CS 5604 Migration from Virginia Tech Computer Science Container Cluster to Digital Library Research Laboratory

Authors

Casey Haney
Bethlehem Teshome
Camellia Pastore
Ashutosh Pant

Instructor: Dr. Edward A. Fox

May 13, 2021
CS4624 (Multimedia, Hypertext, and Information Access)
Department of Computer Science
Blacksburg, Virginia 24061
Abstract

The VTCS container cluster supports an information analysis system that can process and store data on web pages, tweets, and electronic theses and dissertations. This system was developed by previous semesters of Information Storage and Retrieval (CS5604) students and hosted in a Kubernetes cluster within the Virginia Tech Computer Science (VTCS) cloud. Our project goal was to migrate the system from the cloud to a cluster in the Digital Library Research Laboratory (DLRL), a lab dedicated to researching topics in information retrieval, multimedia, and hypertext.

The current cluster supports five main projects, each developed by a different team in CS 5604: Tweet Analysis, Electronic Theses and Dissertations, Web Pages, Integration, and Front End. The first three are the main collection management systems, designed to extract and analyze data from each category of written work that they are named after. The Front End project contains the interface that the user interacts with, and the Integration project ensures each of the other projects can communicate with the Front End and any other services they use, such as the search engine Elasticsearch.

Over the course of the project, we have reconstructed the cluster in DLRL as faithfully as possible. We have ensured each of the namespaces within the cluster hosts the same deployments, and these deployments are configured as closely as possible to their VTCS cloud counterparts. All adjustments made to the container configurations are documented herein. The cluster is fully set up for use; any future work on it would be reconfiguration at the discretion of future developers.
Acknowledgments

We would like to extend our gratitude towards Dr. Edward Fox for his constant guidance and help throughout this project. We would also like to thank our client Xinyue Wang for getting us started on this project, as well as Prashant Chandrasekar for support with the VTCS clusters. We also owe thanks to the CS5604 Fall 2020 class, whose thorough documentation of the applications and the cluster were invaluable to the migration process.
# Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 General Information 2

1.1.1 Containerization 2

1.1.2 Docker Containers and Images 2

1.1.3 Kubernetes Cluster and Kubernetes CLI 3

1.1.4 Rancher and Rancher CLI 3

1.2 Project-Specific Information 4

1.2.1 Computer Science and the Digital Library Research Laboratory 4

1.2.2 CS 5604 Containerized Applications 5

2 Requirements 6

2.1 User Goals 6

2.1.1 Accessibility of Cluster 6

2.1.2 Container Troubleshooting 6

2.1.3 Usable Documentation 7
2.2 Project Tasks ........................................ 7
  2.2.1 Transfer Containers to DLRL ................... 7
  2.2.2 Migrate Dependencies to DLRL .................. 7
  2.2.3 Verify Containers Actively Run .................. 8
  2.2.4 Compile Documentation Into Final Report ........ 8
2.3 Relevant Workflows .................................. 8

3 Implementation ....................................... 10
  3.1 Details of the Cloud Cluster ....................... 10
  3.2 Project Objectives .................................. 12
  3.3 Setting up the DLRL Cluster ....................... 13
    3.3.1 Persistent Volumes and Claims Creation .......... 14
    3.3.2 Image Deployment into DLRL ..................... 17
    3.3.3 Container Services .............................. 18

4 User Guide .......................................... 21
  4.1 Accessing the Front End ........................... 21
  4.2 General Usage ..................................... 22
  4.3 VT Cloud and General Containers Guide ............ 24
    4.3.1 Access Cluster Remotely With Rancher ........... 24
    4.3.2 Rancher CLI .................................. 26
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Docker Containerization [15]</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Kubernetes Architecture [10]</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>How Kubernetes uses Docker to run containers [1]</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Workflow to meet the first requirement of the users</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Workflow to meet the second requirement of the users</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Workflow to meet the third requirement of the users</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>View of DLRL Namespaces</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Example persistent volume yaml file</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of a persistent volume claim yaml file</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Example yaml file for a single image deployment</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Successful container deployments</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>Example yaml file for a container service</td>
<td>19</td>
</tr>
<tr>
<td>3.7</td>
<td>Example yaml file for a node port service</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of the DLRL Cluster</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>API option in Rancher user settings</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Add Key button in the API window</td>
<td>25</td>
</tr>
</tbody>
</table>
4.4 Add Key window. .............................................................. 26
4.5 Rancher CLI Download button. ......................................... 26
4.6 Rancher CLI ................................................................. 27
4.7 Address of a container image from its Docker registry ........ 27
4.8 Contents of a Dockerfile. .................................................. 28
## List of Tables

3.1 Table of Rancher Projects and their namespaces ......................... 10

3.2 DLRL Cluster Nodes ............................................................. 13

5.1 Deployments within cs5604-fe-db ............................................. 30

5.2 Deployments within cs5604-etd-db .......................................... 31

5.3 Deployments within etd ......................................................... 31

5.4 Deployments within cs5604-int-test ....................................... 32

5.5 Deployments within cs5604-twt-db ....................................... 32

5.6 Deployments within cs5604-wp-db ........................................... 33

5.7 Deployments within wp-test .................................................... 33

5.8 All services created for DLRL .................................................. 34

5.9 Current DLRL Deployment Errors ............................................. 37
List of Abbreviations

DLRL Digital Library Research Laboratory at Virginia Tech
ETD Electronic Theses and Dissertations
FE Front End
INT Integration and Implementation
NFS Network File System
TWT Tweet Analysis
VTCS Virginia Tech Computer Science
WP Web Pages
Chapter 1

Introduction

A container is software that packages the code and dependencies of an application, allowing the application to run in different computer environments [12]. This technology dates back to 1979 when the 7th version of Unix (Unix V7) first introduced chroot, or change root directory [16]. The chroot environment allowed containerization by creating and hosting a virtual copy of the system. From that point, different iterations of containers and container managers were introduced, but it was not until 2013 that Docker emerged and popularized containers. Docker gained such popularity because it offered an entire ecosystem to manage containers, making the deployment and use of containers simpler than previous systems. A year after Docker was introduced, Google released Kubernetes, an open-source system for automating deployment, scaling, and management of containerized applications [3]. In this project, we will be using Docker and Kubernetes for the migration of Kubernetes managed containers from one container cluster to another.

This project aims to transfer container images from Virginia Tech’s Computer Science cloud (VTCS cloud) to a local environment in Virginia Tech’s Digital Library Research Laboratory (DLRL). The images in question are the outcome from the Fall 2020 CS5604 course. They include the following images: Electronic Theses and Dissertations (ETD), Twitter (TWT), Web Page archives (WP), Front-End (FE), and Integration and Implementation (INT). The intent upon successful migration is for work with those containers to progress in the DLRL.
1.1 General Information

The following information is meant to supplement understanding of our project and define specific terms and technologies that will be helpful for the user to understand our project.

1.1.1 Containerization

As mentioned earlier, a container is software that packages the code and dependencies of an application, allowing the application to run in different computer environments [12]. The process of packaging the code and its dependencies is referred to as containerization.

1.1.2 Docker Containers and Images

Docker runs containerized applications using what are known as Docker images. They are created by building a read-only file that contains a set of instructions for creating a container to run on the Docker platform as shown in Figure 1.1.

Figure 1.1: Docker Containerization [15]
1.1.3 Kubernetes Cluster and Kubernetes CLI

Kubernetes is an open-source system for deploying, scaling, and managing containerized applications. The architecture is illustrated in Figure 1.2. Generally, anything run with Kubernetes operates on a cluster, i.e., a set of node machines that is used for running containerized applications. A Kubernetes cluster contains six main components which can run on Linux or as a Docker container [14].

1. API server: Exposes a REST interface to all Kubernetes resources [14].

2. Scheduler: Schedules containers according to resource requirements [14].

3. Controller manager: Manages controllers such as node controllers, endpoint controllers, and replication controllers [14].

4. Kubelet: Ensures that containers are running in a Pod by interacting with the Docker engine [14].


6. Etcd: Stores all cluster data [14].

The Kubernetes cluster can be accessed locally using a command line interface. This is done using the Kubernetes command-line tool (CLI), kubectl. Kubectl is powerful enough for deploying applications, inspecting and managing cluster resources, and viewing logs.

1.1.4 Rancher and Rancher CLI

Rancher is a cluster management system designed to assist in running and maintaining Kubernetes clusters. Rancher can be accessed locally using a command line interface which
is also called the Rancher CLI. It is worth noting that while Rancher was used to configure the CS 5604 cloud cluster, it is not necessary for the functionality of the cluster, and thus will not be used within the DLRL cluster.

1.2 Project-Specific Information

1.2.1 Computer Science and the Digital Library Research Laboratory

The Virginia Tech Department of Computer Science has as one of its labs the Digital Library Research Laboratory (DLRL). VTCS has a container cluster that supports its education mission, and was used to support work in Fall 2020 in its course CS5604. Research work with the containers developed in that class is best managed on a container cluster focused on research, i.e., in DLRL. Accordingly, this project aimed to move the containers working on the VTCS container cluster to the container cluster in DLRL, to be

Figure 1.2: Kubernetes Architecture [10]
managed there with Kubernetes.

1.2.2 CS 5604 Containerized Applications

The containerized applications we have been tasked with migrating to the DLRL cluster were developed by a group of graduate students taking CS 5604. The applications are divided into sub-projects. Three of these projects are designed to perform different types of natural language processing and analysis (ETD, WP, TWT), one project integrates the functionality of the previous three (INT), and the final project provides a front end for the combined applications (FE). While our focus is mainly on migration, we still must be aware of how these applications are designed to ensure they will function correctly in DLRL.
Chapter 2

Requirements

This project deals with migration from the Kubernetes cluster in the VTCS cloud to DLRL. Our client aims to use the newly created cluster in favor of the original Kubernetes cluster. With that end goal in mind, there are several requirements that must be met by the DLRL cluster to satisfy this need. The following sections are descriptions of these requirements and their details.

2.1 User Goals

2.1.1 Accessibility of Cluster

Any given user must be able to access and successfully run the CS 5604 application containers within DLRL. One type of run involves a user interacting with the front end UI. Another involves a developer running tests on individual containers. Thus, a key goal is to make sure the containers can be run from the front end or from elsewhere in the cluster.

2.1.2 Container Troubleshooting

If any issues arise while using this cluster, a user must be able to access and debug specific containers as needed to identify the problem. Users in this case will most often be
developers for the cluster. The goal for this user is to give them the necessary information
to be able to explore the details of the cluster to find and determine any issues therein.

2.1.3 Usable Documentation

All users need to have access to comprehensive documentation on the DLRL cluster, its
structure, and how it is used. This includes details of any changes made during the
migration from the cloud to DLRL, a user guide, a manual for new developers, and any
other information relevant to fully understanding the project.

2.2 Project Tasks

2.2.1 Transfer Containers to DLRL

The first task to meet these requirements is to transfer the containers to the DLRL cluster.
In regards to our project, transferring does not refer to a direct movement of an application
from one place to another. Instead, we are rebuilding the containers within DLRL in a way
that mirrors the cloud cluster as closely as possible. To do this, we must deploy entirely
new containers inside the DLRL clusters, but in such a way that they have the same
characteristics as in the cloud cluster.

2.2.2 Migrate Dependencies to DLRL

Part of this migration process will also include adding any dependencies to the DLRL
cluster that are needed for the containers to run properly. This includes external storage,
running services that the containers may call on, and environment variables needed to
ensure the cluster runs as expected.

2.2.3 Verify Containers Actively Run

Of course, for a user to interact successfully with the UI, all the containers must be running without error, and thus be able to run their internal applications. Containers that are working as expected will show a running status when checked in the cluster, whereas failing containers will show an error as their status. Those containers must be debugged so that the DLRL cluster is fully functional.

2.2.4 Compile Documentation Into Final Report

All documentation made by our team throughout this project is compiled in this report. The information herein will provide future users the relevant information on the structure and functionality of the cluster for reference. This provides users with reference material of important information about the cluster that will be needed to correctly run and troubleshoot it.

2.3 Relevant Workflows

With the knowledge gained from the previous two sections in this chapter, we gathered enough information to construct workflows that guided our process throughout this project. These workflows are detailed as flow charts in Figures 2.1, 2.2, and 2.3.
Figure 2.1: Workflow to meet the first requirement of the users

Figure 2.2: Workflow to meet the second requirement of the users

Figure 2.3: Workflow to meet the third requirement of the users
Chapter 3

Implementation

3.1 Details of the Cloud Cluster

The VTCS cluster currently holds the containerized applications produced by CS 5604 students. The cloud runs Kubernetes v1.15.5 alongside Rancher to manage several cluster namespaces, each of which has unique deployments of pods within them. Rancher provides a higher level of division on top of Kubernetes namespaces, referred to as Rancher Projects. Table 3.1 provides a comprehensive list of all the projects registered in the cloud cluster, as well as a brief description of their functions and what namespaces they contain.

Table 3.1: Table of Rancher Projects and their namespaces

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Description</th>
<th>Kubernetes Namespaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 5604 ETD 2020</td>
<td>Electronic Theses and Dissertations analysis</td>
<td>cs5604-etd-db, etd</td>
</tr>
<tr>
<td>CS 5604 FE 2020</td>
<td>Front End interface</td>
<td>cs5604-fe-db, fe</td>
</tr>
<tr>
<td>CS 5604 INT 2020</td>
<td>Integration of all projects</td>
<td>cs5604-int-test, cs5604-int-testdb, int</td>
</tr>
<tr>
<td>CS 5604 TWT 2020</td>
<td>Tweet analysis and processing</td>
<td>cs5604-twt-db, twt</td>
</tr>
<tr>
<td>CS 5604 WP 2020</td>
<td>Web page archival</td>
<td>cs5604-wp-db, wp-test, wp</td>
</tr>
</tbody>
</table>
The following namespaces in the VTCS cloud are empty, and will not be included in the DLRL cluster structure:

- fe
- int
- cs5604-int-testdb
- twt
- wp

Within each namespace there are a series of workloads, otherwise known as Kubernetes deployments. These deployments house a certain number of pods, and each pod within a deployment is running a container of the same image. That way, multiple containers can be running the same application at once within a cluster.

Containers are not typically designed to store data for long periods of time. Instead, their goal is to be able to deploy a running application that can be quickly deleted once its main task is complete. However, the CS 5604 applications are mainly built for data processing and analysis, so they must be able to access and store data safely. For this reason, most of the containers within the VTCS cloud have been given persistent volume claims to allow them to mount external NFS storage to them, where they can read data from and write their outputs to as needed.

The majority of the containers within the cluster utilize NFS storage from camelot.cs.vt.edu, a Virginia Tech server. However, some of the containers in the cloud are mounted to a Ceph file system which, at the request of our client, won’t be used within the DLRL cluster.
3.2 Project Objectives

We have divided this project into different objectives in order to monitor our progress. They are listed as follows:

1. Familiarize ourselves with Docker which included the following resources given to us by our client:
   
   (a) **Docker — Beginner’s Guide — Part 1: Images & Containers** [8]
   
   (b) **Learn how to build and share a containerized app** [13]
   
   (c) **Kubernetes 101: Pods, Nodes, Containers, and Clusters** [17]
   
   (d) **Overview of kubectl** [2]

2. Run the images contained in the Virginia Tech Computer Science cloud locally. This was done in the following steps:

   (a) Create a Docker file for each image using the yaml configuration of the images.
   
   (b) Build the Docker file.
   
   (c) Successfully run the Docker file locally.

3. Migrate the Kubernetes cluster to the Digital Library Research Laboratory using the information gathered in step 2.
3.3 Setting up the DLRL Cluster

The DLRL cluster is made up of a series of nodes running Ubuntu 18.04.5 and using Kubernetes v1.20.4. Table 3.2 describes all current nodes within the cluster.

Table 3.2: DLRL Cluster Nodes

<table>
<thead>
<tr>
<th>Node ID</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>master-node</td>
<td>128.173.49.22</td>
</tr>
<tr>
<td>worker01</td>
<td>128.173.49.25</td>
</tr>
<tr>
<td>worker02</td>
<td>128.173.49.24</td>
</tr>
</tbody>
</table>

All of our work will be done from the master-node, as it will delegate which of the worker nodes will be assigned which deployments to run upon creation. These nodes had already been created for our use, with Kubernetes and Docker already installed. However, each node was missing the packages necessary in order to mount NFS storage to the containers, which we resolved by running the following command on each node:

```
sudo apt install nfs-common
```

As our team is working remotely, we had to use VT Pulse Secure VPN to access the master node to begin migration. Instructions on how to install and use the VPN can be found at https://onecampus.vt.edu/task/all/installing-pulse-vpn. This step is only needed if you are accessing the cluster from a computer not registered within Virginia Tech’s network.

Once connected to the VPN, we are able to ssh into the master node using an account set up for our use. Then, our first setup was to recreate all of the namespaces from the VTCS cloud in the DLRL cluster. We used the following command to create each of the nonempty namespaces referenced in Table 3.1:
kubectl create namespace <namespace_name>

We can verify the namespaces were created correctly with the command `kubectl get namespaces`, as shown in Figure 3.1. The namespaces default, kube-node-lease, kube-public, and kube-system are all created automatically with the cluster, and can be disregarded.

![kubectl output](image)

Figure 3.1: View of DLRL Namespaces

### 3.3.1 Persistent Volumes and Claims Creation

Once we set up the basic skeleton of the DLRL cluster, we could begin to deploy the CS 5604 images into their relevant namespaces and mount the necessary NFS storage to them. Since the containers will only run if they have access to their storage, we began by setting up the persistent volumes that will hold the external storage information.

To do this, we created a directory in the master node called `volumes` which holds the yaml files that the persistent volumes are created from. Next, we created a yaml file for each persistent volume that the cluster uses, which we based off of the preexisting yaml files within the VTCS cloud. Figure 3.2 shows an example of a persistent volume yaml file.
In this file, we have defined the name of the persistent volume, the server and path to be mounted, as well as the access mode of this storage system. The access mode in this case indicates that multiple nodes can read and write to the NFS storage at the same time. Finally, we have given a set of mount options that the containers will need to use for this storage.

Persistent volumes act as a general reference to any external storage that the cluster may use. In order for a container to be able to access the storage, it must use a persistent volume claim to mount the storage to itself. Thus, we needed to create persistent volume claims within each of the DLRL namespaces, so that containers deployed within them will have access to these claims. Figure 3.3 shows an example of a persistent volume claim yaml file.

In this case, we reference the related persistent volume so any container that uses this claim can mount that storage to itself, as well as the namespace this claim is attached to. Once again, we utilized the yaml files that were already within the VTCS cloud as a
reference point for these claims.

Once a persistent volume or persistent volume claim yaml file is written, you can run the following command to create it within the cluster:

```bash
kubectl apply -f filename.yaml
```

You can check if these volumes and claims were created successfully using the commands

```bash
kubectl get pv
```

```bash
kubectl get pvc --namespace <namespace_name>
```

respectively.
### 3.3.2 Image Deployment into DLRL

Once our external storage has been set up in the cluster in the form of persistent volumes and claims, we can begin deploying the CS 5604 application images into their respective namespaces in DLRL.

Utilizing the yaml files from the workloads listed in the VTCS cloud, we created yaml files for each of the deployments within a directory called *namespaces*. Figure 3.4 shows what a yaml file of this type looks like.

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "3"
  generation: 3
  labels:
    workload.user.cattle.io/workloadselector: deployment-cs5604-wp-db-archive-webpage
    name: archive-webpage
    namespace: cs5604-wp-db
  resourceVersion: "7278440038"
  selfLink: /apis/apps/v1/namespaces/cs5604-wp-db/deployments/archive-webpage
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      workload.user.cattle.io/workloadselector: deployment-cs5604-wp-db-archive-webpage

strategy:
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
  type: RollingUpdate

template:
  metadata:
    labels:
      workload.user.cattle.io/workloadselector: deployment-cs5604-wp-db-archive-webpage
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        name: web-cs5604-wp-archive-webpage
        command: "["/bin/sh"]"
        imagePullPolicy: Always
        name: archive-webpage
        resources:
          limits: {}
        securityContext:
          allowPrivilegeEscalation: false
          privileged: false
          readOnlyRootFilesystem: false
          runAsUser: 1000
          runAsNonOwnerUser: false
        stdin: true
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      - name: vol
        imagePullSecrets:
          - name: vol
        volumes:
          - name: vol
            persistentVolumeClaim:
              claimName: cs5604
```

Figure 3.4: Example yaml file for a single image deployment
These yaml files contain all of the information necessary for one or multiple containers to be deployed running the same image. As displayed in the figure, the image is pulled from a private container repository within container.cs.vt.edu. Since this is a private repository, Kubernetes needs to authenticate with the registry in order to access the image, and thus an image pull secret is used, created from one of our teammate’s credentials to the registry. The other information within the yaml file describes the volumes that the container needs to mount, as well as any environment variables and other configuration settings that are necessary for the container to run.

As we have for the persistent volumes, we can use the `kubectl apply` command to deploy the containers from the yaml files. We can see deployments that have successfully launched by the `kubectl get deployments --namespace <namespace_name>` command, as shown in Figure 3.5.

### 3.3.3 Container Services

In some cases for the cluster, some containers will need to be made accessible to other containers. For example, within the ETD portion of the cluster there is a container running Postgres (i.e., PostgreSQL), which the other containers within ETD are meant to send data to for storage and indexing purposes. However, we must notify Kubernetes that the Postgres container must be discoverable, so that Kubernetes will create an internal IP
address for that the other containers will be able to use to connect to Postgres. The way we do this is by setting up Kubernetes services on the necessary containers.

Just like persistent volumes, claims and deployments, we can create services through a yaml file on the master node. Figure 3.6 shows an example of a service yaml file.

In this yaml file, the service is given a name and a namespace to reside in, and then the owner references section defines what specific container attaches to the service. The uid is the identifier of the container, which you can discover using the command

```
kubectl get pods -n <namespace> <pod-name> -o jsonpath='{.metadata.uid}'
```

Within the specification section, we define a port that any client container will be able to attach to, and give it a name and an optional protocol for the port. Then, we apply the service using the `kubectl apply` command described in previous sections to create the service.

Since the service we created essentially exposes a port on the chosen container, we have one
extra step we need to take. DLRL is a multi-node cluster, so it may be that some containers deployed on one node will want to communicate with a container deployed on a completely different node. They will only be able to do that if they can connect to the container port through an exposed port on the node. Therefore, we need to create a service that reserves a port on the node where our container resides. Figure 3.7 demonstrates the yaml file for a such a service.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    #field.cattle.io/targetworkloads: '['deployment:cs5604-etd-db:postgres-test']'
    workload.cattle.io/targetworkload: "true"
  creationTimestamp: "2020-11-11T21:11:04Z"
labels:
  deployment: postgres-test
spec:
  ports:
    - nodePort: 31487
      port: 5432
      protocol: TCP
      targetPort: 5432
  selector:
    workload.user.cattle.io/workloadselector: deployment-cs5604-etd-db-postgres-test
    sessionAffinity: None
    type: NodePort
  externalTrafficPolicy: Cluster

Figure 3.7: Example yaml file for a node port service
```

The major difference to note is that the ports section under the specifications defines the port of the node as well as the port of the container that we defined in the previous yaml file. This effectively links the node port with the container port, giving any containers outside the current node a way to access this Postgres container.

We have created services using this procedure for each container in DLRL that needs to be accessible to other containers. Section 5.1.3 provides a comprehensive list of all services we have created for DLRL.
Chapter 4

User Guide

4.1 Accessing the Front End

At this time, several bugs have been identified within the back end of the CS 5604 applications. These bugs are recorded in Section 5.2. Until these issues are resolved, several functions of the applications may not work as intended.

However, a typical user will want to know how to access the front end so they may use the applications. The current home page link is http://10.244.2.123:3000, which is the exposed API endpoint for the front end. This URL is subject to change if the front end undergoes modifications. Upon accessing this URL, new users may create an account within the system. As this front end deployment is new, any previously registered users should be aware that they will likely need to create a new account within the system, since registration information was not transferred from the VTCS cloud applications.

From the home page and upon successful login, the user will be able to access the various back end applications through the user interface. A comprehensive tutorial on the front end interface and how to navigate it can be found within the user manual section of the CS 5604 Front End team’s final report [7].
4.2 General Usage

For users interested in the CS 5604 system, Figure 4.1 provides a diagram of that system on the DLRL cluster, and how each piece interacts with each other. To sum up, the cluster supports three back end pieces, referred to as TWT [5], WP [6] and ETD [9], which perform the majority of the data processing. These pieces pull initial data from a VT Camelot server, where they have been provided NFS mounts to persistent volumes. Their work is synchronized and supported by the integration system, INT [11], which connects the back end to an ElasticSearch server running on containers on another VM (elasticsearch.cs.vt.edu) for data storage and indexing. Finally, the front end, FE [7], provides the user interface wherein a person can pull up the data processed by the back end.
Figure 4.1: Structure of the DLRL Cluster
Each of these projects that represents a portion of the overall functionality of the cluster is made up of a series of containers within the cluster. These containers are grouped into Kubernetes namespaces for organizational purposes, and the containers are currently housed within one of two worker nodes as detailed in Table 3.2.

Since our main focus for the project was on setting up the DLRL cluster, the following section provides more insight into using Kubernetes and Rancher to set up containers within a cluster environment.

4.3 VT Cloud and General Containers Guide

While the new cluster does not reside in the VTCS cloud, the cluster it is modelled off of was hosted there. The VTCS cloud utilized Rancher to provide a user interface and a command line interface to access the cluster from the web or from a terminal. Rancher is not currently set up on DLRL, but if future developers add it for ease of use, this section can act as a guide on how to access a cluster remotely via Rancher CLI.

4.3.1 Access Cluster Remotely With Rancher

1. Step 1: Create API Key in Rancher UI.

   On the Rancher website, click on your profile and select API and Keys from the drop down menu.

2. Step 2: Select Add Key
3. Step 3: Add a description for the key if desired, then click Create. Do not give the key a scope of any kind – that will prevent the key from being used for authentication with Rancher.

4. Step 4: Once the key is created, save all the information it gives you into a Notepad or Word document. You’ll need it later, and it won’t give you the information ever again.
4.3.2 Rancher CLI

1. Download the Rancher CLI to run some commands from your computer. Click the Download CLI button in the bottom right of the cloud webpage and unzip the downloaded folder onto your computer.

2. Open up a terminal and type in the following command: `rancher login https://cloud.cs.vt.edu/k8s/clusters/c-l4mcr --token <BEARER_TOKEN>`. Replace the `<BEARER_TOKEN>` with whatever your API key gave you, which you saved in step 3 of Section 4.3.1.

3. Now you should be able to manipulate the cluster remotely from your machine using Rancher CLI commands - the command `rancher kubectl` will let you run kubectl commands from Rancher.
4.3.3 Using Dockerfiles to Create Containers

Our team mostly utilized yaml files to deploy containers within DLRL, but you can also use the Docker CLI and Dockerfiles to deploy containers within a cluster. The following is a short guide on how to set up a simple container using a Dockerfile.

1. Get the URL of the application image from the container registry it comes from.
2. Create a folder for the image to sit in, and create a new file called “Dockerfile” without an extension in the same folder. The Dockerfile should have at minimum the following contents, although more details should be added for environment variables, container commands, and other information to customize the container to your needs:

```dockerfile
FROM <image URL>
WORKDIR <working directory path>
COPY . .
```

Figure 4.8: Contents of a Dockerfile.

3. Open up a terminal, navigate to the folder containing the Dockerfile and enter the following command:

```
docker build -t <image_name> .
```

4. Wait for the image’s layers to download. This might take a while. Once complete, use the following command to deploy the image:

```
docker run -dp <your port>:<container port> <image_name>
```

4.3.4 Working with Kubernetes CLI

The Kubernetes CLI is designed to specifically work with Kubernetes clusters, such as the cluster in DLRL. This section provides an overview of how to work with Kubernetes to deploy, delete, and troubleshoot containers.

1. Creating and Deploying
(a) Create YAML files for each Deployment in each namespace. Information on yaml file creation can be found in Section 3.3.2.

(b) Run the following command to deploy the container

```
kubectl apply -f <name of yaml file>
```

(c) Check container(s) are running using the following command,

```
kubectl get pods -n <namespace of container>
```

2. Deleting and troubleshooting

(a) Delete a container deployment with the following command:

```
kubectl delete deployment <name of deployment>
```

(b) To get container error logs and detailed deployment information for a container, use the following commands

```
kubectl describe pod classify-text-0 -n cs5604-wp-db
```
```
kubectl logs classify-text-0 -n cs5604-wp-db
```

3. Some images are kept within private registries, and therefore need authentication to download them. To authenticate, we need to create a secret with the credentials to the registry, and then add that secret as an imagePullSecret to the deployment’s yaml file. Use the following command to create an imagePullSecret:

```
kubectl create secret docker-registry <secret-name>
```
```
--docker-server=<server-url> --docker-username=<username>
```
```
--docker-password=<password> --docker-email=<email>
```
Chapter 5

Developer Manual

5.1 Cluster Component Guide

5.1.1 Containers

For information on the CS5604 namespaces and their functionality within the CS container cluster, see Table 3.1. Describing this CS4624 project, Tables 5.1 to 5.7 provide a comprehensive list of all of the currently deployed containers (referred to as deployments in Kubernetes nomenclature) within the DLRL cluster, divided out by the namespace they are in. These tables contain the given names of the deployments, which are taken directly from the VTCS cloud deployments, the number of replicas, or instances, of the given deployment, and its current status. The caption associated with the table lists the namespace that the given deployments reside in. CLBO means CrashLoopBackOff, an error within Kubernetes containers that we discuss more in Section 5.2.

<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>frontend0</td>
<td>1</td>
<td>Running</td>
</tr>
</tbody>
</table>

Table 5.1: Deployments within cs5604-fe-db
<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>crop-image-test</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>mysql</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>postgres-test</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>test-chapter-extract</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>test-classification</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>test-image-inference</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>test-ingestion</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>test-pdf-to-image</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>test-text-extraction</td>
<td>1</td>
<td>CLBO</td>
</tr>
</tbody>
</table>

Table 5.2: Deployments within cs5604-etd-db

<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>etd-chapter-extraction</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>etd-classification</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>etd-crop-image</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>etd-image-inference</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>etd-metadata-ingestion</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>etd-pdf-to-image</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>etd-test-extraction</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>etd-validate</td>
<td>1</td>
<td>CLBO</td>
</tr>
</tbody>
</table>

Table 5.3: Deployments within etd

5.1.2 Persistent Volumes and Claims

Persistent volumes are a component of Kubernetes that allow containers to pull in and output data for long term storage. Currently each namespace has two persistent volumes assigned to it and two matching persistent volume claims. These persistent volumes both connect to NFS storage in camelot.cs.vt.edu, at different directory mount points. The claims are referenced within any containers that require a mount point to pull data from. For the purposes of minimizing redundancy, we will not explicitly describe the persistent volumes and their claims here, since they are repeated once per namespace. For details on how to configure these components and how they are mounted to containers, see Section 3.3.1.
<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>airflow</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>api</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>reasoner</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>services-db</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>ubuntu</td>
<td>1</td>
<td>Running</td>
</tr>
</tbody>
</table>

Table 5.4: Deployments within cs5604-int-test

<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-parse</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>els-index</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>filter-merge</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>geolocation</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>hashtags</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>id</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>keywords</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>mentions</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>timestamp</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>twirole</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>username</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>warc-json</td>
<td>1</td>
<td>CLBO</td>
</tr>
</tbody>
</table>

Table 5.5: Deployments within cs5604-twt-db

5.1.3 Services

Most of the services created for the cluster are ClusterIPs, which expose a port within the container, and NodePorts, which associate those ClusterIPs with ports on the node the container is deployed on. Two of these services have been modified in DLRL: expose-fe and expose-api. In the cloud cluster, these services were LoadBalancers, which asynchronously creates a load balancer for the service using the host cloud’s load balancing system. [4] However, since the new cluster is not hosted in a cloud environment, no load balancer will be created, making the service effectively the same as a NodePort. We changed the type of these services to be reflective of this. Table 5.8 contains a comprehensive list of every service, the namespace it is housed in, its type, and the ports assigned to it.
<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>archive-webpage</td>
<td>2</td>
<td>CLBO</td>
</tr>
<tr>
<td>classify-text-0</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>extract-url</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>index-data</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>summarize-text-0</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>testnfs</td>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>testnfs2</td>
<td>1</td>
<td>Running</td>
</tr>
</tbody>
</table>

Table 5.6: Deployments within cs5604-wp-db

<table>
<thead>
<tr>
<th>Deployment Name</th>
<th>Replicas</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>classify-text</td>
<td>1</td>
<td>CLBO</td>
</tr>
<tr>
<td>summarize-text</td>
<td>1</td>
<td>CLBO</td>
</tr>
</tbody>
</table>

Table 5.7: Deployments within wp-test

### 5.2 Current Issues

The CS5604 application in DLRL is not fully functional as of the completion of the migration. Many of the containers, as depicted in the deployment Tables 5.1 to 5.7 are showing CrashLoopBackOff errors. This particular error, given the name CrashLoopBackOff since it occurs when a container must be perpetually restarted by the cluster, can occur for a variety of reasons. It could range from an issue with the base image’s source code down to a simple configuration error. Since there are so many reasons a container may enter a CLBO state, we have compiled a list of all the container deployments currently in this state on DLRL, as well as their most recently traced error log, which we found by running the command

```
kubectl logs <pod name> -n <namespace of pod>
```

Table 5.9 contains the full list of all current deployment errors. Many of the containers producing errors are noted as performing a “Restart upon completion.” We have noted this when the container is not producing a specific error, but instead appears to be functioning
<table>
<thead>
<tr>
<th>Namespace</th>
<th>Service Name</th>
<th>Type</th>
<th>Port(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs5604-etd-db</td>
<td>mysql-nodeport</td>
<td>NodePort</td>
<td>3306:32742/TCP</td>
</tr>
<tr>
<td></td>
<td>postgres-test</td>
<td>ClusterIP</td>
<td>5432/TCP</td>
</tr>
<tr>
<td></td>
<td>postgres-test-nodeport</td>
<td>NodePort</td>
<td>5432:31487/TCP</td>
</tr>
<tr>
<td>cs5604-fe-db</td>
<td>frontend0</td>
<td>ClusterIP</td>
<td>3000/TCP</td>
</tr>
<tr>
<td></td>
<td>frontend0-nodeport</td>
<td>NodePort</td>
<td>3000:32577/TCP</td>
</tr>
<tr>
<td></td>
<td>expose-fe</td>
<td>NodePort</td>
<td>3000:30705/TCP</td>
</tr>
<tr>
<td>cs5604-int-test</td>
<td>airflow</td>
<td>ClusterIP</td>
<td>5000/TCP</td>
</tr>
<tr>
<td></td>
<td>airflow-nodeport</td>
<td>NodePort</td>
<td>5000:30335/TCP</td>
</tr>
<tr>
<td></td>
<td>expose-api</td>
<td>NodePort</td>
<td>5000:32640/TCP</td>
</tr>
<tr>
<td></td>
<td>api</td>
<td>ClusterIP</td>
<td>5000/TCP</td>
</tr>
<tr>
<td></td>
<td>api-nodeport</td>
<td>NodePort</td>
<td>5000:31675/TCP</td>
</tr>
<tr>
<td></td>
<td>reasoner</td>
<td>ClusterIP</td>
<td>5000/TCP</td>
</tr>
<tr>
<td></td>
<td>reasoner-nodeport</td>
<td>NodePort</td>
<td>5000:32307/TCP</td>
</tr>
<tr>
<td></td>
<td>services-db</td>
<td>ClusterIP</td>
<td>5432/TCP</td>
</tr>
<tr>
<td></td>
<td>services-db-nodeport</td>
<td>NodePort</td>
<td>5432:30129/TCP</td>
</tr>
</tbody>
</table>

Table 5.8: All services created for DLRL

normally until the application finishes running. When it finishes, the container is not
resetting itself or exiting gracefully. It is instead exiting in a way that Kubernetes assumes
is an error, since containers are not meant to exit without being explicitly instructed to do
so.

While this is an issue particularly prevalent in the TWT namespace, we do not currently
have a solution for it. These CLBO errors existed within the VTCS cloud before the
cluster was migrated, and thus they have appeared again in DLRL. However, it would
require some reconfiguration of the inside of the containers to prevent them from exiting
inappropriately, which is a fix that we do not have the time to implement.
5.3 Troubleshooting Tips

As we have progressed through this migration process, we have accumulated a set of tips that may be helpful for a future developer on the cluster to keep in mind. These are as follows:

1. Refer to other yaml files for help in container configuration. All yaml files used to create the currently deployed containers are located in the master node, organized into folders based on their type and namespace. Using their setup as a baseline for new components will help ensure that Kubernetes won’t throw a formatting error or apply the yaml file incorrectly.

2. If you are deploying new images, be sure the imagePullSecret is valid. If the image is in a private container registry, set up a secret using the instructions in Section 4.3.4 and use the created secret in the yaml file. The secrets currently in place will eventually become invalid, as they were created with a teammate’s credentials, which will expire approximately June 2021. Since the credentials used to grant access to the container registry are created from the login information of a user that has access to the required images, these credentials will remain valid until the user is no longer able to access the registry or the user changes their password. In either of these cases, new credentials must be created for authentication.

3. The kubectl commands referenced throughout the report were used many times as we were setting up the containers. In particular, the commands `kubectl describe deployment <deployment name> -n <namespace of deployment>` and `kubectl logs <pod name> -n <namespace of pod>` provide a lot of useful information to diagnose issues when the container is being launched and when it isn’t running correctly.
4. If a cluster component isn’t working as anticipated, you can delete it with `kubectl
delete <deployment/service/pv/pvc> -n <namespace of component>`. The
yaml file used to create the component will still exist, and can be adjusted as needed.
<table>
<thead>
<tr>
<th>Namespace</th>
<th>Deployment Name</th>
<th>Logged errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs5604-etd-db</td>
<td>test-chapter-extract</td>
<td>No module named fitz</td>
</tr>
<tr>
<td></td>
<td>test-ingestion</td>
<td>Connection to elasticsearch.cs.vt.edu timed out</td>
</tr>
<tr>
<td></td>
<td>test-text-extraction</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td>etd</td>
<td>etd-chapter-extraction</td>
<td>No module named fitz</td>
</tr>
<tr>
<td></td>
<td>etd-metadata-ingestion</td>
<td>Connection to elasticsearch.cs.vt.edu timed out</td>
</tr>
<tr>
<td></td>
<td>etd-text-extraction</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>etd-validate</td>
<td>SQL: Entry already exists in table</td>
</tr>
<tr>
<td>cs5604-wp-db</td>
<td>archive-webpage</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>classify-text</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>extract-webpage</td>
<td>ModuleNotFoundError: No module named 'rouge'</td>
</tr>
<tr>
<td></td>
<td>summarize-text</td>
<td>UnboundLocalError: local variable 'scores' referenced before assignment</td>
</tr>
<tr>
<td>wp-test</td>
<td>classify-text</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>summarize-text</td>
<td>UnboundLocalError: local variable 'scores' referenced before assignment</td>
</tr>
<tr>
<td>cs5604-twt-db</td>
<td>data-parse</td>
<td>Data Parser of RAW Tweet Logs. Usage: ./data_parser.py &lt;FILENAME&gt; &lt;hashtag,username&gt; &lt;QUERY&gt;</td>
</tr>
<tr>
<td></td>
<td>els-index</td>
<td>Connection to elasticsearch.cs.vt.edu timed out</td>
</tr>
<tr>
<td></td>
<td>filter-merge</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>geolocation</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>hashtags</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>AssertionError</td>
</tr>
<tr>
<td></td>
<td>keywords</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>mentions</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>timestamp</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>twirole</td>
<td>ValueError: Expected object or value</td>
</tr>
<tr>
<td></td>
<td>username</td>
<td>Resets upon completion</td>
</tr>
<tr>
<td></td>
<td>warc-json</td>
<td>Resets upon completion</td>
</tr>
</tbody>
</table>

Table 5.9: Current DLRL Deployment Errors
Chapter 6

Summary

6.1 Major Changes from VTCS to DLRL

The transition from the VTCS cloud to DLRL has kept most of the setup intact. The existing namespaces and deployments follow the same naming schemes, as do the services and persistent volumes. All containers that are functional in the VTCS cloud work in DLRL, and the configurations of each are as close as possible to their cloud counterparts. There were, however, a few key changes that were made out of necessity or lack of usefulness, detailed as below:

6.1.1 Removal of the Ceph Filesystem

The Ceph file system has been fully removed. Several containers in the VTCS cloud were mounted to a Ceph filesystem that fed them input data. However, at the request of our client, we have removed the Ceph filesystem entirely. The cloud containers were already reconfigured to pull data from the Camelot server by the time we began this project, so all we had to do was remove the Ceph persistent volumes and useless mounts on the relevant containers.
6.1.2 Exclusion of CentOS Containers

Continuing from the previous section, most namespaces within the VTCS cloud had at least one deployed container of the latest version of CentOS. The only customization these containers had were a mount to the now unused Ceph filesystem. Without that mount, the CentOS containers had no use in the cluster, as no other containers spoke to them and they were not performing any tasks. Therefore, they were not migrated to DLRL.

6.1.3 Exclusion of Unused Services

Almost every container in the VTCS cloud had an associated ClusterIP. However, none of them except for those listed in Table 5.8 had any active NodePorts attached to them, or were being accessed by other containers. Since they were extra services with no apparent use, they were not migrated to DLRL.

6.1.4 Reduction of Cluster Nodes

While the VTCS cloud had a total of nine nodes for all of the containers to be deployed across, the current cluster only has three, as described in Table 3.2. However, there have not been any noticeable performance issues with this smaller cluster, nor have there been any cluster crashes since all of the containers have been deployed. However, if performance issues do arise as a result of future development, new nodes can be added to the existing cluster without issue.
6.2 Lessons Learned

Throughout this project, we had to quickly develop an entirely new set of skills in order to work with a technology that nobody in our team was familiar with at the beginning of the semester. Since we had much to learn, the following section is a brief overview of the major lessons we learned while working on this project.

1. Containerized applications are much different from their non-containerized counterparts. There is much careful planning that needs to be done to properly configure a containerized application, and any problems will almost certainly lead to a CLBO error in the cluster. However, the payoff for the additional setup is that a containerized application is fully platform-independent, so it can be run from any machine that supports virtualization.

2. Failure to carefully document leads to much backtracking. When doing something as procedural as migration, it is vital to be taking consistent notes on what has been moved and its current status. Otherwise, time will consistently be lost by having to manually review previously done work to ensure everything has actually been completed.

3. Prepare for major setbacks. Throughout this semester, we were consistently put behind our planned schedule by various problems that prevented our progress. These issues were often access-related, where we had to wait for a response from an administrator to grant us the privileges we needed to access the images or ssh into the DLRL worker nodes. Administrators are often quite busy, and so responses can be delayed for some time. Therefore, having plans for additional work that can be done while you wait can keep progress moving forward during a stalling moment.
4. Communication is vital in computer science. Whether it be with your assigned team, a mentor, an administrator, or other related group, you are guaranteed to interact with many different people when working on any given project. Having the skills to communicate professionally and concisely are extremely important in these instances, and learning from communication issues even more so.

6.3 Future Work

At this time, the DLRL cluster is not fully functional. The obvious concern is to address the various containers that are currently in a CLBO state, and getting them running. After that is addressed, however, future work could include scaling the cluster to accommodate more users of the front end. Developers may also consider thoroughly testing the back end applications with newer and larger sets of data, and possibly even adding a new back end application that can process a new kind of written work, such as other VT documents.
References


42


