_text	CLASSIFIED DATA	<Text file>	mnt/nfs1/wp/classify_text_output_predictions.txt	WEBPAGE	WP

Figure 61: WP Goal Metadata



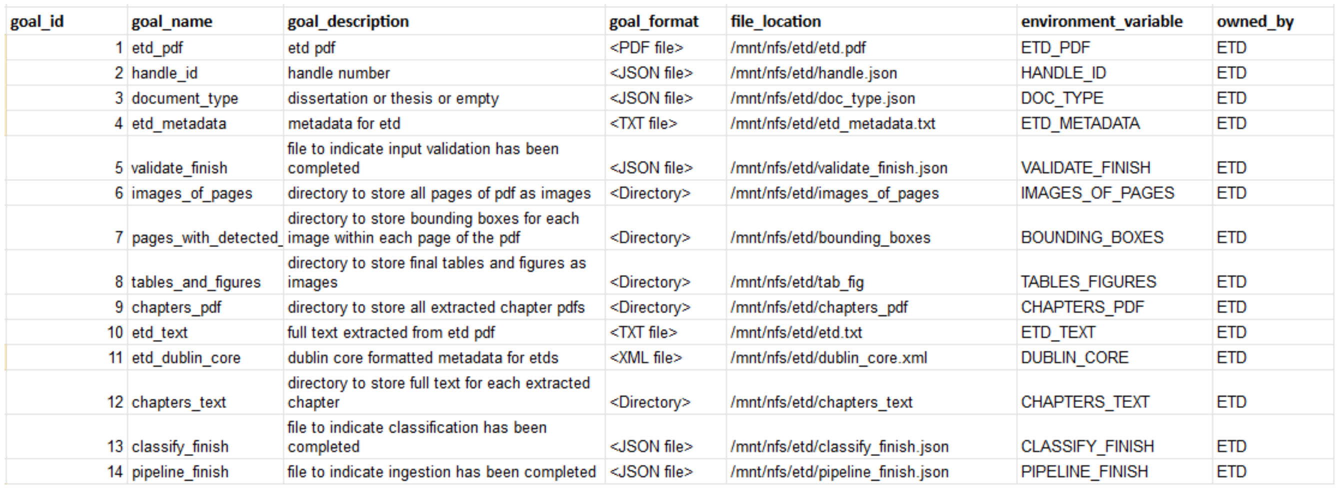
goal_id	service_id	input_goal_id
1		
2	1	1
3	2	2
4	3	3
5	4	4
7	5	6

Figure 62: WP Reasoner Metadata



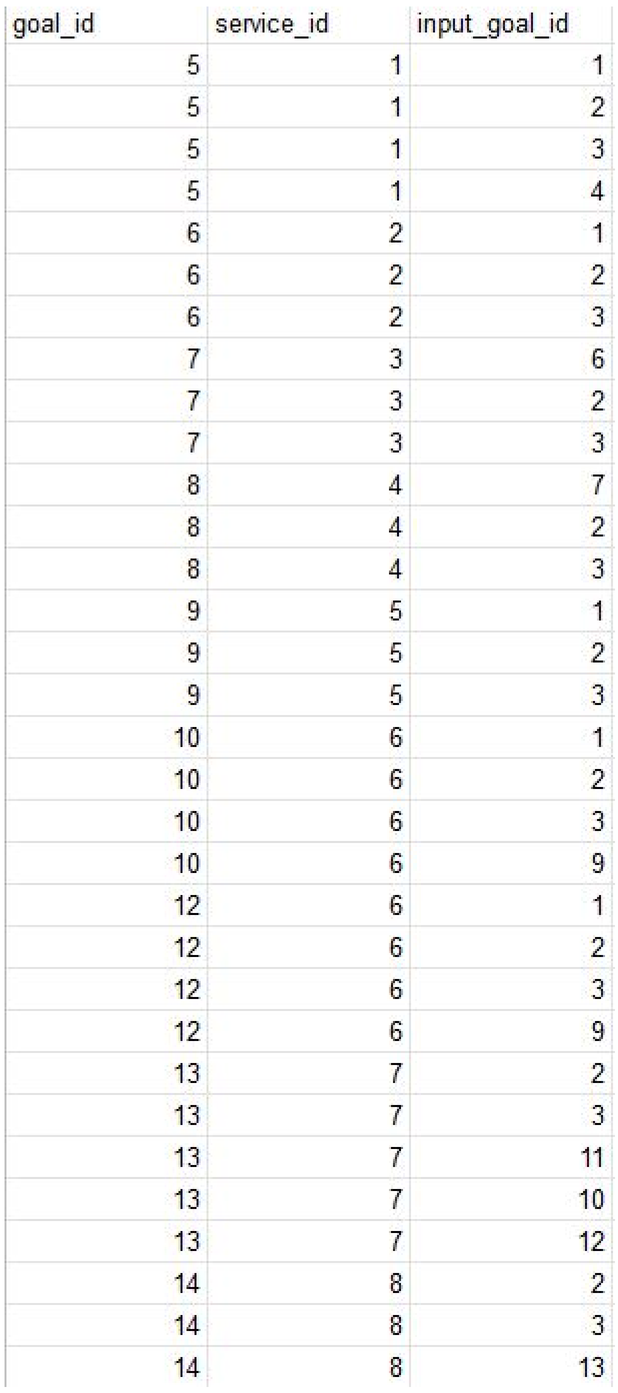
service_id	service_name	service_description	image_url	cluster_name	owned_by
1	validate_input	Validates and Populates the metadata into Postgresql	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_validate	etd	ETD
2	pdf_to_image	Converts each page in a PDF to an image	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_pdf_to_image	etd	ETD
3	image_inference	Creates bounding boxes for each page	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_image_inference	etd	ETD
4	crop_image	Crops each figure using the bounding boxes	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_crop_image	etd	ETD
5	chapter_extraction	Segments the PDF into its appropriate Chapters as PDF	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_chapter_extraction/chapter_extraction	etd	ETD
6	etd_text_extraction	Extracts text from both the ETD PDF and the Chapter PDFs	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_text_extraction	etd	ETD
7	etd_probabilistic_classification	Generates Subject labels for each ETD & Chapter Text	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_probabilistic_classification	etd	ETD
8	etd_metadata_ingestion	Ingests and Indexs metadata from Postgresql into Elastic Search for front end consumption	container.cs.vt.edu/cs-5604-fall-2020/etd/etd_metadata_ingestion	etd	ETD

Figure 63: ETD Service Metadata



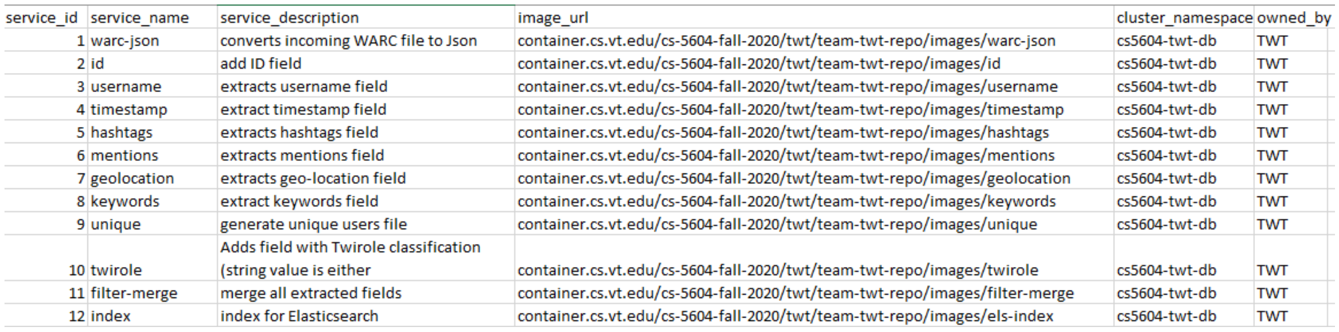
goal_id	goal_name	goal_description	goal_format	file_location	environment_variable	owned_by
1	etd_pdf	etd pdf	<PDF file>	/mnt/nfs/etd/etd.pdf	ETD_PDF	ETD
2	handle_id	handle number	<JSON file>	/mnt/nfs/etd/handle.json	HANDLE_ID	ETD
3	document_type	dissertation or thesis or empty	<JSON file>	/mnt/nfs/etd/doc_type.json	DOC_TYPE	ETD
4	etd_metadata	metadata for etd	<TXT file>	/mnt/nfs/etd/etd_metadata.txt	ETD_METADATA	ETD
5	validate_finish	file to indicate input validation has been completed	<JSON file>	/mnt/nfs/etd/validate_finish.json	VALIDATE_FINISH	ETD
6	images_of_pages	directory to store all pages of pdf as images	<Directory>	/mnt/nfs/etd/images_of_pages	IMAGES_OF_PAGES	ETD
7	pages_with_detected_	directory to store bounding boxes for each image within each page of the pdf	<Directory>	/mnt/nfs/etd/bounding_boxes	BOUNDING_BOXES	ETD
8	tables_and_figures	directory to store final tables and figures as images	<Directory>	/mnt/nfs/etd/tab_fig	TABLES_FIGURES	ETD
9	chapters_pdf	directory to store all extracted chapter pdfs	<Directory>	/mnt/nfs/etd/chapters_pdf	CHAPTERS_PDF	ETD
10	etd_text	full text extracted from etd pdf	<TXT file>	/mnt/nfs/etd/etd.txt	ETD_TEXT	ETD
11	etd_dublin_core	dublin core formatted metadata for etds	<XML file>	/mnt/nfs/etd/dublin_core.xml	DUBLIN_CORE	ETD
12	chapters_text	directory to store full text for each extracted chapter	<Directory>	/mnt/nfs/etd/chapters_text	CHAPTERS_TEXT	ETD
13	classify_finish	file to indicate classification has been completed	<JSON file>	/mnt/nfs/etd/classify_finish.json	CLASSIFY_FINISH	ETD
14	pipeline_finish	file to indicate ingestion has been completed	<JSON file>	/mnt/nfs/etd/pipeline_finish.json	PIPELINE_FINISH	ETD

Figure 64: ETD Goal Metadata



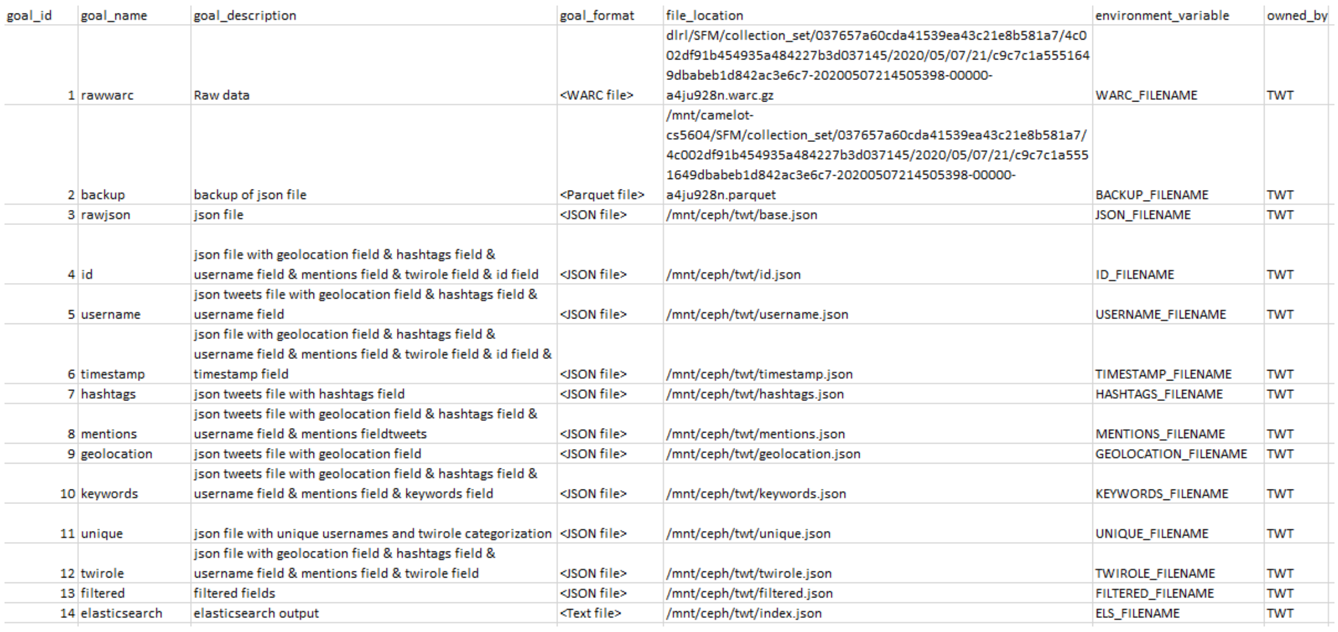
goal_id	service_id	input_goal_id
5	1	1
5	1	2
5	1	3
5	1	4
6	2	1
6	2	2
6	2	3
7	3	6
7	3	2
7	3	3
8	4	7
8	4	2
8	4	3
9	5	1
9	5	2
9	5	3
10	6	1
10	6	2
10	6	3
10	6	9
12	6	1
12	6	2
12	6	3
12	6	9
13	7	2
13	7	3
13	7	11
13	7	10
13	7	12
14	8	2
14	8	3
14	8	13

Figure 65: ETD Reasoner Metadata



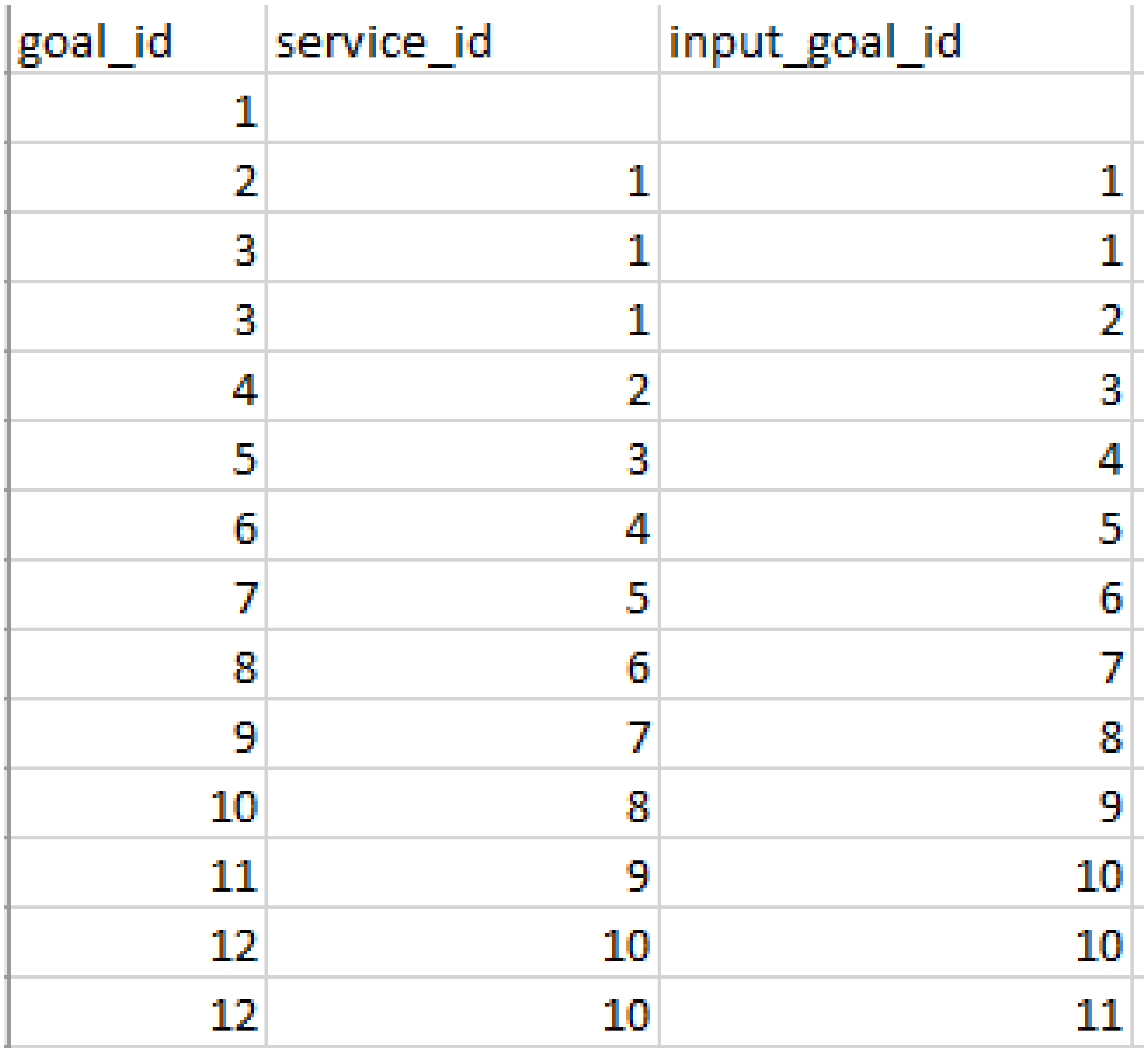
service_id	service_name	service_description	image_url	cluster_namespace	owned_by
1	warc-json	converts incoming WARC file to Json	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/warc-json	cs5604-twt-db	TWT
2	id	add ID field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/id	cs5604-twt-db	TWT
3	username	extracts username field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/username	cs5604-twt-db	TWT
4	timestamp	extract timestamp field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/timestamp	cs5604-twt-db	TWT
5	hashtags	extract hashtags field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/hashtags	cs5604-twt-db	TWT
6	mentions	extracts mentions field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/mentions	cs5604-twt-db	TWT
7	geolocation	extracts geo-location field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/geolocation	cs5604-twt-db	TWT
8	keywords	extract keywords field	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/keywords	cs5604-twt-db	TWT
9	unique	generate unique users file	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/unique	cs5604-twt-db	TWT
10	twirole	Adds field with Twirole classification (string value is either	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/twirole	cs5604-twt-db	TWT
11	filter-merge	merge all extracted fields	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/filter-merge	cs5604-twt-db	TWT
12	index	index for Elasticsearch	container.cs.vt.edu/cs-5604-fall-2020/twt/team-twt-repo/images/els-index	cs5604-twt-db	TWT

Figure 66: TWT Service Metadata



goal_id	goal_name	goal_description	goal_format	file_location	environment_variable	owned_by
1	rawwarc	Raw data	<WARC file>	dlrl/SFM/collection_set/037657a60cda41539ea43c21e8b581a7/4c002df91b454935a484227b3d037145/2020/05/07/21/c9c7c1a5551649dbabeb1d842ac3e6c7-20200507214505398-00000-a4ju928n.warc.gz	WARC_FILENAME	TWT
2	backup	backup of json file	<Parquet file>	/mnt/camelot-cs5604/SFM/collection_set/037657a60cda41539ea43c21e8b581a7/4c002df91b454935a484227b3d037145/2020/05/07/21/c9c7c1a5551649dbabeb1d842ac3e6c7-20200507214505398-00000-a4ju928n.parquet	BACKUP_FILENAME	TWT
3	rawjson	json file	<JSON file>	/mnt/ceph/twt/base.json	JSON_FILENAME	TWT
4	id	json file with geolocation field & hashtags field & username field & mentions field & twirole field & id field	<JSON file>	/mnt/ceph/twt/id.json	ID_FILENAME	TWT
5	username	json tweets file with geolocation field & hashtags field & username field	<JSON file>	/mnt/ceph/twt/username.json	USERNAME_FILENAME	TWT
6	timestamp	json file with geolocation field & hashtags field & username field & mentions field & twirole field & id field & timestamp field	<JSON file>	/mnt/ceph/twt/timestamp.json	TIMESTAMP_FILENAME	TWT
7	hashtags	json tweets file with hashtags field	<JSON file>	/mnt/ceph/twt/hashtags.json	HASHTAGS_FILENAME	TWT
8	mentions	json tweets file with geolocation field & hashtags field & username field & mentions fieldtweets	<JSON file>	/mnt/ceph/twt/mentions.json	MENTIONS_FILENAME	TWT
9	geolocation	json tweets file with geolocation field	<JSON file>	/mnt/ceph/twt/geolocation.json	GEOLOCATION_FILENAME	TWT
10	keywords	json tweets file with geolocation field & hashtags field & username field & mentions field & keywords field	<JSON file>	/mnt/ceph/twt/keywords.json	KEYWORDS_FILENAME	TWT
11	unique	json file with unique usernames and twirole categorization	<JSON file>	/mnt/ceph/twt/unique.json	UNIQUE_FILENAME	TWT
12	twirole	json file with geolocation field & hashtags field & username field & mentions field & twirole field	<JSON file>	/mnt/ceph/twt/twirole.json	TWIROLE_FILENAME	TWT
13	filtered	filtered fields	<JSON file>	/mnt/ceph/twt/filtered.json	FILTERED_FILENAME	TWT
14	elasticsearch	elasticsearch output	<Text file>	/mnt/ceph/twt/index.json	ELS_FILENAME	TWT

Figure 67: TWT Goal Metadata



goal_id	service_id	input_goal_id
1		
2	1	1
3	1	1
3	1	2
4	2	3
5	3	4
6	4	5
7	5	6
8	6	7
9	7	8
10	8	9
11	9	10
12	10	10
12	10	11

Figure 68: TWT Reasoner Metadata