Final Report
CS 5604: Information Storage and Retrieval

Integration and Implementation (INT) Team:
Rahul Agarwal, Hadeel Albahar, Eric Roth,
Malabika Sen, Lixing Yu

January 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 the second major goal, supporting modern search and browse capabilities for two major content collections: (1) ~200,000 ETDs (electronic theses and dissertations), and (2) ~14 million settlement documents from the lawsuit wherein 39 U.S. states sued the major tobacco companies.

The backbone of the information system is a Docker container cluster running with Rancher and Kubernetes. Information retrieval and visualization is accomplished with containers 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 exercises 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 6 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.

These are the teams on this project: Collection Management ETDs (CME), Collection Management Tobacco Settlement Documents (CMT), Elasticsearch (ELS), Front-end and Kibana (FEK), Integration and Implementation (INT), and Text Analysis and Machine Learning (TML).

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. For example, the ELS team container environment shall include Elasticsearch with its internal associated database. INT also administers the integration of the Ceph data storage system into the CS Department Cloud and provides support for interactions between containers and Ceph. During formative stages of development, INT also has a role in guiding team evaluations of prospective container components.

Beyond the project formative stages, INT has the responsibility of deploying containers in a development environment according to mutual specifications agreed upon with each team. The development process is fluid. INT services team requests for new containers and updates to existing containers in a continuous integration process until the first system testing environment is completed. During the development stage INT also collaborates with the CME and CMT teams on the data pipeline subsystems for the ingestion and processing of new collection documents.

With the testing environment established, the focus of the INT team shifts toward gathering of system performance data and making any systemic adjustments necessary based on the analysis of testing results.

Finally, INT provides a production distribution that includes all embedded Docker containers and sub-embedded Git source code repositories. INT archives this distribution on Docker Hub and deploys it on the Virginia Tech CS Cloud.

Contents

List of Tables

List of Figures

1 Overview

1.1 Project Management

There are five members in our team, so team management is crucial to accomplish the project goal. Including class sessions, we schedule Zoom meetings [40] two times a week on average to discuss tasks and issues, and to plan ahead for future work to ensure progress. Because the INT team's role is to ensure an operating portable container environment by integrating all the teams' efforts, we communicate with the other teams in class sessions and on designated Slack [36] channels (e.g., #int-service-requests) 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 Slack and Zoom for team communication, Trello [39] for task management, GitHub [12] for code collaboration, and Docker Hub [9] 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. Transferring large processed datasets into Ceph File System storage. It is not simple to copy large data from VMs that store the tobacco and ETD datasets over the internet to Ceph storage; a connection between the VMs and Ceph must be established. This problem is solved. The steps to follow are in §8.1.

c. The time limitation of containers' shell on the Computer Science (CS) cloud (i.e., 1-hour timeout), making all processes running shut down after that time interval. This problem is solved. The steps to follow are in §8.2.

d. The ephemeral execution environment provided by the containers' shell on the Computer Science (CS) cloud. This meant that all installations and executions are deleted after closing the shell. This problem is solved. The steps to follow are in §8.2.

e. The inability to save and package a modified container as a new container image by using kubectl or Rancher (see §2 and §4.2). 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. This problem is solved. The solution is explained in §8.7.

f. Some of the Docker container images that are created and built (e.g., via Dockerfiles [2]) are very large. This will slow down the deployment and the scaling of the services that these containers provide. A solution to this problem is under development. We propose to shrink the size of Docker images by using tools such as Docker-Slim [11] or by creating very efficient Dockerfiles. To facilitate the latter, other project teams must finalize task requirements. This will guide us toward removing unused and unneeded packages and libraries.

g. We have faced some challenges while deploying Kafka on our Kubernetes cluster. We investigated several approaches. We tried using Kafka's Helm charts but this approach failed because it requires Helm to be installed on Kubernetes `default` namespace for which we do not have access. We also tried using Rancher's catalogs, specifically Confluent Kafka Chart, which was deployed, however, we could not configure Kafka topics and metadata to successfully push data. Another approach was by deploying a single *zookeeper* Docker image and a single *broker* Docker image. However, we were unable to connect to Loadbalancer Ingress possibly because Kafka does not support IPv6. We finally tried Spotify's Single Docker packaged Kafka, however, it was only supported on Docker-swarm (Docker's built-in orchestration support).

h. In order to set up a CI/CD pipeline on GitLab [14], we are required to deploy a GitLab *Runner* instance, a service used to run jobs like tests and send the results back to GitLab. The way to do it is to deploy a GitLab Runner instance into our Kubernetes cluster by using the gitlab-runner Helm chart [16]. However, that option requires administrator permissions and Helm installed on Kubernetes `default` namespace. We tried integrating our Kubernetes cluster by using the kubeconfig file, but got a `403 forbidden` error that is probably due to not having access to the default namespace.

i. After setting the Kafka broker cluster[1] and zookeeper containers and then creating all the topics we need for automating the data pipeline, it was challenging to decide where to exactly place Kafkacat producer and consumer commands within our system, especially since we don't have all of our system components containerized. Nevertheless, knowing where to place the Kafkacat commands, it is straightforward to send/receive data to/from Kafka topics by using the Kafkacat command in producer or consumer mode and specifying the topic and directory path to the location of the new data.

j. The production cluster (Teaching cluster) does not have any physical nodes, therefore, until the cluster has some nodes, no containers can be deployed.

1.3 Solutions Developed

In order to deal with some of the challenges mentioned in §1.2, we often contacted Chris Arnold, of the CS technical staff at Virginia Tech. Chris's significant efforts to help us are much appreciated. We were able to solve the problem of transferring large processed datasets into Ceph by installing a Ceph client on the Virtual Machines that host the tobacco and ETD datasets and mounting Ceph as a volume for direct interaction. We have created a tutorial (see §8.1) that explains the above process for reference.

To address the 1-hour timeout limit on the CS cloud containers' shell, we leveraged the kubectl tool by installing it on our local machines and configuring it to connect to the CS cloud cluster. The kubectl tool facilitates direct control and management of all the containers and connected services (e.g., Ceph) in the cluster. The detailed process of installing and configuring kubectl has been documented in §8.2.

To solve the inability to save and package a modified container, we can utilize Docker's command line interface, specifically the `Docker commit` [8] command, to save a running container's

[1] In our case we only deployed one broker and exposed it on `testing133.cs.vt.edu:9092`.

current state as an image for portability purposes. With that said, building a Docker container from a Dockerfile is how we will do it. Moreover, since we will be deploying multiple containers, we will be leveraging a YAML file which will describe all the containers to be deployed in our system. Such a YAML file describes a multi-container application system, and is deployed either via `docker-compose` on hosts where Docker is installed or `Kompose` (discussed in §5.4) on Kubernetes infrastructures.

In pursuit of a solution for a continuous integration and continuous deployment (CI/CD) pipeline, we performed a comprehensive study on which tool is most suitable for the job (refer to Table 1 in §4.6). We laid out a foundation that can be used as a basis for complete CI/CD integration. On GitLab there is a CS5604 project that we created where teams (analogous to those in our project) can push their work. We also provided a *Hello World* style demonstration on how to configure a CI/CD pipeline.

Part of the process of setting up our GitLab pipeline involved the challenge of deploying a GitLab *Runner*. We tackled this by installing the Runner on a dedicated virtual machine instead of connecting GitLab to our Kubernetes cluster. We followed a procedure [20] for installing GitLab Runner that was archived in the GitLab Docs repository.

Another aspect of our system that we pursued was a solution to ingest new documents into our collections. This ingestion system was designed to be based on Apache Kafka to serve as the underlying data pipeline (See §4.5). In the development process we encountered a problem of deploying a single zookeeper container and a single Kafka broker, but we were able to fix the problem by following an article [59] on DZone. The only change we had to make was to assign a static IPv4 address as the LoadBalancer Ingress of the Kafka-service since our Kubernetes cluster could not handle IPv6 for internal communications[2]. Finally, the communication of the Kafka-service and the Kafka *broker* was established by assigning the same IPv4 as the environment variable in kafka-broker.yml.

1.4 Potential Development

Building Docker [7] containers is a big part of our team's objective. To get the teams started, we have deployed a few basic Docker containers on Rancher for development and testing purposes. The deployed containers include a base image of CentOS and another CentOS container with a Ceph Volume mounted for each team. We have equipped the teams with different containers that package different tools and libraries in the testing cluster based on an initial set of requirements the teams provided. Some teams have not finally provided a clearly established and evaluated requirements and scripts for their data processing, machine learning models, etc., for us to create customized and dedicated Docker containers for their tasks.

We wanted to support the ingestion of new data, specifically ETDs, into our information retrieval system. Therefore, we started exploring Kafka (explained in §4.5) and have successfully deployed Kafka and created the needed topics. However, we have not integrated Kafkacat (explained in §4.5.1) producer and consumer commands into the components of our system since they are not completely containerized. With that said, we have provided a diagram connecting all the producers and consumers to all the topics in our system.

We had also planned to create a CI/CD pipeline for each of the services being deployed on our

[2] For more details follow this article: [42]

cluster. But due to time constraints and dependency on other teams to provide us with the unit tests, we were not able to integrate the CI/CD pipeline with their solutions. Future development to fully implement CI/CD could continue using the CS5604 GitLab project that we set up.

Finally, we were expected to deploy everything on the production cluster, but we were unable to achieve that due to resource constraints and also because other teams were not ready with their production level code. However, we did research how we can leverage `Kompose` [37] to deploy multiple dependent containers together.

2 Literature Review

Containers [51, 58] are replacing virtual machines and are being adopted to modernize applications. Containers are favored for their lightweight, isolation, and portability features [52]. Such features are possible by leveraging Linux primitives such as cgroups and namespaces [57]. The container market will be a $4.3 billion market by 2022 [41]. Docker, the leading container management platform [44], is the main agent that is driving the adoption of containers [52].

For this project, we are leveraging the CS cloud [4]. The CS cloud deploys Rancher [53] that provides services on top of Kubernetes [45] 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 [7] container clusters. Rancher provides a simple workload environment, a centralized control plane, and enterprise-grade technical support.

Kubernetes [45], 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 [6]. 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.

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) 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 Git Lab).

3.2 INT Team Requirements

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

- Designing and deploying Docker containers and managing the Kubernetes cluster on the CS Cloud via Docker, kubectl, and Rancher.
- Managing the interactions and communications between the containers and Ceph storage.
- Effective data collection/delivery from/to each team to ensure progress.
- Evaluating and testing the cluster components at various stages of development.
- Seamless deployment of all the cluster components in both testing and production environments.

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 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 six projects are CS5604-CME, CS5604-CMT, CS5604-ELS, CS5604-FEK, CS5604-INT, and CS5604-TML. Under each project, we have deployed (1) a basic CentOS container and (2) another CentOS container that is connected to Ceph storage by mounting an ephemeral volume (Ceph File System) to the CentOS container. In order to be able to do that, a Ceph storage server must be running. The CS Technical Staff has already set up the Ceph storage system and it is available as a Kubernetes persistent storage, i.e., part of the CS Cloud cluster.

4.1 Docker Containers

Containers are a lightweight alternative to virtual machines. They are an operating-system-level virtualization whereby the operating system isolates the application and limits its resources (see Figure 1). In contrast to hardware-level virtualization, containers ensure fast deployment with low performance overhead.

Containers enable users to run software applications in a reliable and portable way by encapsulating an entire execution environment (e.g., application, dependencies, libraries, and needed configuration files) into one package. This facilitates fast and consistent application deployment regardless of the deployment environment. That has led to the widespread adoption of container technology [46]. Containers' impact on data-center efficiency is significant because of the resulting reduction in operational costs due to higher consolidation density and power minimization.

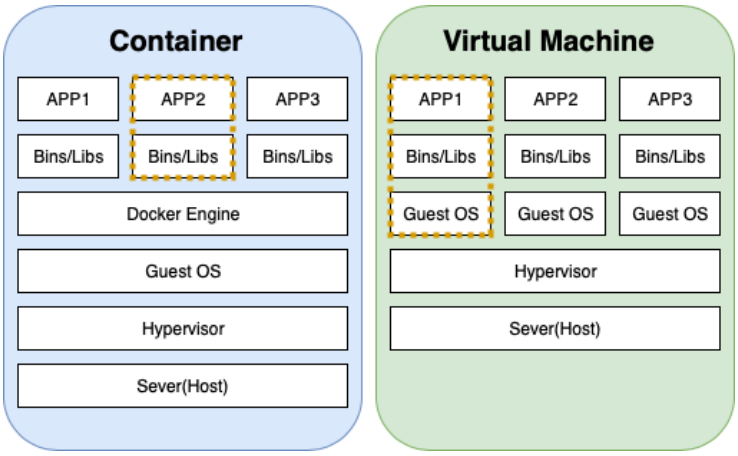


Figure 1: The difference between a container and virtual machine

Containers leverage Linux kernel features[3] – control groups (i.e., cgroups) and namespaces – to achieve the desired isolation and portability features [29]. Docker is currently the world's leading container management platform, followed by CoreOS rkt, Mesos, and LXC [43]. Docker simplifies

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

the containerization and distribution of applications by providing a tool for packing, shipping, and running applications.

Containers are created from images that are stored in an online store called a container registry such as Docker Hub [9], Quay [35], and Google Container Registry [17]. Users access container registries by pulling and pushing images using a Docker engine on end hosts that process user commands through a daemon process.

The INT team is responsible for creating and deploying containers according to other teams' requirements. However, because of the portable nature of containers, all of the teams have the option to deploy Docker containers on their own machines and refine them according to their project needs. The simplicity of Rancher and kubectl allows us to work expediently with containers in the CS cloud cluster. From there all the team containers can be integrated into one operating container cluster.

4.2 Kubernetes, Rancher, kubectl, and Containers

Large-scale applications often involve many containers that are associated in clusters. Kubernetes (K8s) is an open-source system that automates the deployment, scaling, and management of container clusters. It simplifies the management of container clusters by combining the containers that form an application into one unit [45].

Rancher is an open-source platform that provides services on top of Kubernetes [53]. In addition to providing Kubernetes-as-a-Service, Rancher has the following capabilities:

- Rancher provides a centralized management of clusters and the containers running in them. Since every team has its own project and namespace, operational and security challenges can be managed efficiently.

- Rancher is resource-agnostic; thus we can bring up clusters from different providers and migrate resources [54]. Rancher unifies the individual deployments as a single Kubernetes Cloud.

- Rancher has easy authentication policies for users and groups. Admins can enable self-service by delegating the administration of Kubernetes clusters or projects and namespaces directly to individual users or groups [55]. This is especially useful since it allows us to control memberships, ownerships, permissions, and capabilities of the six different team projects.

Kubectl [33] is the Kubernetes command-line tool to control the Kubernetes cluster manager. Kubectl can be used to deploy and manage pods (made up of a container, or several closely related containers) in the Kubernetes cluster.

Figure 2 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 [32]. 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 [34].

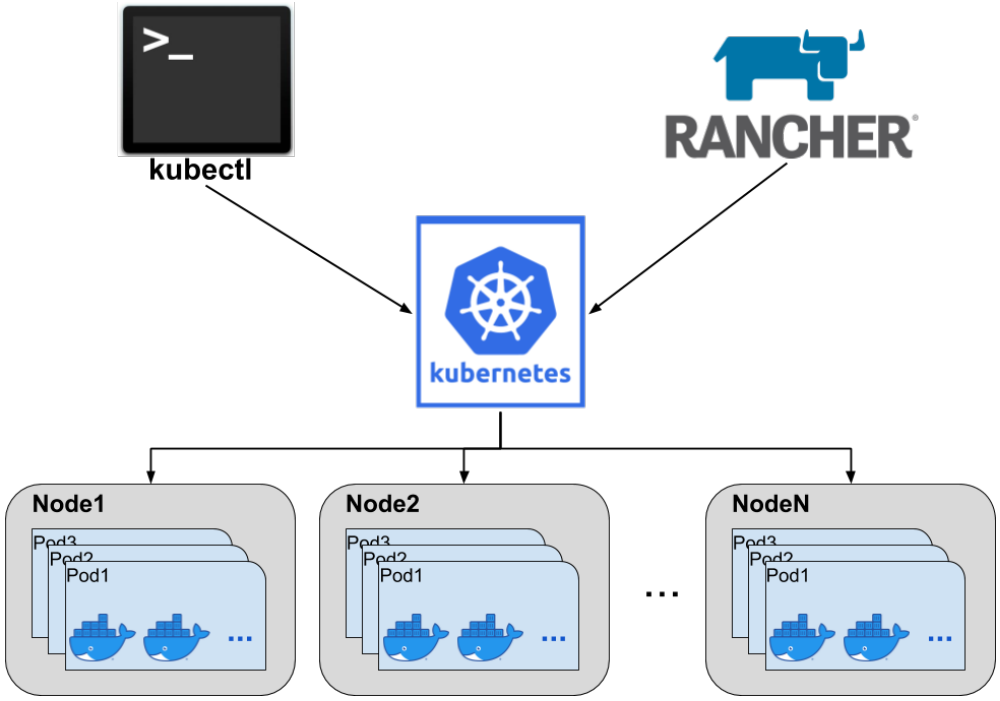


Figure 2: Container administration hierarchy

4.3 Ceph

The storage system used in our project is Ceph [60, 56]. Ceph basically is a shared storage system that is mounted on different components of our storage and retrieval system including containers and Tobacco and ETDs VMs. Ceph is a distributed *object*, *block*, and *file system* storage that is widely used for its scalability, flexibility, and reliability. Ceph is supported by many cloud computing vendors and is widely used in modern data centers.

4.4 System Architecture

Figure 3 shows the structure and the components of the information system that is designed in this project.

The main metadata of the current system are for ETDs and Tobacco Settlement Documents, which are sourced respectively from Universities (Virginia Tech, for now) and the UCSF Deposition Documents.

Both of the sets of documents are pre-processed to extract useful information like text data and metadata. Kibana and Elasticsearch, in conjunction, provide the framework for data analysis. Furthermore, the system works on analyzing user logs to provide efficient recommendation.

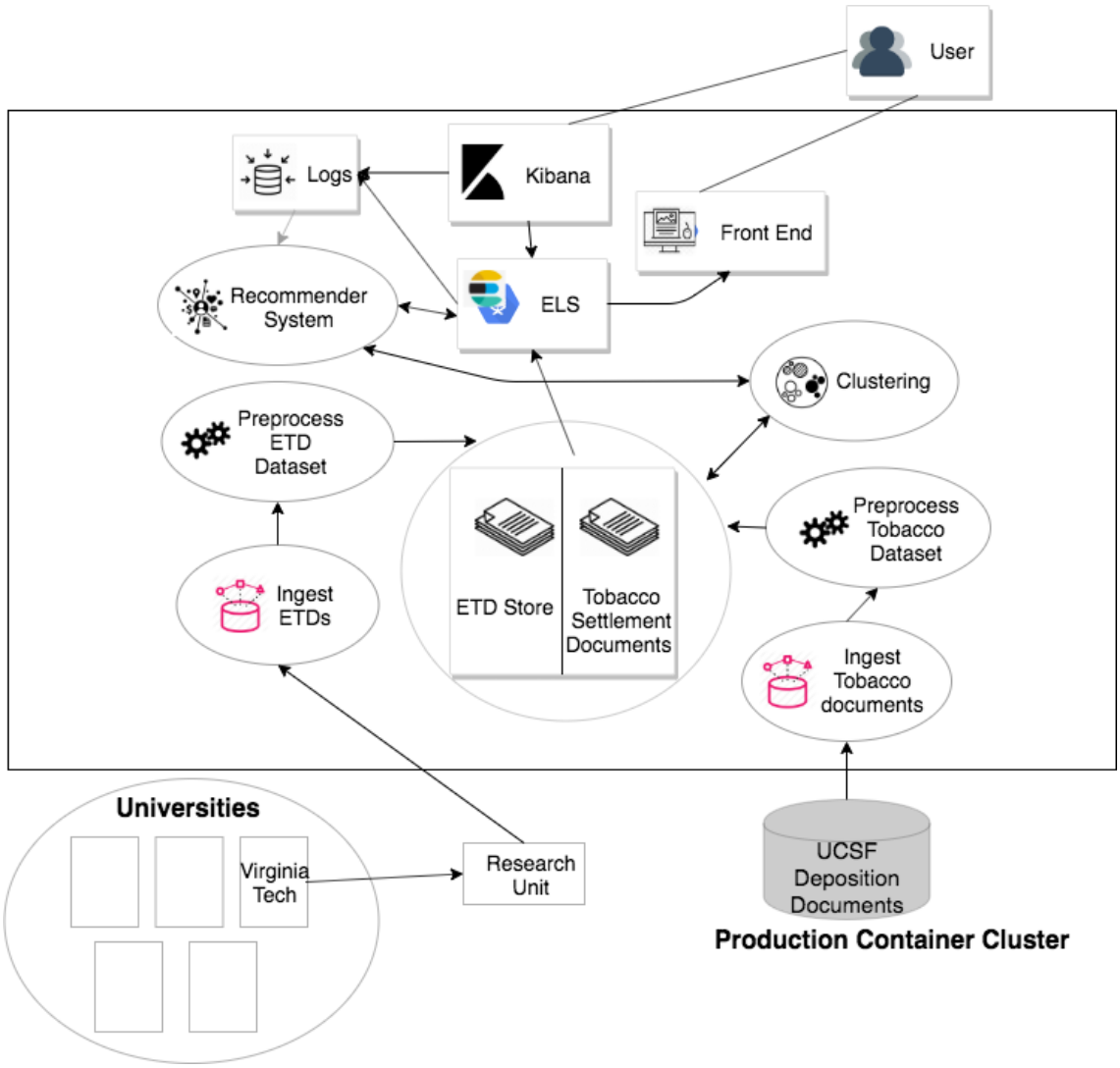


Figure 3: System diagram for information retrieval and analysis system

4.5 Data Ingestion

Because both data collections in our system, ETDs and Tobacco Settlement Documents, are continually expanding, there is a need to handle the ingestion of new documents. There are both front-end and back-end components to implementing the ingestion system. The discussion that follows describes the ideal system that we pursued to implement.

The front-end component is to be a web interface created by the FEK team that will be specific to administrators for each collection. This user interface ideally will support two options. The first option facilitates uploading a single document for a specified collection to be queued for ingestion. The second option allows specifying a source for a batch of documents to be queued for ingestion. The queued documents will be processed by the back-end component of the ingestion system.

The back-end of the ingestion system centers around the use of Apache Kafka [48]. Apache Kafka is a distributed publish-subscribe messaging system. It is a fast, scalable, design-integrated, distributed, partitioned, and replicated commit log service [21].

Apache Kafka's architecture includes the following components:

1. Topic: A specific type or category of message flow;
2. Producer: Any object that can post/publish messages to any topic;
3. Broker: To buffer unprocessed messages and to decouple data processing from data producers and consumers, published messages are stored in a Kafka cluster, which is a group of servers that are called brokers;
4. Consumer: It can pull data from/subscribe to one or more topics, that are maintained by the Broker, to consume published messages.

The back-end of our ingestion system will be configured such that the front-end will feed or link the source of the provided documents to Kafka as the producer for queuing into *Raw Tobacco/ETDs* topics and/or have Kafka connectors send new records to these topics once added to the databases that host raw ETD and tobacco data as shown in Figure 4. The CME and CMT data processing containers will be configured as consumers for the queued documents for their respective data collections.

The CME and CMT data processing containers will also serve as a second layer of producers for a second level of queuing of newly processed documents for indexing into Elasticsearch. The processed documents are stored in *Processed Tobacco/ETDs* topics. The Elasticsearch container will be configured as a consumer of those topics. The TML data clustering container will consume raw data for clustering. The TML clustering container then becomes a producer and sends the clustering results to Kafka's *Clustering IDs* topic to be consumed by Elasticsearch. TML also performs text summarization on the tobacco documents only as ETDs already have an abstract that summarizes the full text. TML sends summarization results into the *summaries* topic to be consumed by ELS. Finally, as was planned if time permits, the FEK containers will send logs such as user logs to the *Logs* topic to be consumed by TML to present recommendations.

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.

Using Kafka for queuing of documents also allows for scalability with respect to adding new collections to our system. 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 consume produced topics in real time.

4.5.1 Changes to System Architecture After Including Apache Kafka

Apache Kafka glues together our system's microservices. It serves as a changelog when processing data in real-time and at scale. Kafka can automate the process of adding new documents by monitoring a timestamp column in the source table to identify new and modified rows to facilitate real-time processing. To connect data sources with Kafka, we will integrate the ETD data VM as a source using a MongoDB Connector [31], and integrate the tobacco data VM as a source using Java Database Connectivity (JDBC) connector [22] to import (new) data from MongoDB and MySQL, respectively, into Apache Kafka topics.

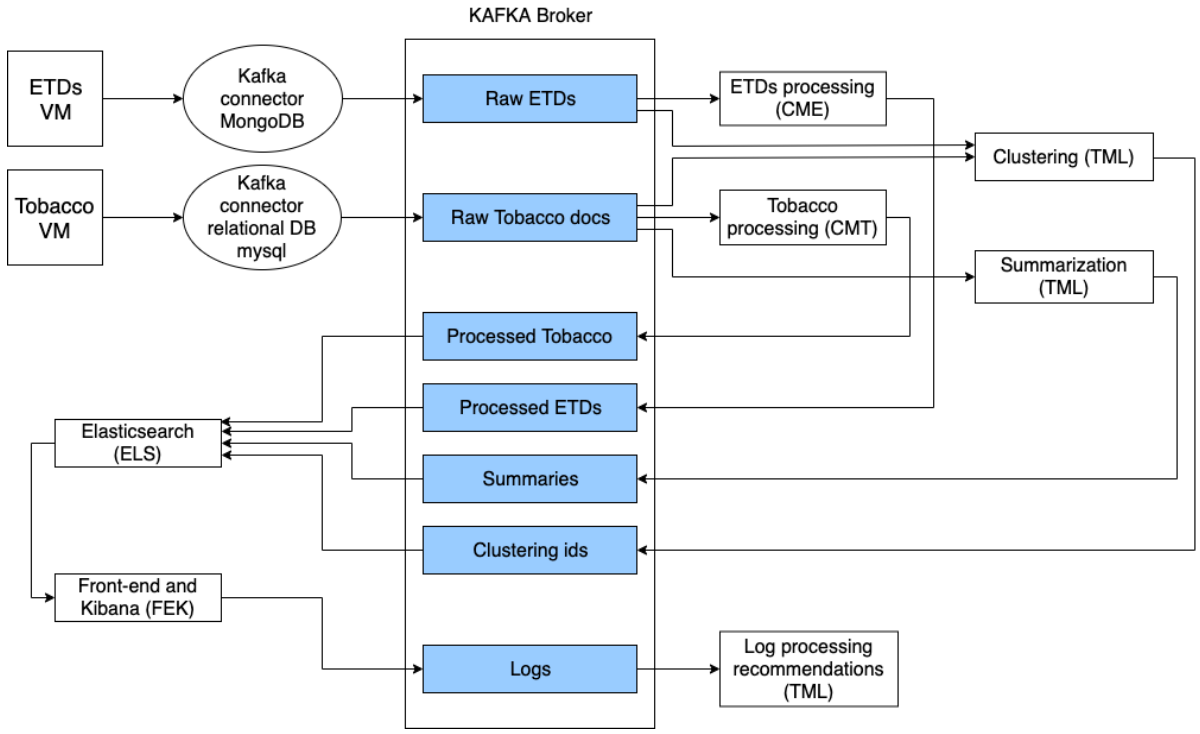


Figure 4: Kafka topics (blue) and producers and consumers

Kafka (and ZooKeeper [1] which maintains the Kafka cluster nodes and keeps track of Kafka topics, etc.) must be configured to store data on Ceph for persistence as Kafka and Zookeeper containers are ephemeral [26].

At every point in the containerized system, data producers must send the data to the respective topics, and consumers must subscribe to the respective topic as well. Kafkacat [50] is an open-source command line tool that enables such interaction with Kafka topics. It is used to produce, consume, and list topic and partition information for Kafka.

4.5.2 Monitoring new additions to directory using inotify-tools

The ELS team explored an alternative — a simpler way of automatically ingesting new documents by continuously monitoring the input directory via inotify-tools [49, 19].

inotify-tools is a C library and a set of command-line programs for Linux providing a simple interface to inotify [18]. It can be used to monitor and act upon file system events.

A Python script continuously listens to inotify-tools, which is configured to output the name of the new document been uploaded into the monitored directory. This way new documents can be referenced and ingested into the Elasticsearch pipeline once produced.

4.6 Continuous Integration & Continuous Deployment (CI/CD)

Continuous Integration and Continuous Deployment (*CI/CD*) lays out practices to follow in order for *unpushed* but committed code to more quickly be integrated while maximizing safety to result in the best experience for the end users. It is a software development methodology which allows

developers to push code multiple times a day instead of waiting till the end of a typical release cycle.

The advantage of this method is that users get small features released very quickly. From the perspective of developers, it is the management of small lines of code changes which helps minimize the instability of the production environment.

The need for *CI* in *CI/CD* can be better explained with the view that a developer can push all the changes to their code in the same place, with all the same processes being run on it. This makes the output code to be more coherent with the system.

Once all the code and the changes are in the same place, the same processes will run on the code repository. These *processes* could be:

- Run automatic code quality scans on it and generate a report of how well your latest changes adhere to good coding practices.
- Build the code and run any automated tests that you might have written to make sure your changes did not break any functionality.
- Generate and publish a test coverage report to get an idea of how thorough your automated tests are.

The process could be explained diagrammatically as shown in Figure 5.

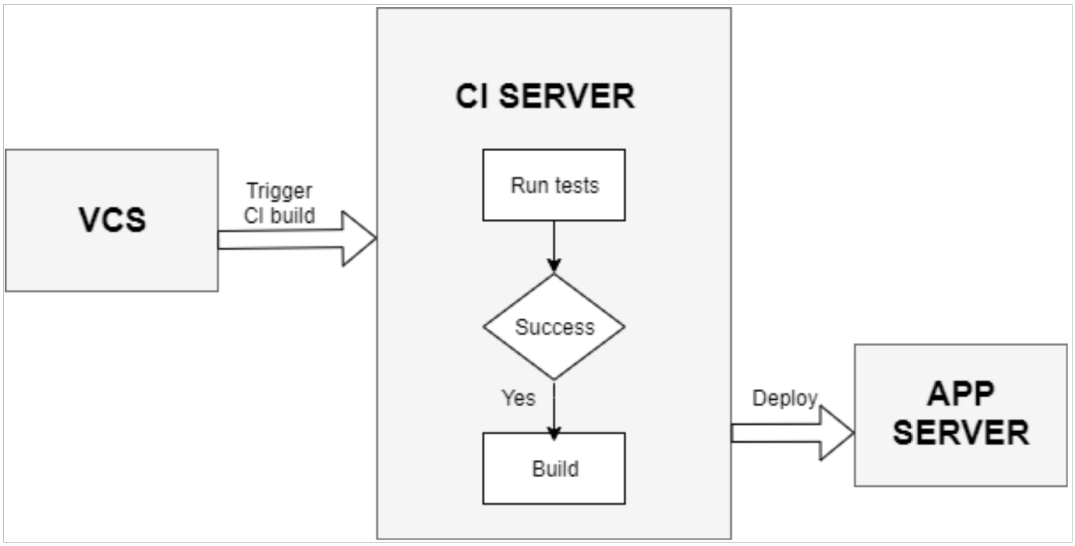


Figure 5: Continuous Integration & Continuous Deployment

For the *CI/CD* purposes of the project, we have explored three tools: Jenkins [23], Travis CI [38], and GitLab [13]. Table 1 provides a comparison between them.

Table 1: Comparison of CI tools

Feature	Jenkins	Travis CI	GitLab
Ease of Setup	Jenkins needs elaborate setup. Long wait time is required for the complete installation.	Setting up is as easy as creating a config file.	GitLab provides easy setup with a *yaml* file.
Hosted service	No	Yes	Yes
Performance	Has unlimited customization options.	Best choice for open-source project because of ease of use and setup.	Supports public and private repositories along with a lot of other options like supporting container registry.
Tool Type	Open-source	Commercial	Open-Source
Usage	Free	Free for Open Source Project and paid for Enterprise	Free for public projects and offers different kinds of membership to organizations such as educational institutions. GitLab is thus free to use for this project.
Server Machine	Server-based	Cloud-based	Cloud-based
Support for cloud services	Yes	Yes	Yes
Notification services	Yes	Yes	Yes

4.6.1 Introduction to GitLab CI/CD

We initially started working with Travis CI because of the ease of use, but then due to its large number of limitations we shifted to GitLab.

GitLab CI/CD is a powerful tool built into GitLab that allows you to apply all the continuous methods (Continuous Integration, Delivery, and Deployment) to your software with no third-party application or integration needed.

To use GitLab CI/CD, all you need is an application codebase hosted in a Git repository, and for your build, test, and deployment scripts to be specified in a file called .gitlab-ci.yml, located in the root path of your repository.

In this file, you can define the scripts you want to run, define include and cache dependencies, choose commands you want to run in sequence and those you want to run in parallel, define where you want to deploy your app, and specify whether you will want to run the scripts automatically or trigger any of them manually.

4.6.2 Design of our GitLab CI/CD pipeline

Once you've added your `.gitlab-ci.yml` configuration file to your repository, GitLab will detect it and run your scripts with the tool called GitLab Runner, which works similarly to your terminal.

We installed our GitLab runner on a dedicated VM (cicd.vt.edu) using the official documentation [15]. Whenever there is a commit on any repository that contains a `.gitlab-ci.yml` file located in the root path of the repository, it triggers a CI/CD pipeline on the GitLab runner. The Runner then performs the sequence of operations as listed in the configuration file. A basic example would be to create an environment to run the test suite and once all the tests passes, build and deploy the application according to the specified set of instructions.

Figure 6 shows a workflow diagram of the system CI/CD pipeline.

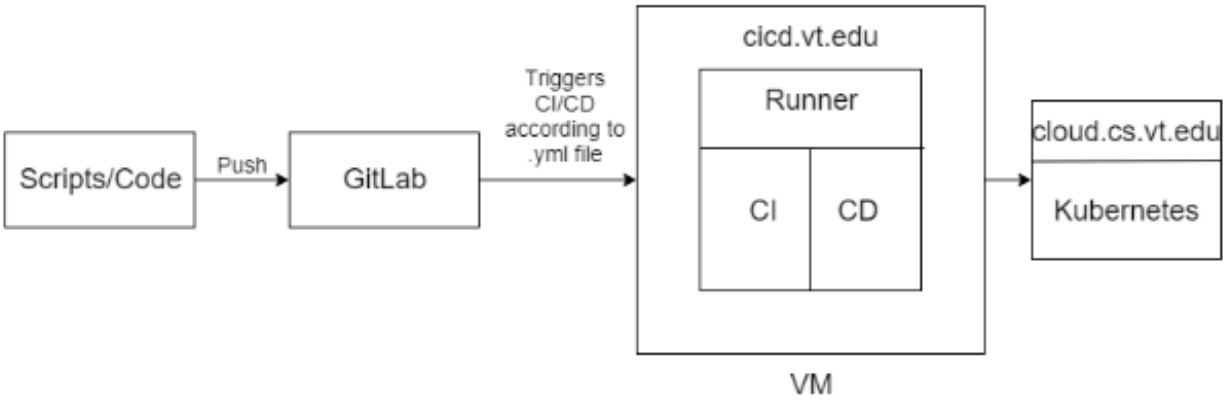


Figure 6: GitLab CI/CD pipeline

5 Implementation

5.1 Timeline

Table 2 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 2: Tasks and Timeline

Task	Timeline (week)	Assignee	Status
CS Cloud project creation for each team with Rancher	1	ALL	DONE
Student assignments as owners for their corresponding projects	1	ALL	DONE
Deployment of a CentOS container under each project	2	ALL	DONE
Ceph client installation on ETD and Tobacco VMs (virtual machines) and mounting of CephFS for cloud storage access	3	ALL	DONE
Creation of pods (containers) for the ELS and FEK 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 and Kibana tutorial	4	Lixing	DONE
Creation of a Kubectl tutorial	4	Hadeel	DONE
Creation of a tutorial to provide Access to Ceph Persistent Storage from ETD and Tobacco VMs	4	Hadeel	DONE
Assessment of additional container requirements for each team that came out of their initial discovery processes	5-7	ALL	DONE
Implementation of additional container requirements for each team's container	6-8	ALL	DONE

Continued on next page

Table 2 – continued from previous page

Task	Timeline (week)	Assignee	Status
Creation of a tutorial for deploying custom Docker container and exposing external IP	7	Rahul	DONE
Creation of a tutorial for leveraging Rancher's Catalog and App to deploy ready-made containers	8	Rahul	DONE
Creation of a tutorial for leveraging Docker Hub to deploy ready-made containers	8	Lixing	DONE
Container testing, evaluation, and integration into the CS cloud Kubernetes cluster	6-8	ALL	Ongoing
Development of a system for automatic/direct inclusion of future new raw ETD and Tobacco data into our information system (time permitting)	7-8	ALL	Kafka Topics DONE; Needs producers and consumers
Evaluation study of system performance (time permitting)	9-11	Malabika	Load Test DONE
CS Cloud deployment	12	ALL	DONE

5.2 Milestones and Deliverables

Our milestones over time are shown in Table 3. We will provide deliverables as listed in Table 4. Table 5 lists the container images per team along with their associated function within the system.

Table 3: Milestones

Task #	Completion Date	Milestone
1	09/03	Setup of namespaces and team projects in CS testing cluster, with students each added to their group project
2	09/05	Setup of Docker containers for each team with Ceph mounted
3	09/09	Fixed the issue of not being able to connect Virtual Machines to Ceph
4	09/10	Prepared documentation/tutorial on installing kubectl and connecting to the CS cloud cluster
5	09/12	Discussed and constructed a directory structure for everyone to follow on Ceph

Continued on next page

Table 3 – continued from previous page

Task #	Completion Date	Milestone
6	09/19	Discussed and built various custom Docker containers
7	09/21	Documented and prepared important tutorials of the process being followed for future reference
8	10/1	Prepared tutorials for Jupyter Notebooks installation and exposing external IP
9	10/1	Prepared tutorial on deploying containers from Rancher catalogs
10	10/7	Deployed initial development versions of containers requested by other team
11	10/8	Prepared tutorial on committing changes to a new container image
12	10/10	Prepared tutorial on building a Docker image from a Docker file
13	10/10	Prepared tutorial on deploying containers from Dockerhub
14	10/15	Researched on how to change the Elasticsearch configurations for the FEK team
15	10/17	Deployed MySQL container and the Flask Application to get the first version of system running
16	10/22	Research on CI/CD and Kafka
17	10/29	Prepared tutorial and gave a demo on how to leverage Travis CI for achieving CI/CD

Table 4: Deliverables

Task #	Completion Date	Deliverables
1	09/5	Project setup with initial baseline containers for the other teams
2	09/19	Interim Report 1
3	09/20	Tutorials for how to use various containers
4	10/10	Interim Report 2
5	10/25	Docker container files
6	10/31	Interim Report 3
7	11/15	Integration of each team's efforts into an operating container environment
8	11/19	Demo on CI/CD using GitLab
9	11/28	Load testing
10	12/2	Ingestion system framework
11	12/11	Final Project Report

Table 5: Containers and their function in the System

Team Name	Container Image	Function in the System
CME	cme-grobid1	Preprocessing the ETD PDF documents
	cme-nltk	Natural Language Toolkit
	cme-scienceparse	Parses PDF and returns structured form
	cme-python3-gensim	Alternative tool for Preprocessing ETD documents
CMT	cmt-python	Processing Tobacco Settlement documents
ELS	elasticsearch-master	Support Elasticsearch
	kibana-kibana	Support Kibana
	els-python	Processing Data
FEK	fek-web	Web Interface for Front-End GUI
	kibana	Log Collection and Analytics
	mysql	Data Storage and Analysis
TML	anaconda	Processing Data
	tml-spacy	Natural Language Processing
	tml-jupyter	IDE for Python scripts
	tml-clustering	Running K-Means Clustering
	tml-hcluster	Running hierarchical Clustering

5.3 Methods Evaluation

In this section we discuss evaluations that led to the selection of best methods.

In order to utilize containers in the system, we had to have a way to access the container's CLI. The first option was the Execute Shell, which was easy to use as it was part of the Rancher UI. However, there is a one-hour time limitation to the shell, which leads to having all the operations–execution, processing, and data–to be lost. The second and currently used option is using kubectl (explained in §8.2).

We explored different methods of developing containers with specific tools and libraries. The first method was deploying a container with a base image (for steps on how to do so, see §8.3 and §8.4) and then accessing the CLI of that container (via Kubectl) and installing task-specific dependencies and tools. However, with this method, any changes made after basic-image deployment were not persisted and the changes were lost (due to container restart/reset).

To work around that challenge, we explored developing container images via Docker CLI, accessing the container system through CLI, installing tools, packages, and dependencies; and then committing changes via `Docker commit`. Committing changes to a Docker image results in saving the entire runtime into a new image. Then, the resulting image can be deployed in Rancher (as explained in §8.6).

Another method is to create and build containers by specifying a Dockerfile, which is the preferred way. Dockerfiles provide a recipe for Docker to create container images. They describe the tools, packages, and dependencies needed to run a Docker container. However, creating images via Dockerfiles is not simple. We must take into consideration the efficiency, size, and correctness

of the resulting image after building the Dockerfile. See §8.7 for a tutorial on creating Docker images via Dockerfiles.

5.4 Migration to Production

In a production environment, Docker makes it easy to create, deploy, and run applications inside of containers. Containers allow us to gather applications and all their core necessities and dependencies into a single package that we can turn into a Docker image and replicate. We use Dockerfiles to build Docker images. The Dockerfile is a file in which we can define what the image will look like, what base operating system it will have, and which commands will run inside of it (see §8.7 for more information).

Docker-compose is a tool for defining and running multi-container Docker applications. With Docker-compose, we can use YAML files to configure the application's services and push the custom-built Docker image to Docker Hub (see Figure 7).

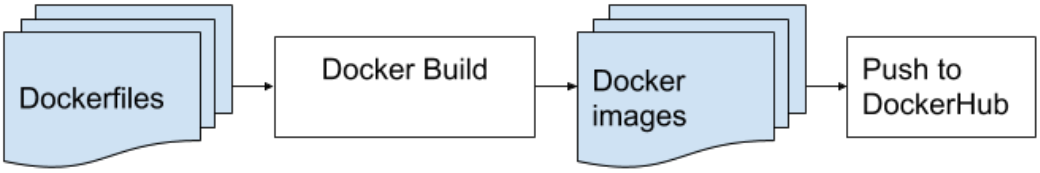


Figure 7: Building Docker images and pushing them to Docker Hub

However, instead of Docker-compose, we can use kompose [27], a conversion tool to move compose workflows to container orchestrators like Kubernetes. In addition to deployment on Kubernetes clusters, kompose supports both building and pushing Docker images. To use kompose, we will use the same YAML file we would use for Docker-compose (e.g., `docker-compose.yml` file). Running the `kompose up` command will start the deployment to Kubernetes directly (see Figure 8).

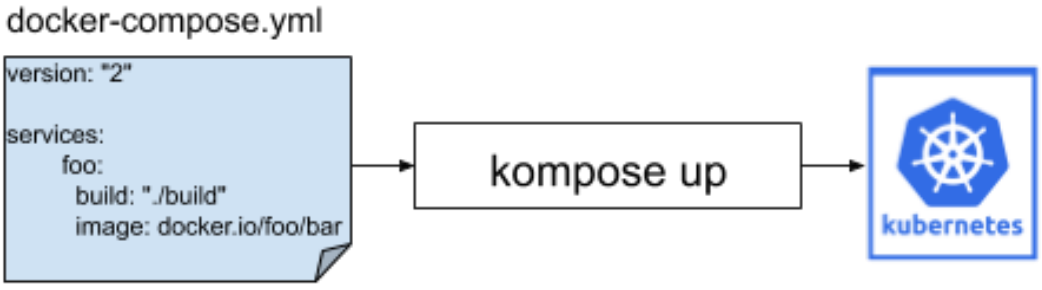


Figure 8: Building Docker images, pushing them to Docker Hub and deploying them on a Kubernetes Cluster

We can also use kubectl, but first we must convert the `docker-compose.yml` file to files that can be used with kubectl (by running `kompose convert`) and then we can use `kubectl apply` to deploy to Kubernetes [37]. Therefore, with a single kompose command, we can deploy all our system services on our Kubernetes cluster.

6 System Evaluation

6.1 Performance Evaluation

To measure the system's performance we perform a stress test for different components of the system. Such testing requires the collaboration of all the CS5604 teams. One form of stress test that we considered is to replicate logged sessions at higher speeds. Through trial and error we can measure the maximum throughput of our system to process user sessions. To do that, we can perform a stress test with the most recent user sessions and/or archived sessions that represent typical user activity. Another form of stress test can be performed when the system is deployed on a new cluster. For that, we can simulate archived user sessions at typical user load and/or at maximum user load to gauge the capabilities of the underlying system hardware. After evaluating our system and the resources with respect to time and learning curve, we decided to go ahead with load testing our system which we will be discussing in depth in §6.4.

6.2 Automatic Failure Recovery

Ideally, if a component of our information system fails we will be able to recover from the situation automatically. We can recover from application crashes that occur within containers provided that persistent data for the application is located on Ceph (or another persistent storage). In this case, the application can be restarted without significant problems as long as no critical data was held in memory or in temporary files at the time of failure.

One of the core features of Kubernetes is that it's designed to maintain the desired state defined by operators and app admins. A series of reconciliation loops constantly works to find the optimal path from the current state to the desired state for all components of the cluster [28].

From the system's perspective, handling a container failure is easy because several simple loops are always running. To handle a single failure of a highly available app, each loop

- ensures that the correct number of containers is running and creates any that aren't;
- ensures that all unassigned containers are assigned to nodes that are healthy;
- ensures that each node is running containers assigned to it;
- ensures that traffic is load balanced only to healthy back-end containers.

6.3 Failures That Require Manual Recovery

We cannot automatically recover from failures that occur due to systemic problems. The following are examples of systemic issues that may lead to failure:

- Problems with Pushed Updates: Servers within the cluster may fail as a result of activity from system updates. System updates can disrupt the system configuration. They can also break a library dependency chain that affects whether an installed application can run.
- Consumed System Resources: Cluster activity that consumes too many resources may cause cascading failures of uncoupled applications and/or containers. If the system resources are not freed, crashed applications and containers will not have the resources to be restarted.

Since systematic failures require manual intervention, the most that can be done is that the system may notify a system administrator, if the failure is such that the system is left partially running with the capability of detecting failures within its subsystems.

6.4 Stress Testing

As discussed before, Stress Testing is a type of performance testing which validates the highest limit of a system; which could be a computer, a device, a program or a network – with heavy load.

Stress testing tries to break the system under test by taking away the resources from it. Hence, it is also called as *negative* testing. The main purpose of doing a stress test is to see that the system fails and recovers gracefully – a quality which is known as *recoverability*.

A stress test helps verify the stability and reliability of the system. It makes sure that the system does not break under extreme conditions by forcing those conditions and in turn preparing the system for the worst. A stress test prepares the system to be robust, available at all times, and handle the errors.

Load testing is a type of stress testing which checks how systems function under a large number of concurrent simulated users accessing the system over a period of time. This is used to verify how systems handle heavy load volumes.

Load testing can be used to determine the number of users the system can handle. Load testing tools allow to create different user scenarios which is helpful to test different parts of the system, like the login page.

6.4.1 Stress Testing Tools

For our project, we have explored two stress testing tools: Locust[30] and JMeter[24]. See Table 6 for a comparison of the tools. Because of Locust's ease of setup and its support for Python, we decided to adopt it for load testing.

Table 6: Comparison of stress testing tools

Feature	JMeter	Locust
Scripting	Supports GUI and Scripting	Supports Python coding
Best For	Performance testing of Web Applications	It provides a functionality to check the simultaneous number the system can handle
Capability	It works for Web applications, Servers, Group of servers, and network	It can perform load testing on multiple distributed machines
Pricing	Free	Free
Open-Source?	Yes	Yes
Easy to use with VCS?	No	Yes

Continued on next page

Table 6 – continued from previous page

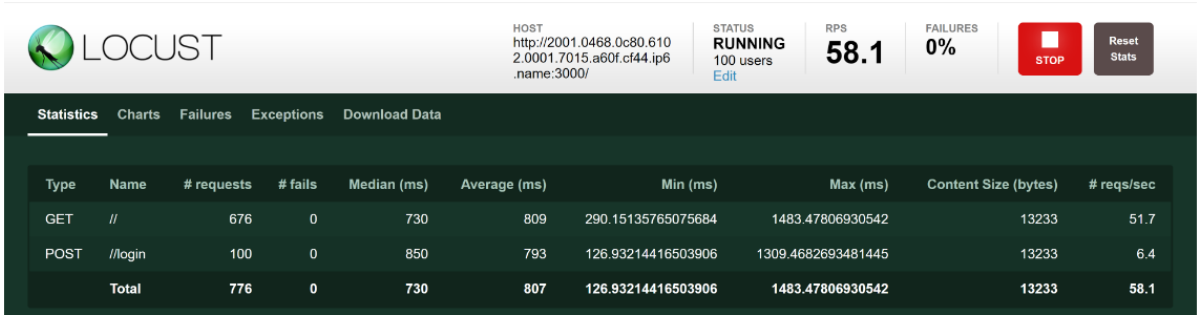
Feature	JMeter	Locust
Concurrency	JMeter is thread-based so it requires a separated thread to simulate a user. JMeter relies more on the machine's performance, and it is more expensive when simulating the same amount of users.	Locust is based on co-routine, and uses the async approach. It is possible to simulate large amount of users in one machine.
Support for distributed testing	Yes	Yes

6.4.2 Locust Stress Test Results

We used Locust to implement load testing. The general idea behind load testing is to see the number of maximum requests the front-end website could support and further implement load balancing techniques to manage the load.

The steps that we followed were:

1. Locust provides use of 2 parameters – number of users to be simulated and the *Hatch Rate* – which is the number of users spawned per second. In our experiment, we keep both the number of simulated users and the hatch rate to be the same number. Figure 9 shows the result of load testing when the website is simulated with 100 users with a hatch rate of 100 users per second.

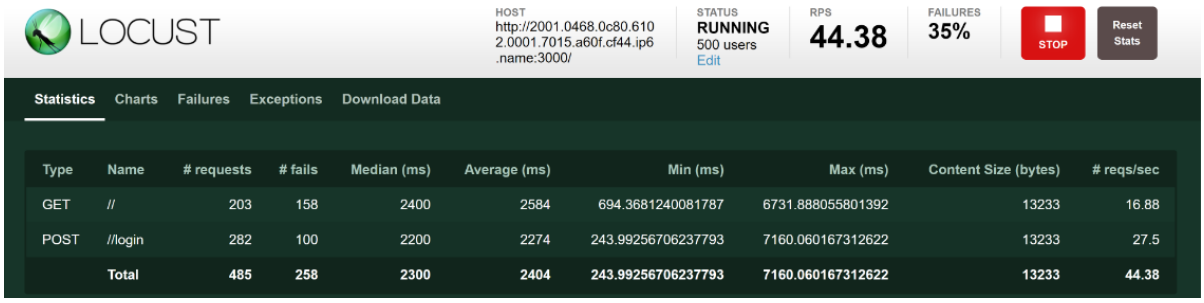


Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	//	676	0	730	809	290.15135765075684	1483.47806930542	13233	51.7
POST	//login	100	0	850	793	126.93214416503906	1309.4682693481445	13233	6.4
	Total	**776**	**0**	**730**	**807**	**126.93214416503906**	**1483.47806930542**	**13233**	**58.1**

Figure 9: Load testing of front-end website by simulating 100 users with 100 users spawned per second on one container.

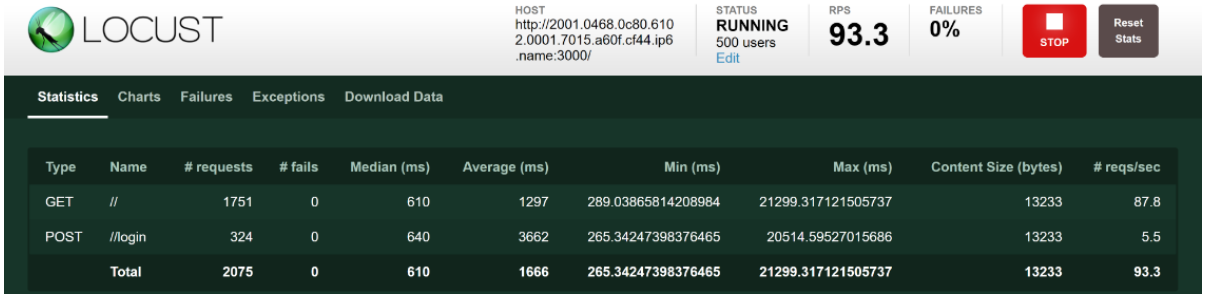
2. If the number of fail requests is zero, increase the number of simulated users. Since the front-end website was able to handle the initial 100 users as seen in Figure 9, we increased the number until the website failed to return some of the responses. We found that the website started sending fail requests at 500.

3. We then increased the number of container instances to two and carried on the load test with the last simulated number of users, i.e., 500.



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	//	203	158	2400	2584	694.3681240081787	6731.888055801392	13233	16.88
POST	//login	282	100	2200	2274	243.99256706237793	7160.060167312622	13233	27.5
	Total	**485**	**258**	**2300**	**2404**	**243.99256706237793**	**7160.060167312622**	**13233**	**44.38**

Figure 10: Load testing of front-end website by simulating 500 users with 500 users spawned per second on one container.

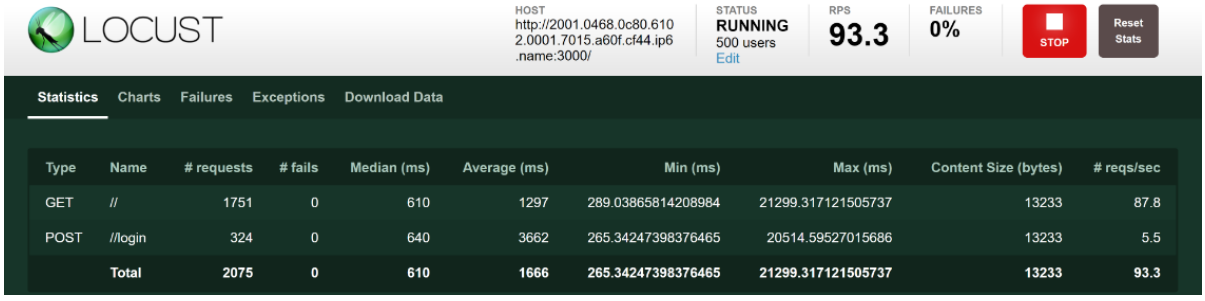


Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	//	1751	0	610	1297	289.03865814208984	21299.317121505737	13233	87.8
POST	//login	324	0	640	3662	265.34247398376465	20514.59527015686	13233	5.5
	Total	**2075**	**0**	**610**	**1666**	**265.34247398376465**	**21299.317121505737**	**13233**	**93.3**

Figure 11: Load testing of front-end website by simulating 500 users with 500 users spawned per second on two containers.

4. As seen from Figure 12, the number of fail requests is zero which means that the container was successfully able to balance the number of requests (500) amongst the two containers.

5. To test the limits of our system, we decided to further increase the number of users to 1000 and the number of fail requests still remained at zero. Thus, we can safely say that the front-end website can successfully handle requests from 1000 users, spawned per second if the number of containers is increased to two to balance the load.



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size (bytes)	# reqs/sec
GET	//	1751	0	610	1297	289.03865814208984	21299.317121505737	13233	87.8
POST	//login	324	0	640	3662	265.34247398376465	20514.59527015686	13233	5.5
	Total	**2075**	**0**	**610**	**1666**	**265.34247398376465**	**21299.317121505737**	**13233**	**93.3**

Figure 12: Load testing of front-end website by simulating 1000 users with 1000 users spawned per second on two containers.

7 User Manual

7.1 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*.

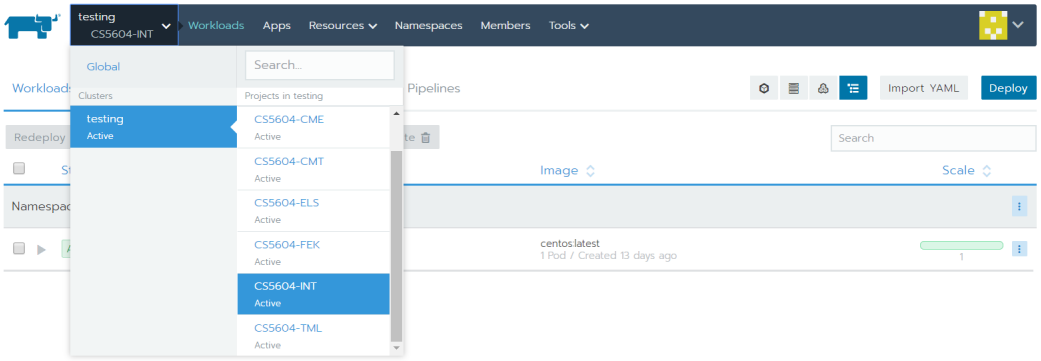


Figure 13: CS cloud cluster

2. A submenu appears under *testing* with a listing of all projects in which you have membership or ownership privileges. Figure 13 shows an example of a user with membership in all the six 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 14).

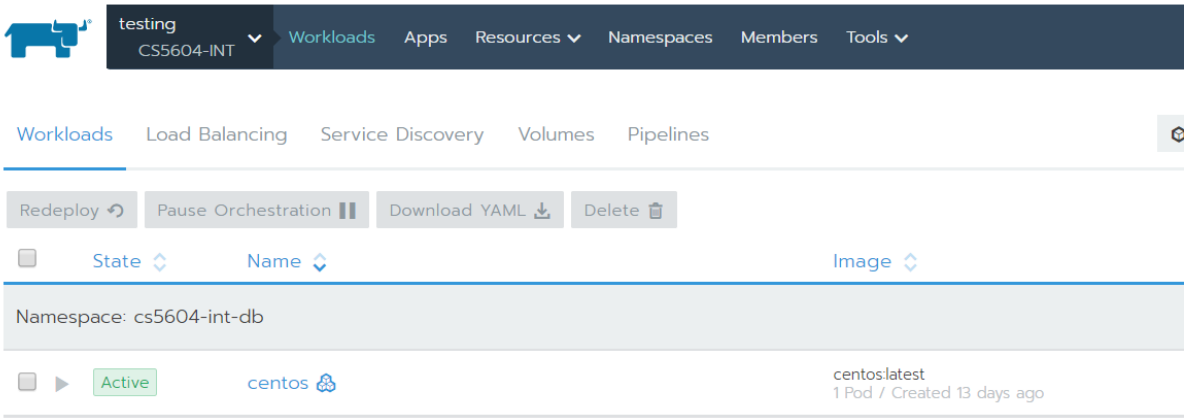


Figure 14: Pods and namespaces

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 15). 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 Ceph File System as a persistent volume is a way to save execution results.

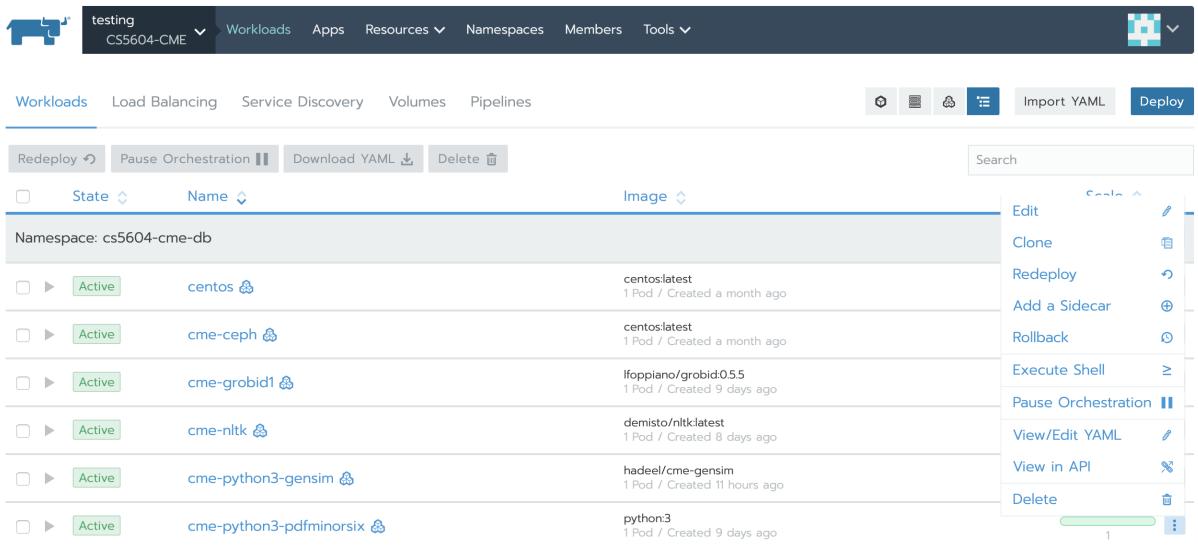


Figure 15: Container running entrance

5. In the container's shell, a user can execute commands such as running programs including running Elasticsearch and MySQL commands. In addition, you can also access the shared mounted volume (Ceph) via `cd /mnt/ceph` (see Figure 16). 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.

≥ Shell: cme-python3-pdfminorsix

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

```
root@cme-python3-pdfminorsix-548d895b4-lmhpx:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@cme-python3-pdfminorsix-548d895b4-lmhpx:/# cd mnt/ceph
root@cme-python3-pdfminorsix-548d895b4-lmhpx:/mnt/ceph# ls
cme  cmt  els  fek  int  shakespeare.json  shared  test  tml
root@cme-python3-pdfminorsix-548d895b4-lmhpx:/mnt/ceph# python -v
import _frozen_importlib # frozen
import _imp # builtin
import '_thread' # <class '_frozen_importlib.BuiltinImporter'>
import '_warnings' # <class '_frozen_importlib.BuiltinImporter'>
import '_weakref' # <class '_frozen_importlib.BuiltinImporter'>
# installing zipimport hook
import 'zipimport' # <class '_frozen_importlib.BuiltinImporter'>
# installed zipimport hook
import '_frozen_importlib_external' # <class '_frozen_importlib.FrozenImporter'>
import '_io' # <class '_frozen_importlib.BuiltinImporter'>
import 'marshal' # <class '_frozen_importlib.BuiltinImporter'>
import 'posix' # <class '_frozen_importlib.BuiltinImporter'>
import _thread # previously loaded ('_thread')
import '_thread' # <class '_frozen_importlib.BuiltinImporter'>
import _weakref # previously loaded ('_weakref')
import '_weakref' # <class '_frozen_importlib.BuiltinImporter'>
# /usr/local/lib/python3.7/encodings/__pycache__/__init__.cpython-37.pyc matches /usr/local/lib/python3.7/encodings/__init__.py
# code object from '/usr/local/lib/python3.7/encodings/__pycache__/__init__.cpython-37.pyc'
# /usr/local/lib/python3.7/__pycache__/codecs.cpython-37.pyc matches /usr/local/lib/python3.7/codecs.py
# code object from '/usr/local/lib/python3.7/__pycache__/codecs.cpython-37.pyc'
import '_codecs' # <class '_frozen_importlib.BuiltinImporter'>
import 'codecs' # <_frozen_importlib_external.SourceFileLoader object at 0x7fe30136d5d0>
# /usr/local/lib/python3.7/encodings/__pycache__/aliases.cpython-37.pyc matches /usr/local/lib/python3.7/encodings/aliases.py
# code object from '/usr/local/lib/python3.7/encodings/__pycache__/aliases.cpython-37.pyc'
import 'encodings.aliases' # <_frozen_importlib_external.SourceFileLoader object at 0x7fe30138b850>
```

Figure 16: Terminal of container

8 Developer Manual

8.1 Providing Access to Ceph Persistent Storage From ETD and Tobacco VMs

In order to be able to transfer large processed datasets from the virtual machines, that store the ETD and tobacco 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@tobaccovm`

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 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 Kubectl)
   `ls /mnt/ceph -l`
10. To exit the VM and go back to your machine
    `exit`

8.2 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. Each container is named `team-ceph`, so for the CME team, the container (and pod) is `cme-ceph`. Mounting Ceph File System allows us to persist and share data regardless of the ephemeral life of a container.

The following steps show how to install and work with Kubectl.
Steps 1-3 are for the CME, CMT, ELS, and INT groups – and possibly for TML.

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

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 17).

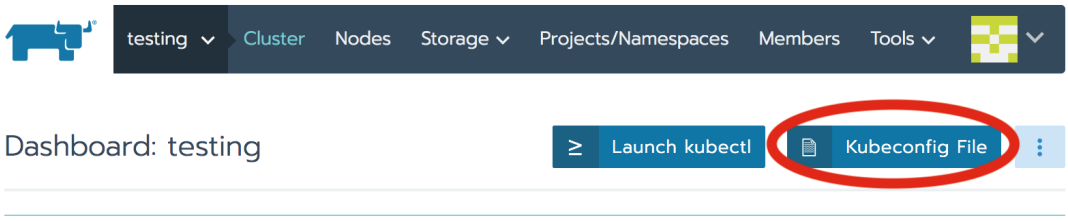


Figure 17: Kubectl configuration

   d. You'll see the following page; click on "Copy to Clipboard" (see Figure 18).

   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/`

4. Display the pods that are within the namespace cs5604-team-db. For example, for the CME team, the namespace is `cs5604-cme-db`.

   `kubectl get pods -namespace cs5604-cme-db` (see Figure 19 for the output)

5. Run a shell in the `cme-ceph-758b7ffccc-sn52n` pod to access Ceph File System.

   `kubectl exec -it -n cs5604-cme-db cme-ceph-758b7ffccc-sn52n /bin/bash`

6. Once in the container's shell, check disk space usage info of the container's file systems.

   `df -h`

   You can see Ceph is mounted on `/mnt/ceph` in Figure 20.

7. To access Ceph via the mount point, change to that location.

   `cd /mnt/ceph`

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

```
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"
```

Copy to Clipboard

Figure 18: Kubeconfig File

```
hadeel@hadeel-dssl:~$ kubectl get pods --namespace cs5604-cme-db
NAME                          READY   STATUS    RESTARTS   AGE
centos-5596447975-r9ln5       1/1     Running   0          8d
cme-ceph-758b7ffccc-sn52n     1/1     Running   0          8d
```

Figure 19: List of pods in the namespace `cs5604-cme-db`

8. List the contents of Ceph (see Figure 21).

   `ls`

9. You can store files in Ceph, e.g., we'll create a file called test.txt.

   `touch test.txt`

10. Don't forget to exit the pod shell once you're done.

    `exit`

The following steps are for the ElasticSearch Group:

1. To get the cluster IP address and External-IP for elasticsearch, list all resources under the namespace elasticsearch.

   `kubectl get all -n elasticsearch`

```
hadeel@hadeel-dssl:~$ kubectl exec -it -n cs5604-cme-db cme-ceph-758b7ffccc-sn52n /bin/bash
[root@cme-ceph-758b7ffccc-sn52n /]# df -h
Filesystem                                                                  Size  Used Avail Use% Mounted on
overlay                                                                     391G   23G  369G   6% /
tmpfs                                                                        24G     0   24G   0% /dev
tmpfs                                                                        24G     0   24G   0% /sys/fs/cgroup
128.173.41.10:6789,128.173.41.11:6789,128.173.41.12:6789:/courses/cs5604  2.8T   32M  2.8T   1% /mnt/ceph
/dev/mapper/centos-home                                                     391G   23G  369G   6% /etc/hosts
shm                                                                          64M     0   64M   0% /dev/shm
tmpfs                                                                        24G   12K   24G   1% /run/secrets/ku
tmpfs                                                                        24G     0   24G   0% /proc/acpi
tmpfs                                                                        24G     0   24G   0% /proc/scsi
tmpfs                                                                        24G     0   24G   0% /sys/firmware
[root@cme-ceph-758b7ffccc-sn52n /]#
```

Figure 20: List mounted volumes on the container cme-ceph-758b7ffccc-sn52n

```
[root@cme-ceph-758b7ffccc-sn52n ~]# cd /mnt/ceph/
[root@cme-ceph-758b7ffccc-sn52n ceph]# ls -l
total 35677
-rw-r--r-- 1 root root  2893015 Sep 14 23:55 2017_metadata.json
-rw-r--r-- 1 root root   242848 Sep 16 17:55 accounts.json
drwxr-xr-x 1 root root        0 Sep 16 23:28 cme
drwxr-xr-x 1 root root        0 Sep 16 23:28 cmt
-rw-r--r-- 1 root root  2689296 Sep 14 18:58 edt_metadata.json
drwxr-xr-x 1 root root        0 Sep 16 23:28 els
-rw-r--r-- 1 root root  2689296 Sep 15 20:38 eqt_metadata.json
drwxr-xr-x 1 root root        0 Sep 16 23:28 fek
drwxr-xr-x 1 root root        0 Sep 16 23:28 int
-rw-r--r-- 1 root root  2689296 Sep 16 17:33 metadata.json
-rw-r--r-- 1 root root 25327465 Sep 17 18:46 shakespeare.json
drwxr-xr-x 1 root root        2 Sep 16 23:27 shared
drwxr-xr-x 1 root root        0 Sep  4 22:01 test
drwxr-xr-x 1 root root        0 Sep 16 23:28 tml
[root@cme-ceph-758b7ffccc-sn52n ceph]#
```

Figure 21: Listing the contents of Ceph

2. Get a shell into the `els-ceph-dc4799796-d6wvj` pod.

   `kubectl exec -it -n elasticsearch els-ceph-dc4799796-d6wvj /bin/bash`

3. To connect Ceph with elasticsearch (using the ClusterIP)

   `curl 10.43.38.7:9200`

4. To check all the indices that are currently on Elasticsearch from the `els-ceph` pod (see Figure 22)

   `curl 10.43.38.7:9200/_cat/indices?v`

```
[root@els-ceph-dc4799796-d6wvj /]# curl 10.43.38.7:9200/_cat/indices?v
health status index                       uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   .kibana_task_manager        lYEWhsrKStmaSUHWF-5hOA   1   1          2            0      63kb         31.5kb
green  open   kibana_sample_data_ecommerce tyRLSIrYSau6EBzKUDtlTA   1   1       4675            0      9.8mb          4.8mb
green  open   shakespeare                 VUu2ub9FQgOJsNu1PLWacw   1   1     111396            0     40.5mb         20.2mb
green  open   kibana_sample_data_logs     Dd2nQ0jyR9miDukSrFmSqg   1   1      14648            0     24.2mb         12.1mb
green  open   .kibana_1                   SbJvmIgqSgS6h7eT_pi0nA   1   1         94            7       2mb            1mb
```

Figure 22: List of indices that are currently on Elasticsearch from the `els-ceph` pod

```
hadeel@hadeel-dssl:~$ kubectl exec -it -n elasticsearch els-ceph-dc4799796-d6wvj /bin/bash
[root@els-ceph-dc4799796-d6wvj /]# curl 10.43.54.87:9200
{
  "name" : "elasticsearch-master-0",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "M7gJSQVkSYi3THDYCTvIew",
  "version" : {
    "number" : "7.3.0",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "de777fa",
    "build_date" : "2019-07-24T18:30:11.767338Z",
    "build_snapshot" : false,
    "lucene_version" : "8.1.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
[root@els-ceph-dc4799796-d6wvj /]#
```

Figure 23: Connect Elasticsearch outside the testing cluster (using the External-IP)

5. To connect Elasticsearch outside the testing cluster (using the External-IP) (see Figure 23)

6. Exit the pod shell once you're done.

   `exit`

8.3 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 [3].

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

2. Under the Apps tab click on launch.

3. Search for a ready-made container that you want to deploy (see Figure 26).

4. Configure your App with required parameters like namespace, environment variables, volume, etc. (see Figure 27).

5. Execute the shell of the newly deployed App and type the command to test if it works properly or not (see Figure 28).

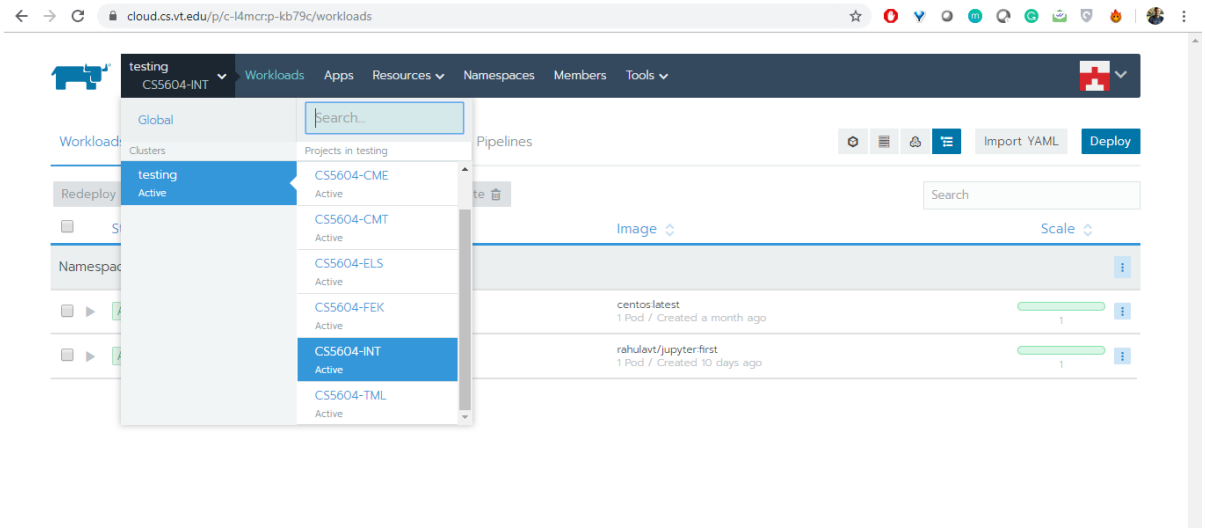


Figure 24: Navigating to CS5604-INT namespace

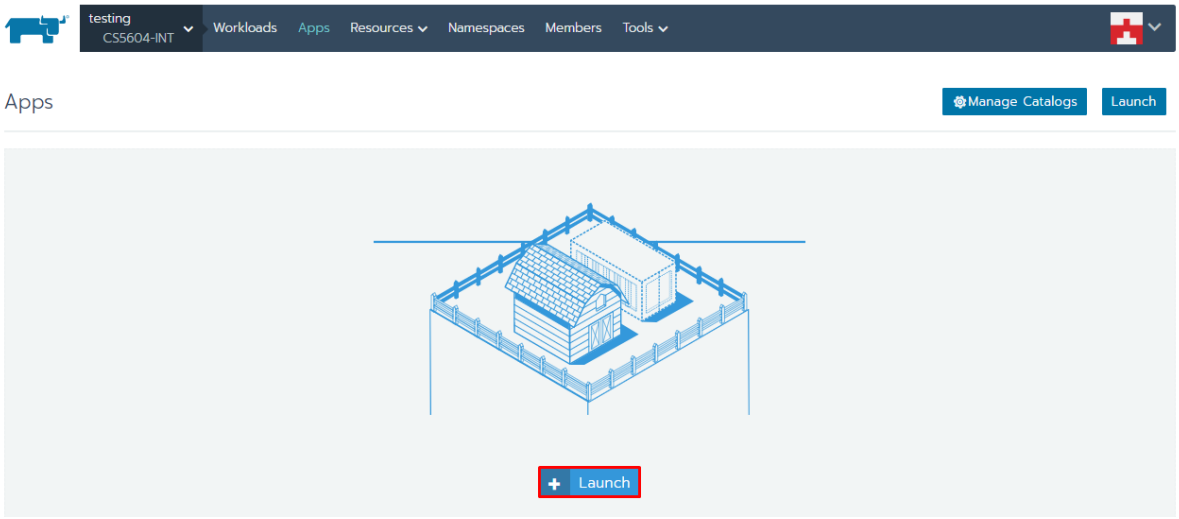


Figure 25: Launching a Rancher App

8.4 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 29).

2. Login to `cloud.cs.vt.edu` and navigate to the namespace where the application/container has to be deployed (see Figure 24).

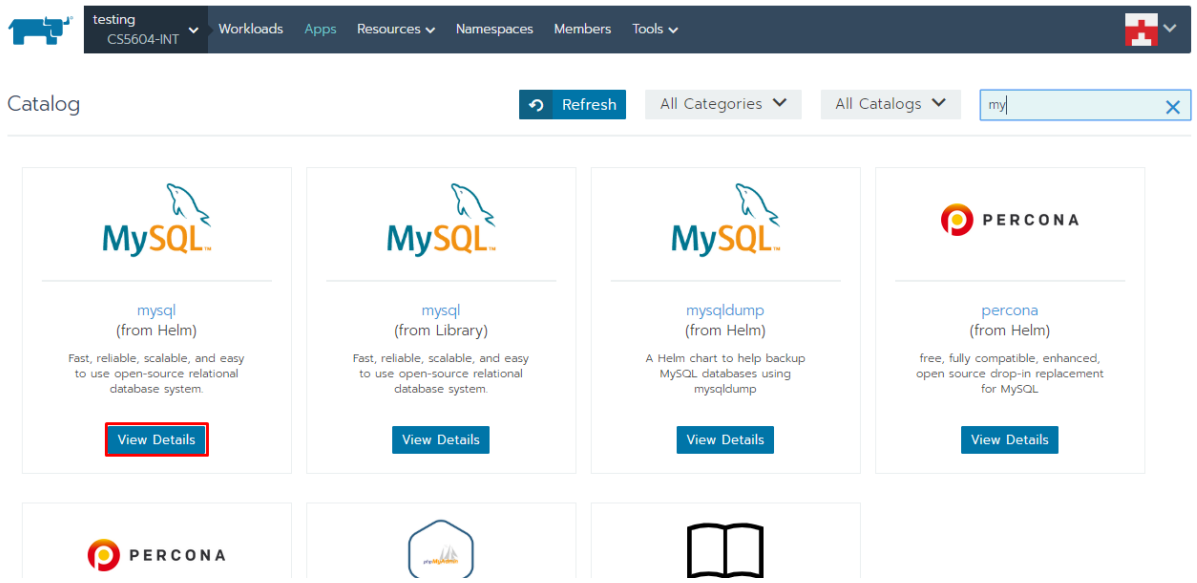


Figure 26: Launching MySQL App

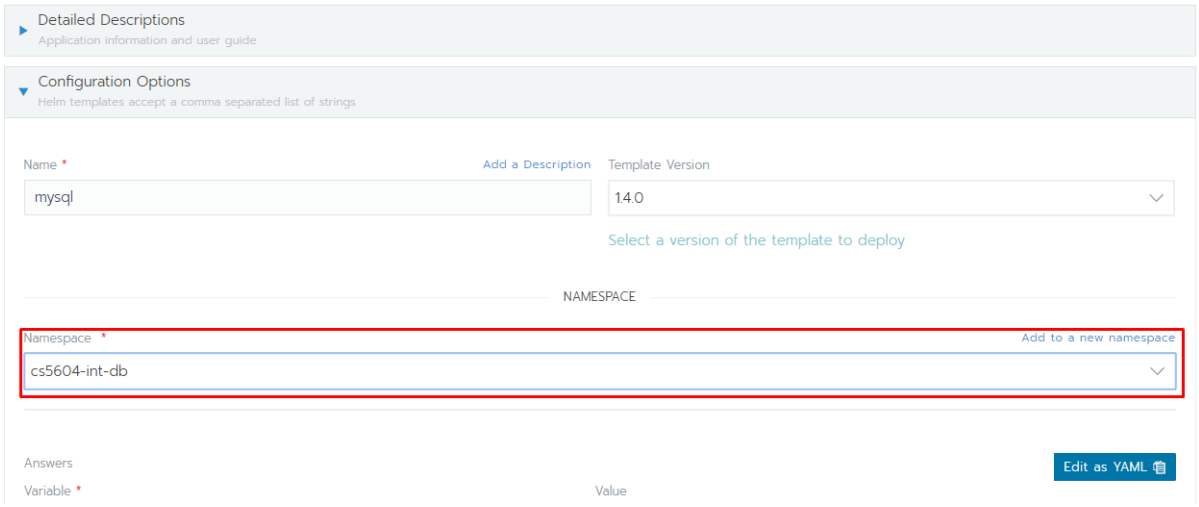


Figure 27: Configuring the App

3. Under the Workloads tab click on Deploy.

4. Configure your Container with required parameters like namespace, environment variables, volume, etc. (see Figure 27). 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 30). 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 (see Figure 16).

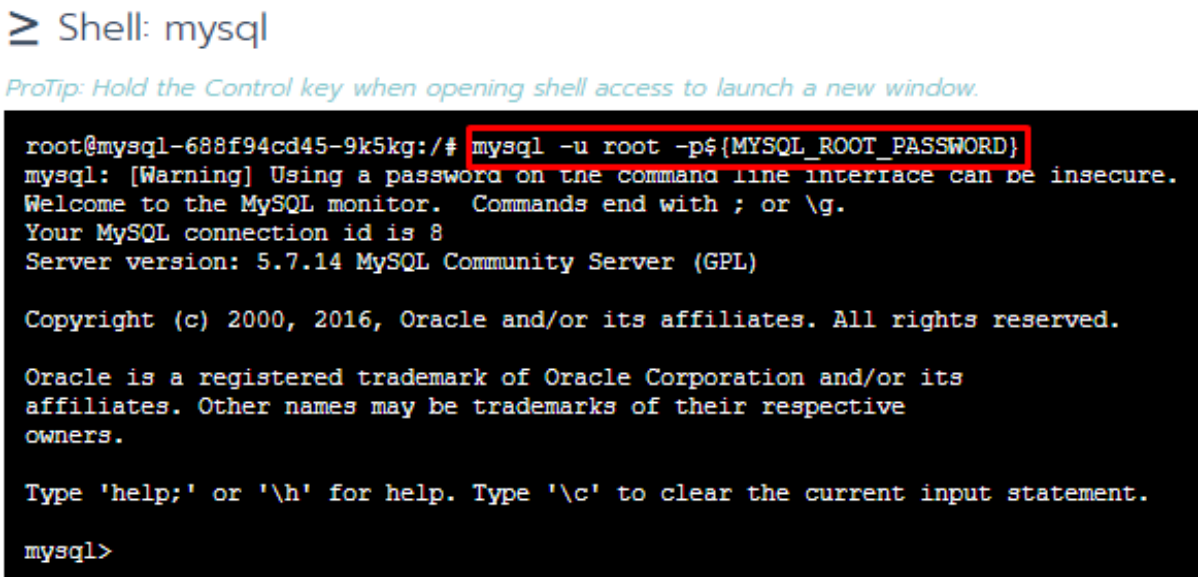


Figure 28: Testing if MySQL App has been launched successfully

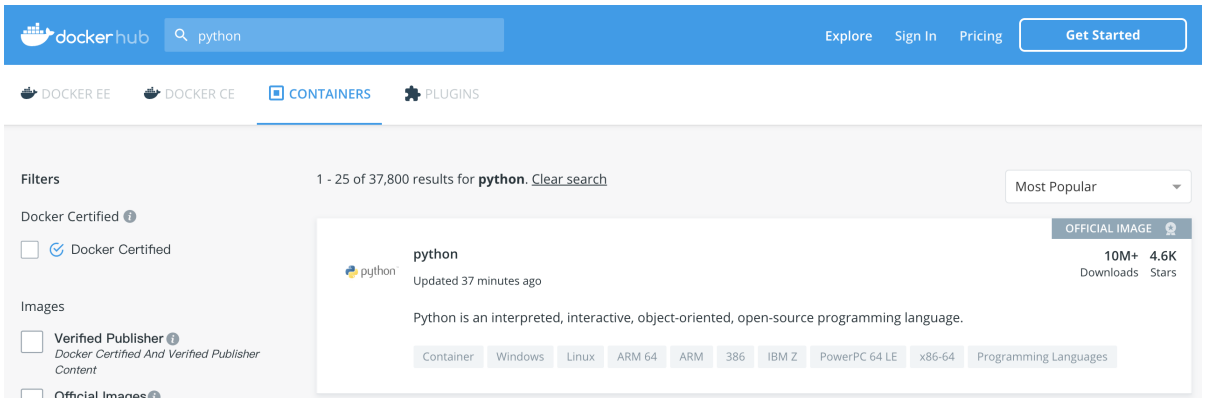


Figure 29: Search for container on Docker Hub

8.5 Jupyter Notebooks Installation and Exposing an External IP

In this section we will describe how to deploy a custom Docker container in the CS cloud cluster and expose it as a service for external access by using a static IP address and basic authentication.

What is Jupyter Notebook?

The Jupyter Notebook [25] is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. It's the most popular tool for interactively developing and presenting data science projects.

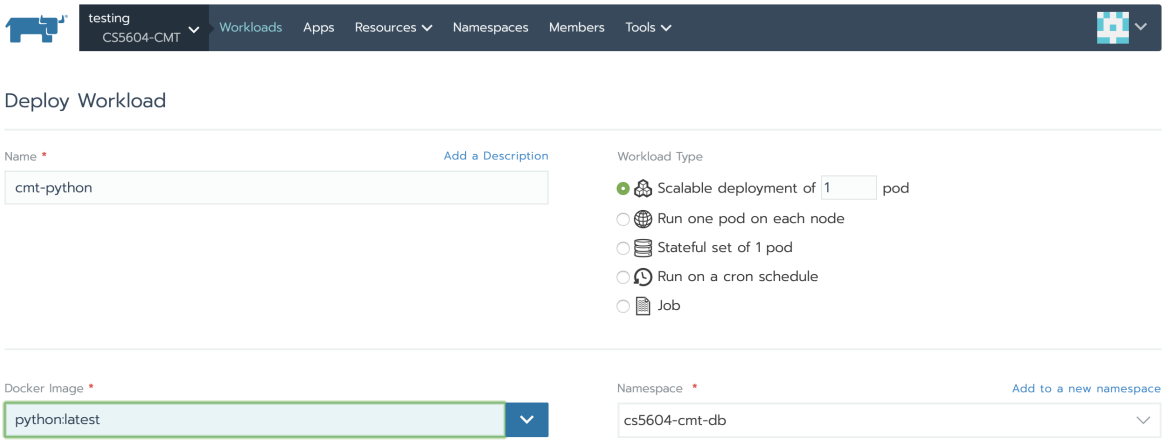


Figure 30: Search container on Docker Hub

Build and upload Docker images on Docker Hub

1. Create a Dockerfile in an new empty directory. The following commands should be inserted in an empty file named Dockerfile with no file extension:

   ```
   FROM python:3
   RUN pip install jupyter
   WORKDIR /mnt/
   ENTRYPOINT ["jupyter", "notebook", "-ip=0.0.0.0", "-no-browser","-allow-root"]
   ```

   Note that the `ENTRYPOINT` instruction allows you to configure a container that will run as an executable. It is similar to the `CMD` instruction, because it also allows you to specify a command with parameters. The difference is that the `ENTRYPOINT` command and parameters are not ignored when a Docker container runs with command line parameters.

2. To run the following command, Docker must be installed. In the directory where you created the Dockerfile above, run the following:

   ```
   $ docker build -t jupyter-docker .
   ```

3. You can run the container using the following command on your local system. We also need to mount the volume to store our notebooks outside the Docker directory.

   ```
   $ docker run -p 8888:8888 -v <path_to_your_local_directory>:
     <path_inside_docker> -it jupyter-docker
   ```

4. To publish the image to use in any deployment environment (CS cloud in our case), create an account on Docker Hub and login to it via the terminal using the command:

   ```
   $ docker login --username=<username>
   ```

5. Tag the newly created Docker image.

   ```
   $ docker tag 754a7069962a <username>/jupyter:first
   ```

Note that “754a7069962a" is the image ID. To list images use the command: `docker images`. “First” is the tag/version.

6. Push your container’s image to Docker Hub:

```
$ docker push <username>/jupyter:first
```

Deploy on Kubernetes Cluster (CS Cloud)

1. Login to CS cloud. Go to your namespace and click deploy. The remaining fields should be filled as shown in Figure 31.

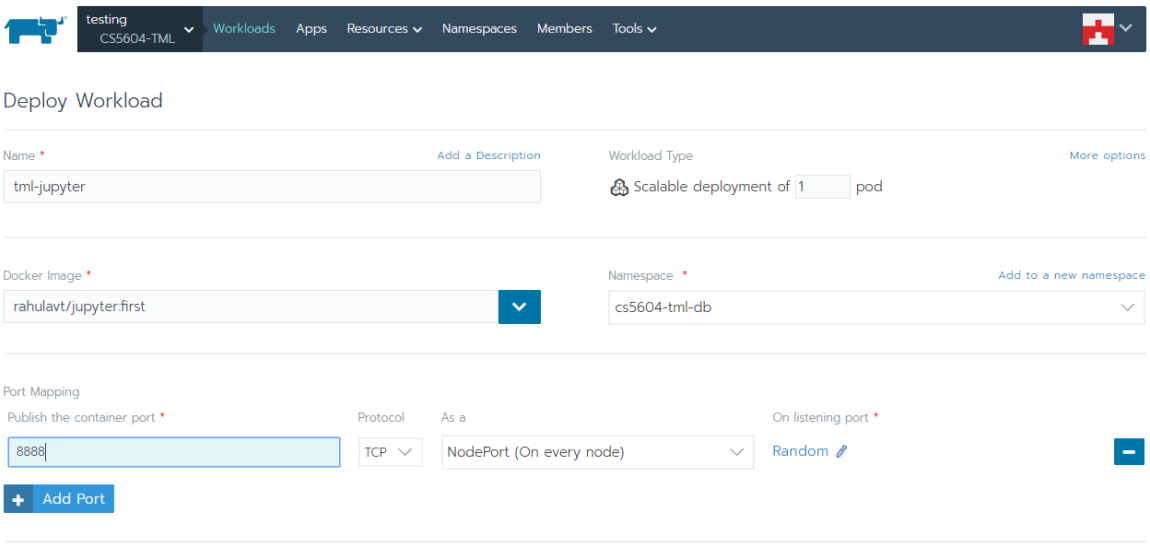


Figure 31: Deploying a container on Rancher

Note that "rahulavt" is a Docker hub username from where the Docker image can be pulled. Add a port 8888 because the application will run on 8888.

2. Attach a Ceph volume to where the Notebooks will be stored.

3. Once successfully deployed, Jupyter runs on localhost with some token. But we want to expose that to the outside world with the token as a security filter.

Create and expose a service using kubectl

1. In the terminal on your local system where you have installed and configured kubectl, and type [4]:

```
$ kubectl expose deployment tml-jupyter
  --type=LoadBalancer --name=jupyter-service -n cs5604-tml-db
```

2. Type the following command to see a list of all the services:

[4]Here we are creating a service of type LoadBalancer. For more details follow this article: [5].

```
$ kubectl get services -n cs5604-tml-db
```

```
rahul@DESKTOP-1KJ11Q3:/mnt/c/Windows/System32$ kubectl get services -n cs5604-tml-db
NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP                                                  PORT(S)          AGE
anaconda             NodePort       10.43.53.228    <none>                                                       8888:31637/TCP   2d22h
centos               ClusterIP      None            <none>                                                       42/TCP           24d
example-service      NodePort       10.43.196.103   <none>                                                       8080:30404/TCP   2d21h
jupyter              LoadBalancer   10.43.62.58     2001.0468.0c80.6102.0001.7015.0659.1c6a.ip6.name             80:30480/TCP     2d20h
jupyter-service      LoadBalancer   10.43.236.24    2001.0468.0c80.6102.0001.7015.5ebc.70b8.ip6.name             8888:31050/TCP   2d15h
my-service           LoadBalancer   10.43.50.187    2001.0468.0c80.6102.0001.7015.f163.50a9.ip6.name             8080:31918/TCP   2d20h
tml-ceph             ClusterIP      None            <none>                                                       42/TCP           17d
tml-jupyter          ClusterIP      10.43.35.142    <none>                                                       8888/TCP         2d15h
tml-jupyter-nodeport NodePort       10.43.208.94    <none>                                                       8888:31007/TCP   2d15h
tml-spacy            ClusterIP      None            <none>                                                       42/TCP           3d6h
```

Figure 32: List of running services for a given namespace

3. Now copy the external IP with port on your browser to open the Jupyter notebook.

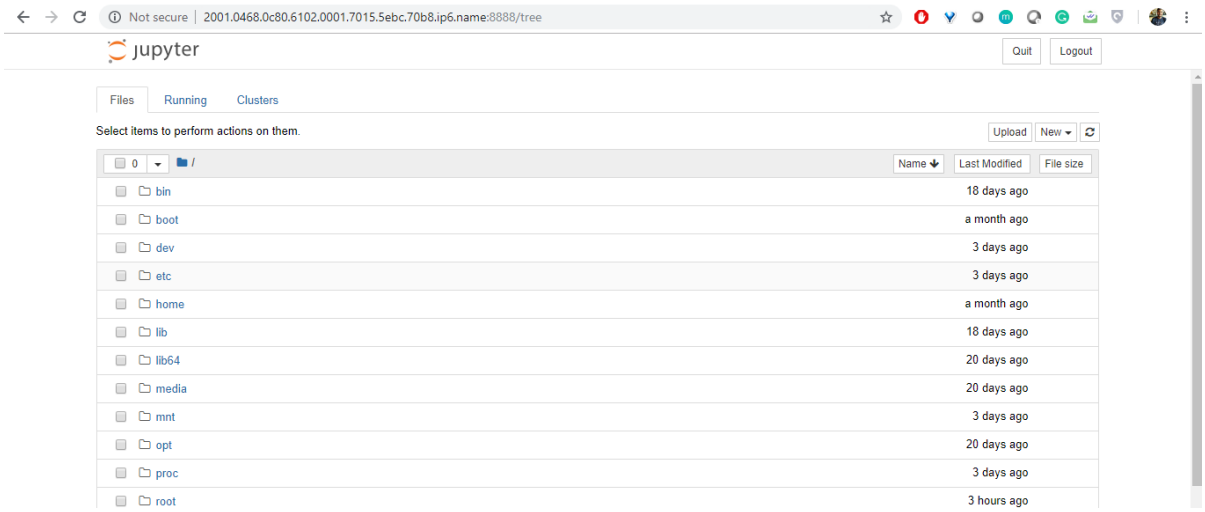


Figure 33: Jupyter Notebook's Interface

4. Once being asked for access token you can get it from the logs generated by the Jupyter pod in the CS cluster.

8.6 Committing Changes to a New Container Image

The purpose of this tutorial is to demonstrate how to incrementally develop Docker images in a local setting, specifically to save a running container's current state as an image by committing a container's modifications and changes into a new image.

The following instructions can run on any host system with Docker installed. To push an image to Docker Hub, you must have a Docker Hub account. To login to Docker Hub via the command line interface (CLI), use the following command:

```
docker login -username=Dockerhub-USERNAME -email=yourEmail@domain.com
```

Once you modify a Docker container using Docker CLI on a host machine, you can follow the steps to come.

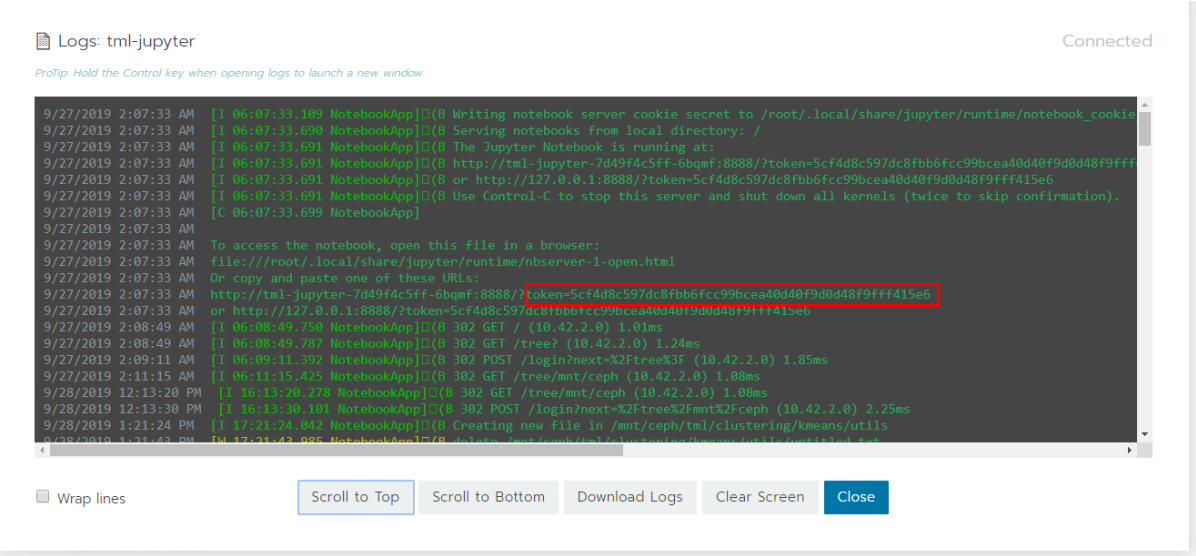


Figure 34: Accessing logs of a pod in Kubernetes

- Save modifications made in a container into a new container image. Use the ID of the container that was modified and name the new container image.

  `docker commit CONTAINER_ID NEW-IMAGE-NAME`

  The previous command will return the SHA256 hash of the image (see Figure 35).

```
hadeel@hab-mbp:~$ docker container ls
CONTAINER ID        IMAGE                                COMMAND
d8fb0894ad03        hadeel/cme-python3-gensim:latest   "/bin/bash"
hadeel@hab-mbp:~$ docker commit d8fb0894ad03 cme-gensim
sha256:8b4ebc36909277e004dcebd7fc3bda669626a173d2aec29b51c8521b01e9a94c
```

Figure 35: The commit command

To get the new image information:

`docker image ls`

See Figure 36 for a sample output. Notice the change in size in the new image compared to the original image.

```
hadeel@hab-mbp:~$ docker image ls
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
cme-gensim                  latest              8b4ebc369092        19 seconds ago      1.27GB
hadeel/cme-python3-gensim   latest              75f3694b69ee        33 minutes ago      918MB
```

Figure 36: Displaying the docker images

- To push the image to Docker Hub for fast distribution, you must tag the image as the following:

  `docker tag NEW-IMAGE-NAME Dockerhub-USERNAME/NEW-IMAGE-NAME`

Push the image to the registry using the image ID.

`docker push Dockerhub-USERNAME/NEW-IMAGE-NAME`

After that, the container image will be stored in Docker Hub with the repository name "`Dockerhub-USERNAME/NEW-IMAGE-NAME`". The new modified image can now be pulled and deployed on any host machine by using a `docker pull` command, or deployed on Rancher.

8.7 Building a Docker image from a Dockerfile

In this tutorial we explain how to build an image from a Dockerfile for container development. Running the built image will start a Docker container. A Dockerfile is a description of the software that constitutes an image. Dockerfiles contain a set of instructions that specify what environment to use and which commands to run. For demonstration, we will dockerize a simple Hello World Python application that uses Flask, an HTTP server for Python applications.

To follow this tutorial, you can use a local host and should have Docker installed.

1. Create a fresh directory (e.g., hello-world/). This directory defines the context of a Docker build. It will contain all of the things needed to build the Docker image.

2. In the hello-world directory, create a file named "Dockerfile" with no extension required. To the Dockerfile, write what is shown in Figure 37.

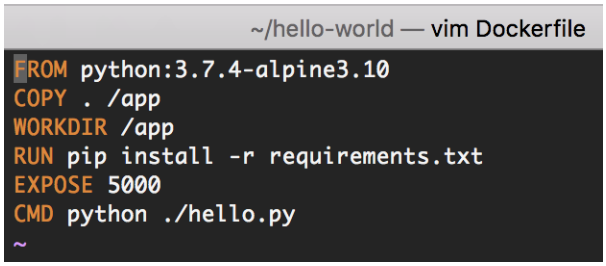


Figure 37: Dockerfile of Hello World Python application that uses Flask

The `FROM` instruction specifies the Base Image from which we are building. The `RUN` instruction specifies which additional commands to execute. The commands are in shell form (they will be run in a shell). The `CMD` specifies which command to execute when the image loads. There can only be one `CMD` instruction in a Dockerfile, and that can be in three different forms (explained in [10]). In this demonstration, we use the shell form.

3. In the hello-world directory, create another file named "requirements.txt". This file will contain the names of packages and dependencies needed to run the application. In this case, we need Flask (see Figure 38).

4. In the hello-world directory, create another file named "hello.py". This file will contain the Python code (see Figure 39). The IP address of the application is specified to be the localhost and the port is 5000.

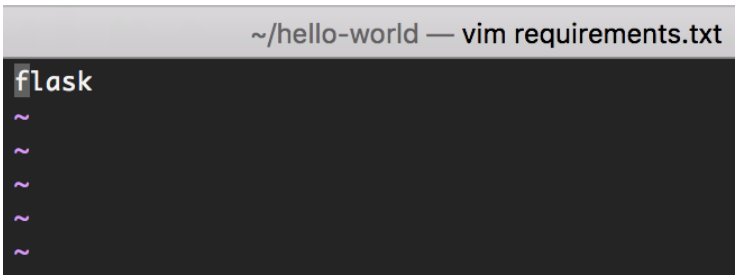


Figure 38: Contents of requirements.txt file

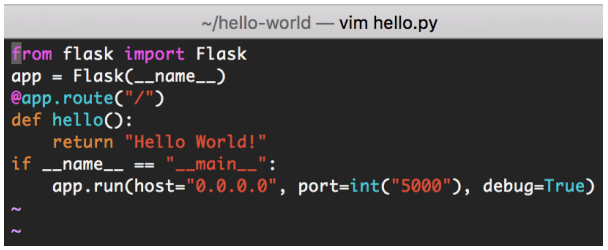


Figure 39: Python code of the Hello World! application that uses Flask

5. In the hello-world directory run the following command:

   `docker build -t flask-hello:1 .`

   Here flask-hello:1 is the name:tag of the Docker image that is being created.

6. The application is running now. Browse to `http://localhost:5000/`, see Figure 40.

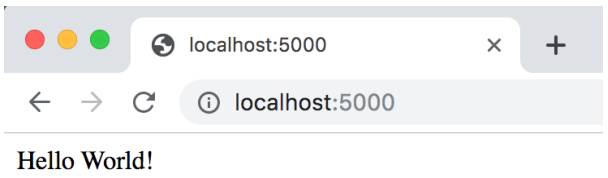


Figure 40: Hello World!

8.8 Implementing CI/CD with Travis CI

In the sections that follow, we will demonstrate step-by-step how to integrate an existing project with *Travis CI* [38] and set up a simple CI/CD process. We will create a simple hello world Python application for the purpose of this demonstration.

1. Connect Travis CI to your project. Most CI tools integrate seamlessly with Git services – especially GitHub. Thus, if you already have a repository on GitHub [12], add that to Travis CI, otherwise create a new repository. Your repository might look something like what is shown in Figure 41.

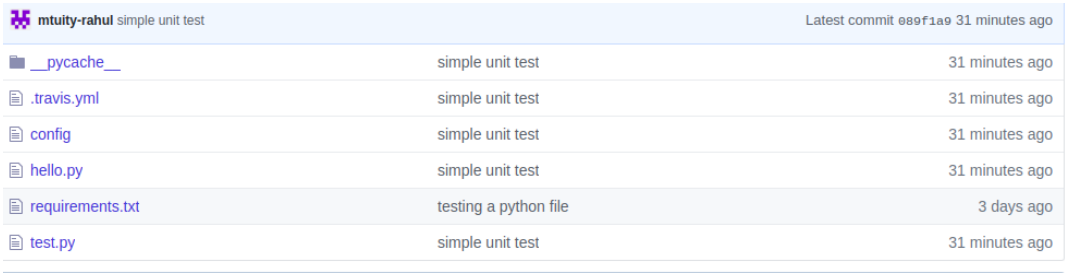


Figure 41: A sample GitHub repository

2. Most CI tools expect a configuration file to exist in the project. Travis CI expects a .travis.yml file to exist in the project root. The .travis.yml [5] file being used for demonstration would look like what we show in Figure 42.

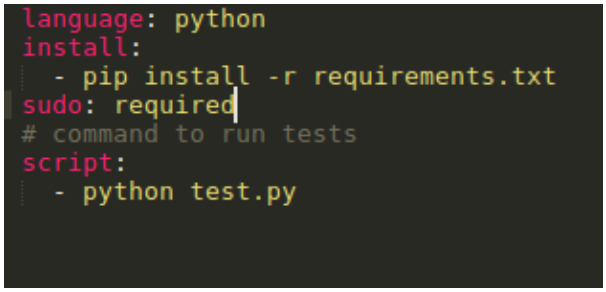


Figure 42: A sample .travis.yml file

3. Once the repository is ready, you do a git commit. The moment you push a change, Travis will trigger the build process (see Figure 43).

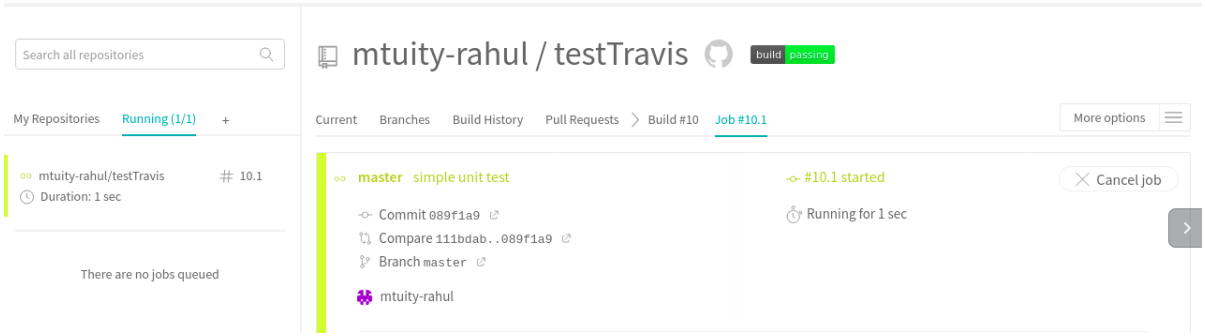


Figure 43: Travis build process

4. Once all test cases pass you will see a result as shown in Figure 44.

   Testing is just one step in the CI process. You can add more lines to the script stage, including processes like linting, E2E tests, performance tests, and any other metric you can think of to test new code before you merge it to the main code base [47].

[5] This has language: which is Python in this case, install: dependencies, and script: the set of unit tests to run.

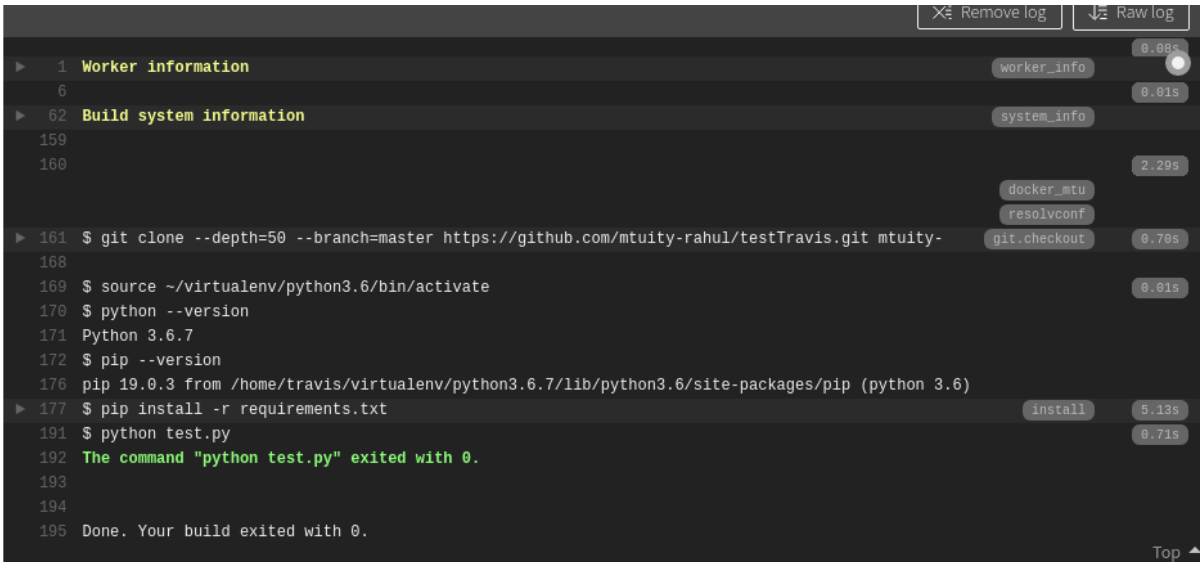


Figure 44: Travis build results

References

[1] Apache ZooKeeper. https://zookeeper.apache.org/, last accessed on Oct 28, 2019.

[2] Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, last accessed on Oct 21, 2019.

[3] Catalogs and Apps. https://rancher.com/docs/rancher/v2.x/en/catalog/, last accessed on Oct 27, 2019.

[4] Computer Science Cloud. https://cloud.cs.vt.edu, last accessed on Dec 8, 2019.

[5] Create a Kubernetes Service object that exposes an external IP address. https://unofficial-kubernetes.readthedocs.io/en/latest/tutorials/stateless-application/expose-external-ip-address/, last accessed on Oct 24, 2019.

[6] Deploy Kubernetes: The Ultimate Guide. https://platform9.com/docs/deploy-kubernetes-the-ultimate-guide/, last accessed on Dec 8, 2019.

[7] Docker. https://www.docker.com/, last accessed on Dec 8, 2019.

[8] Docker command for creating a new image from a container's changes. https://docs.docker.com/engine/reference/commandline/commit/, last accessed on Oct 21, 2019.

[9] Docker Hub. https://hub.docker.com/, last accessed on Dec 8, 2019.

[10] Docker Hub. https://docs.docker.com/engine/reference/builder/, last accessed on Oct 21, 2019.

[11] DockerSlim, Minify and Secure Your Docker Containers. https://dockersl.im, last accessed on Dec 8, 2019.

[12] Github. https://github.com, last accessed on Dec 8, 2019.

[13] GitLab. https://about.gitlab.com/, last accessed on Dec 11, 2019.

[14] GitLab CI/CD. https://docs.gitlab.com/ee/ci/, last accessed on Dec 8, 2019.

[15] GitLab Runner Docs. https://docs.gitlab.com/runner/, last accessed on Dec 12, 2019.

[16] GitLab Runner Helm Chart. https://docs.gitlab.com/runner/install/kubernetes.html, last accessed on Dec 8, 2019.

[17] Google's Container Registry. https://cloud.google.com/container-registry/.

[18] inotify. https://en.wikipedia.org/wiki/Inotify, last accessed on Dec 11, 2019.

[19] inotify-tools. https://github.com/rvoicilas/inotify-tools/wiki, last accessed on Dec 11, 2019.

[20] Install GitLab Runner using the official GitLab repositories. https://docs.gitlab.com/runner/install/linux-repository.html, last accessed on Dec 8, 2019.

[21] Introduction - Kafka Documentation. https://kafka.apache.org/082/documentation.html, last accessed on Oct 29, 2019.

[22] JDBC Source Connector for Apache Kafka. https://docs.oracle.com/en/cloud/paas/event-hub-cloud/admin-guide/jdbc-source-connector.html, last accessed on Oct 5, 2019.

[23] Jenkins. https://jenkins.io/, last accessed on Oct 27, 2019.

[24] JMeter. https://jmeter.apache.org/, last accessed on Dec 11, 2019.

[25] Jupyter Notebook. https://jupyter.org, last accessed on Oct 27, 2019.

[26] Kafka Connect Tutorial on Docker. https://docs.confluent.io/5.0.0/installation/docker/docs/installation/connect-avro-jdbc.html, last accessed on Oct 28, 2019.

[27] Kubernetes + Compose = Kompose. https://kompose.io, last accessed on Oct 25, 2019.

[28] Learn Kubernetes Testing Failure and Recovery. https://coreos.com/tectonic/docs/latest/tutorials/kubernetes/watch-recovery.html, last accessed on Oct 25, 2019.

[29] Linux Containers. https://linuxcontainers.org/, last accessed on Sep 28, 2019.

[30] Locust. https://locust.io/, last accessed on Dec 11, 2019.

[31] MongoDB Connector for Apache Kafka. https://www.mongodb.com/kafka-connector, last accessed on Oct 25, 2019.

[32] Nodes - cluster Architecture - Kubernetes Concepts. https://kubernetes.io/docs/concepts/architecture/nodes/, last accessed on Oct 27, 2019.

[33] Overview of kubectl. https://kubernetes.io/docs/reference/kubectl/overview/, last accessed on Nov 11, 2019.

[34] Pod Overview - Workloads - Kubernetes Concepts. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, last accessed on Oct 27, 2019.

[35] Quay. https://quay.io/, last accessed on Dec 8, 2019.

[36] Slack. https://slack.com, last accessed on Dec 8, 2019.

[37] Translate a Docker Compose File to Kubernetes Resources. https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/#use-kompose, last accessed on Oct 25, 2019.

[38] Travis CI. https://travis-ci.com/, last accessed on Oct 27, 2019.

[39] Trello. https://trello.com, last accessed on Dec 8, 2019.

[40] Zoom. https://zoom.us, last accessed on Dec 8, 2019.

[41] 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, last accessed on Dec 8, 2019.

[42] Mary Branscombe. Kubernetes Warms Up to IPv6. Feb 25 2019. https://thenewstack.io/kubernetes-warms-up-to-ipv6/, last accessed on Dec 8, 2019.

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

[44] Dave Bartoletti and Charlie Dai. The Forrester New Wave™: Enterprise Container Platform Software Suites, Q4 2018. https://www.docker.com/resources/report/the-forrester-wave-enterprise-container-platform-software-suites-2018, last accessed on Dec 8, 2019.

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

[46] 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/.

[47] Yonatan Kra. Setting up a CI/CD Process on GitHub with Travis CI, May 30, 2019. https://blog.travis-ci.com/2019-05-30-setting-up-a-ci-cd-process-on-github/.

[48] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[49] Robert Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[50] Magnus Edenhill. Kafkacat. https://github.com/edenhill/kafkacat, last accessed on Dec 9, 2019.

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

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

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

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

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

[56] Red Hat, Inc., and contributors. Ceph. https://ceph.io, last accessed on Dec 8, 2019.

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

[58] 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.

[59] Bill Ward. Ultimate Guide to Installing Kafka Docker on Kubernetes, Aug 13 2018. https://dzone.com/articles/ultimate-guide-to-installing-kafka-docker-on-kuber, last accessed on Dec 8, 2019.

[60] 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.