University of Victoria
Department of Physics & Astronomy

Containerization of Cloudscheduler VM Provisioning Service

In partial fulfillment

of the requirements of the Physics & Astronomy Co-op Program

Fall 2019

Work term #2

By: Matthew Ens

Performed at:

High Energy Physics Research Computing Group (HEPRC)

Department of Physics, University of Victoria

Supervisor(s): Rolf Seuster (HEPNET/Canada) &

Randy Sobie (IPP Research Scientist)

Job title: Software Developer

Confidentiality notice (Please select all that apply)

Students - Please confirm the confidentiality status of the report with your employer!

☐ This report **is** confidential. Do not share for any purpose other than evaluation and record keeping.

☒ This report is **not** confidential

☒ This report **may** be shared with other co-op students

☐ This report **may not** be shared with other co-op students

Student Signature: _____

Contents

List of Figures

Abstract

Using a Kubernetes pod running four Docker containers, a functioning cloudscheduler virtual machine (VM) provisioning service was built. A MariaDB database container was built with the ability to create the cloudscheduler database from a backup along with an HTCondor container which scheduled jobs and ran the agent which retired virtual machines (VMs). A third container contained InfluxDB and stored the timeseries data for cloudscheduler. The final container had the rest of the cloudscheduler subprocesses. With this configuration cloudscheduler ran normally. It booted VMs, scheduled and ran jobs, and retired VMs. Some work is still necessary to improve the code, including allowing the user to use their own data easily.

1 Introduction

1.1 High Energy Physics Research Computing Group (HEPRC)

The University of Victoria's high energy physics research computing group (HEPRC) [1] is involved in a number of projects in the areas of computing, storage and networking for the ATLAS experiment at the CERN Laboratory in Switzerland and the Belle II experiment at the KEK Laboratory in Japan. They also manage HEPNET/Canada, and are responsible for networking and connectivity both internationally and nationally for the Canadian particle physics community.

1.2 Project Details

The HEPRC group develops and maintains a cloud computing web-application, known as cloudscheduler [2]. Cloudscheduler is a virtual machine provisioning service: it receives batch jobs and starts virtual machines (VMs) from one of its available clouds. Once a VM has started and registered with HTCondor, cloudscheduler sends the job to the VM where it runs and the results are sent back to the submitter. If the VM has sat idle for some configurable time, cloudscheduler retires the VM. This is done with cloudscheduler running on production, see Figure 1.

Development is ongoing and the group continues to add features and fix new issues that arise. At the moment, the cloudscheduler application is run off of VMs with development happening on separate instances of cloudscheduler. Currently, when a new instance of cloudscheduler is desired, cloudscheduler and its dependencies are installed using Ansible, an IT automation language [3] which creates the desired environment and sets up cloudscheduler using the cloudscheduler git repository. This makes cloudscheduler somewhat easy to install but the application of containers for cloudscheduler could increase portability along with making development as well as integration into a continuous integration (CI) system easier.

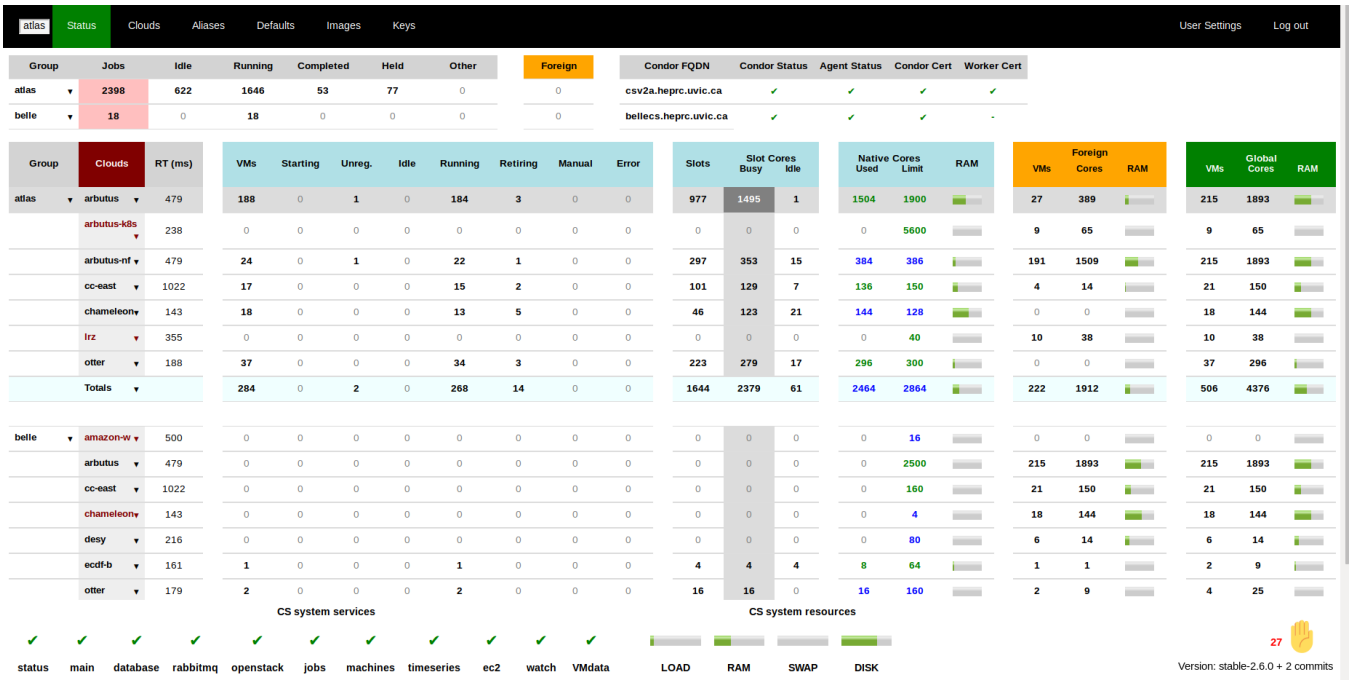

Figure 1: Status page for cloudscheduler running production jobs. The number of jobs along with their status can be seen along with the list of available clouds and how many jobs each cloud is running. The status of the various cloudscheduler services can be seen on the bottom along with the status for HTCondor in the top right

Containers are a method of virtualization roughly similar to a VM. Things such as processes, files, memory, etc. are isolated from the host for both containers and VMs but containers run off of the host kernel instead of booting a full operating system which makes the container setup much more lightweight,[4] see Figure 2. For the containerization of cloudscheduler, we chose to use Kubernetes[5] running Docker[6] containers. Kubernetes runs containers out of "pods", a group of one or more containers. It is allocated it's own private IP address and all containers running in the pod also share a local network. This was ideal as cloudscheduler contains many different parts which should be run in separate containers, to stay as close as possible to the "one process per container" rule and Kubernetes allows individual containers to easily communicate with each other over the pod's private network. Kubernetes also provides services which interact with pods and facilitate networking between pods and between a pod and the outside. Kubernetes has a command-line interface (CLI) which allows a user to interact with and start pods and networking services as pods must be run on a dedicated Kubernetes cluster. Kubernetes also comes with an API for configuring pods and services. An example YAML file for creating a pod with a container and a port forwarding service for this pod can be seen in Appendix I.

Docker was chosen as the container runtime for cloudscheduler as it has a large community around it and is the current industry standard for containers. Docker comes its own CLI which allows for easy building and running of

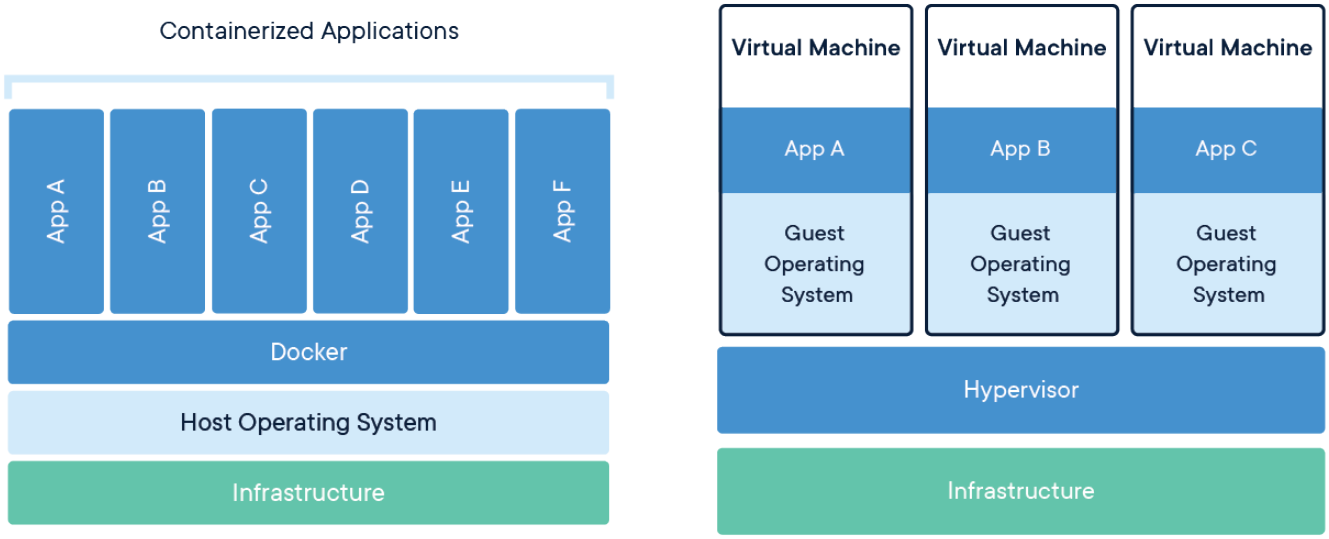

Figure 2: A visual comparison of Virtual Machines (VMs) and Containers. A VM run full guest operating systems on top of a hypervisor which manages them whereas Containers run off of the host's operating system and use a container runtime (Docker) to manage the containers.

containers and is also compatible with Kubernetes. Docker images can be pushed and pulled from repositories on the Docker cloud and pulled images can the basis for more specific applications. A new container image can be built from a Dockerfile or using other packages for building containers. For this project containers were built with Dockerfiles or using the ansible-bender[7] container building package.

The Dockerfile allows you specify a base image and install dependencies for the process(es) you want to run. Necessary files can be copied onto the container image and packages can be installed directly onto the image. This allowed me to create containers which were ready to go immediately on startup instead of having to install things after the container had already been built. A sample Dockerfile can be seen in Appendix II.

Ansible-bender was the second method used in this project for creating container images. Ansible-bender uses playbooks, normally used to configure and maintain running machines, to build the container images. The main reason for using ansible-bender was the HEPRC group used ansible-playbooks to build cloudscheduler on a VM, so with some modification most of the playbook could be reused to build the cloudscheduler container image. Ansible-bender also makes certain operations, such as adding a git repository to a container image, easier compared to Dockerfiles. The main disadvantage with ansible-bender is that it is still in beta development and the cache mechanism which remembers previous builds currently does not work. This made ansible-bender much slower than Dockerfiles for building images. Ansible-bender was used to build the image for the cloudscheduler and MariaDB containers. A sample YAML file defining a playbook can be seen in Appendix III.

One of the goals of the project was to have the software working out of containers without any extra configuration

on startup. Some of the configuration for the containers depended on where the container was running, which could not be known until the containers booted. Container entrypoints address this issue. A container entrypoint is a command or script that can run as the main process of a container or can be used in conjunction with the main process of a container as an initialization script for the container. The containers built for this project made use of the latter, running initialization prior to the container's main process. Referring to the Dockerfile in Appendix II, the "docker-entrypoint.sh" script will run and do extra setup and configuration prior to the "mysqld" command, the main process for the container.

2 Present State

Currently, I have divided cloudscheduler into four separate containers and have these containers running in a Kubernetes pod with a port forwarding service to allow certain connections from the outside. There is a dedicated container for the cloudscheduler MariaDB[8] database, a dedicated container for HTCondor[9], cloudscheduler's workload management system, a third container for the storage of cloudscheduler's time series data using InfluxDB[10], and a fourth container for rest of cloudscheduler's software. Using a test cloud, jobs submitted to this containerized cloudscheduler will have a VM booted for them, be run and the VM will be retired if no jobs remain.

2.1 MariaDB Container

The cloudscheduler database uses the MariaDB relational database which uses an SQL data access interface. It contains almost all of the information used by cloudscheduler including available clouds and virtual machines, data on users and groups, and configuration data. Cloudscheduler's subprocesses continuously access and modify information from MariaDB.

2.1.1 Networking Challenges

The first main challenge with separating MariaDB from the rest of cloudscheduler involved connections between the database and cloudscheduler. MariaDB has two ways of receiving connections. A socket is used for connections within the same machine and a network connection is used for connections from elsewhere. Since cloudscheduler and MariaDB had always been run on the same machine, connections to MariaDB from cloudscheduler were made via the socket. This needed to be changed as MariaDB was now running in a separate container. The -h (or --host=) flag was added to the appropriate places where cloudscheduler subprocesses contacted the database. Since the containers run in the same Kubernetes pod which shares a localhost network, the localhost ip address (127.0.0.1) was sufficient

for these database connections and this was the address placed after the flag. The cloudscheduler configuration already had a place for the host address flag, however some pieces of code needed to be changed to add this option when database connections were made.

2.1.2 Building the Container

Since one of the goals for containerizing cloudscheduler was having cloudscheduler run when the containers are created with minimal setup after container creation, I wanted a working database populated with local data from a previous backup on container creation. This posed a challenge as the MariaDB image provided on the Docker cloud only builds a blank database with no tables or definitions. A script, db_upgrade, had already been written by the HEPRC group to define a new cloudscheduler database but this script only either built a new blank database from scratch or backed up and upgraded an existing one. This script was modified to allow a previous local backup to be written into a blank database and copied along with a previous backup of data and the other necessary files into the container image. An initialization script was written to run db_upgrade when the container was run. The final change that needed to be made was to the cloudscheduler database user. MariaDB connections are only allowed from certain hosts, defined via configuration. A wildcard option does exist for the host category, but it only includes connections over the network and not through the socket. The db_upgrade script accessed the database using the socket so the wildcard option assigned to the database user was changed to "localhost" in the initialization script so that db_upgrade could run using the socket since it was in the same container as the database. A second database user was also created during the initialization to allow connections over the network, the host was set to 127.0.0.1 as all cloudscheduler connections would be coming from this IP address. The MariaDB container required some environment variables to be set during the container startup as well, these were specified in the yaml file for creating the pod. The user also has the option of specifying a git branch different from the one built into the container image, this allows developers to quickly and easily test other versions of the database if changes have been made, although if major schema changes are made or if the user wishes to use a different database schema they still must create a new database image.

MariaDB also required two configuration files which contained passwords and other sensitive information. Sensitive information should not be part of the container image, since anyone who downloads the image would have access to this information. Instead a shared volume was created between the Kubernetes cluster and the pod to allow these configuration files to be accessed by the MariaDB container without loading them into the container image itself.

2.2 HTCondor Container

HTCondor is the job scheduler for cloudscheduler. When cloudscheduler boots a VM, it is added to HTCondor's pool of available computer resources and HTCondor will send the job to the VM to run it. HTCondor needs to connect to both cloudscheduler and the outside clouds where the VMs are booted so some networking challenges needed to be overcome.

When a pod is created in Kubernetes, it is assigned a private IP address and all containers in the pod share a localhost network. The pod is not given a public IP address and any networking the pod does with the outside is one way, things outside the pod cannot contact the pod unless extra steps are taken. The Kubernetes service resource handles networking connections for pods. The service type "NodePort" was used to forward connections from the outside into the pod. The NodePort service opens up a port in the range 30000-32767, specified in configuration, and forwards this port to a desired port in the pod. Attempts to connect to the pod must be directed at the public IP address of the Kubernetes node at the specified port in order for a connection to be successful. For HTCondor, configuration changes needed to be made in order for VMs to be assigned to the HTCondor pool and for jobs to be sent from the master HTCondor running in the container to the HTCondor running inside of the worker VM.

2.2.1 HTCondor Master Configuration

Since a different IP address was being used to forward connections from the outside the TCP_FORWARDING_HOST configuration variable needed to be set to this public IP address of the Kubernetes node. When this variable is used, the port that is being forwarded must be the same as the port being advertised by the forwarding host. Since Kubernetes only allows ports between 30000 and 32767 to be opened, the default HTCondor port needed to be changed from 9618 to a port within this range. This was done by specifying the COLLECTOR_PORT configuration variable to the value set in the configuration of the NodePort service. Since I needed the pod to run on any Kubernetes node with any IP address and any port within range specified, I wrote an initialization script for the HTCondor container which finds the public IP address of the node it's on and updates the configuration accordingly. I also required that the COLLECTOR_PORT be specified via an environment variable so that a user could choose a different port to be forwarded.

2.2.2 HTCondor Worker Configuration

Each VM that gets booted by cloudscheduler has its own HTCondor running, this allows the VM to connect to the master HTcondor and get jobs from it. When cloudscheduler boots a new VM, it sends the necessary configuration data to the VM via the defaults.yaml.j2 template file. The new COLLECTOR_PORT configuration variable was

specified in this file so the VM would know where to connect to. Since the defaults template file is meant to be for any VM booted by any cloudscheduler located at any address, the file uses variables configurable on the cloudscheduler online graphical user interface (GUI) to set many of the values in the defaults file. For HTCondor configuration, the address of the HTCondor master must be specified so that the worker VM can join the pool run by the master. This variable is configured with the "HTCondor FQDN" setting on the cloudscheduler GUI. Since the public IP address of the Kubernetes node is where outside connections must be sent for the HTCondor master to see them, this setting needed to be set to the domain name of the Kubernetes node. This posed a problem, however, since the HTCondor master was actually running on the private network of the pod. By specifying the "HTCondor container hostname" setting on the cloudscheduler GUI and making some changes to some of the cloudscheduler code the problem was remedied (See section 2.2.3). Currently there is no setting on the GUI for a different HTCondor port so a specific port is hard-coded into the defaults file but this is temporary as other changes beyond the scope of my project need to be made to allow this to be a setting on the GUI.

2.2.3 HTC Agent and Cloudscheduler changes

In order for cloudscheduler to retire a VM, an agent must send a command to the VM to stop HTCondor and retire the VM. While the agent is used regardless of whether HTCondor runs in the same machine as cloudscheduler or not, the agent needs to run in the same location as HTCondor. Because of this, the agent needed to be installed inside the HTCondor container. The HEPRC group had already written a script to install the agent if HTCondor was not running in the same machine as cloudscheduler and this script was modified and put into the container image so that the agent was installed when the container is run, along with the option to specify which git branch a user wishes to install the agent from.

Since the agent waits for messages from cloudscheduler, during the container startup the agent would fail and become inactive due to the fact that the cloudscheduler container had not finished initializing. This was fixed by modifying the agent service file; the agent was changed to attempt a restart every five seconds for five minutes if it failed. This worked, allowing the rest of the HTCondor container to initialize while the agent waited for cloudscheduler.

Other cloudscheduler subprocesses needed to be changed in order to communicate with HTCondor in another container. When cloudscheduler is running with its own public IP, the cloudscheduler machine poller and job poller use the "HTCondor FQDN" as the address to gather information on the HTCondor jobs and the VMs available to HTCondor. Since HTCondor was actually on the private container host, the code for these two pollers needed to be changed so that the "HTCondor container hostname" was being polled instead. Once these changes were made, the

pollers updated the database to use the "container hostname".

MariaDB contains "views" for other subprocesses to easily see all the relevant data for their process. These views compile specified data from other tables into one large table. Two of these views, namely view_available_resources and view_condor_host, needed to be modified to include the data on the "container hostname" in order for all of the relevant subprocesses to check the HTcondor located in the container.

2.3 InfluxDB Container

The third container runs InfluxDB, the database which stores the time series data for cloudscheduler, which is easily accessed by the cloudscheduler timeseries poller over the local network assigned to the pod. Few changes needed to be made to cloudscheduler when InfluxDB was moved to a separate container because the database is accessed via the localhost network, similar to MariaDB, although InfluxDB uses the network by default.

2.4 Cloudscheduler container

The fourth container runs the rest of the cloudscheduler processes including the web interface and the pollers which run checks and update the database accordingly. Functionality was added to the container image to allow a user to update to a different git branch than the one currently built into the container image.

The cloudscheduler status poller checks the cloudscheduler subprocesses and alerts the user on the GUI if one of them is having a problem or has failed. The status poller worked well inside of this container except for polling the database. The poller's code was changed to allow polling of the database from a different container. This was achieved by sending a dummy query to the database and recording if the query was successful.

The NodePort service was used in order to forward http and https requests to the web interface running inside the container along with forwarding for ssh connections. The default ports did not need to be changed in the container as the NodePort service can forward from a port in its range to any port exposed on the container.

The four container cloudscheduler ran smoothly once the changes for communicating with MariaDB and HTCondor were made and the NodePort service was in place. Some more changes should still be made, however, to further improve on the containerized cloudscheduler. These are outlined in the following section (3).

3 Future/Ongoing Work

Several small changes could be made in order improve the functionality of cloudscheduler running from containers. The most current and most pressing issues are outlined here.

3.1 MariaDB Container

The way the MariaDB container is currently built, the backup file of local configuration data is copied straight into the container image and in order to be changed a new image with alternate configuration data needs to be created. This was done since the cloudscheduler pod is created on a separate dedicated Kubernetes cluster which makes it difficult for a user to load their own configuration data. Although it would be ideal to have the user able to upload their own configuration data without having to create a whole new image, a decision needs to be make as to whether or not a user will be able to put their data on the Kubernetes cluster for the MariaDB container to access. Further research can also be done to find out if there is a way for the pod to get files from the user's machine instead of having the user access the Kubernetes cluster.

3.2 Cloudscheduler Container

As described in Section 2.2.2 the defaults.yaml.j2 template file should be able to be used for any VM booted by any cloudscheduler. Currently the COLLECTOR_PORT is hard coded into this file for container usage but a place to configure this on the GUI should be added to allow this to be a variable that changes depending on where cloudscheduler is being run. There may be a way to add the COLLECTOR_PORT to the file during the container initialization and this option could be explored if adding a place on the GUI is undesirable.

3.3 Far Future

Eventually work will need to be done with regards to HTCondor and web security, including how certificates will be handled and where they will go, as well as possibly improving the status poller's ability to poll processes in other containers. Further breaking up of the cloudscheduler container into smaller containers may also be pursued but these issues are less pressing and do not need to be taken care of immediately.

4 Other Work

There was extra time left over at the end of the work term after the containerization of cloudscheduler was complete. During this extra time I modified and configured mimic[11], to work with cloudscheduler. Mimic is an openstack mock api service. To cloudscheduler, mimic looks like a normal cloud where VMs can be booted and jobs can be run but mimic does not actually boot VMs, instead it creates a fake VM which has the option of being put in a variety of error states. This is useful for testing how cloudscheduler handles VMs that boot in error and other problems often

not reproducible in real systems. Appendix IV shows a file with some of the available error states in the format which mimic would read them.

Since the software had not been maintained since 2017, I made changes to the software to allow cloudscheduler to interact with mimic and simulate errors[12]. I also configured some mock networks and VM images for testing cloudscheduler as well.

5 Conclusion

A functioning cloudscheduler VM provisioning service was built and run out of a Kubernetes pod consisting of four Docker containers: one container holding the MariaDB database, one with the HTCondor job scheduler and agent, one with InfluxDB which stored the time series data for cloudscheduler, and one container with the rest of cloudscheduler's subprocesses. This pod is able to boot VMs based on submitted jobs, run jobs and retire superfluous VMs. Some changes still need to be made to allow the user to use their own backup data, along with some changes to improve the code and give correct status and error messages. Mimic was also configured and modified to allow testing of cloud errors on cloudscheduler.

6 Acknowledgements

I would like to thank the HEPRC group for all of their help in teaching me the various parts of cloudscheduler. I would like to thank Randy Sobie for the opportunity to work in software development through this coop. Thanks to Colin Leavett-Brown for helping me figure out MariaDB and improving db_upgrade and Marcus Ebert and Danika MacDonell for their assistance with figuring out HTCondor and the issues with running it out of containers. I would also like to thank Colson Drimmel for his help with all of the cloudscheduler python code and helping me decipher the various error messages I came across. Finally I would like to thank Rolf Seuster for all of his assistance week to week and his vision and guidance for the containerization of cloudscheduler.

References

[1] HEP RC. http://heprc.phys.uvic.ca/.

[2] Berghaus, F.; Casteels, K.; Driemel, C.; Ebert, M.; Galindo, F. F.; Leavett-Brown, C.; MacDonell, D.; Paterson, M.; Seuster, R.; Sobie, R. J.; Tolkamp, S.; Weldon, J. 10.

[3] Ansible: Simple IT Automation. https://www.ansible.com.

[4] What is a Container? https://www.docker.com/resources/what-container.

[5] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[6] Docker: Enterprise Container Platform. https://www.docker.com/.

[7] Tomecek, T. ansible-bender: A tool which builds container images using Ansible playbooks. https://github.com/TomasTomecek/ansible-bender.

[8] About MariaDB. https://mariadb.org/about/.

[9] HTCondor. 2019; https://research.cs.wisc.edu/htcondor/.

[10] InfluxDB: Purpose-Built Open Source Time Series Database. https://www.influxdata.com/.

[11] rackerlabs/mimic. 2019; https://github.com/rackerlabs/mimic, original-date: 2013-09-27T19:01:16Z.

[12] hep-gc/mimic. 2019; https://github.com/hep-gc/mimic/tree/mimic-csv2, original-date: 2019-12-02T08:01:16Z.

Appendix I: Sample Kubernetes YAML

Below is a sample yaml file which defines a Kubernetes pod containing an HTCondor container, an environment variable is defined for the collector port and that port is opened on the container. A service is also defined which opens the defined port to the outside via the NodePort service.

K8s.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: condor-test-svc
spec:
  selector:
    app: test
  ports:
  - name: condor
    nodePort: 31961
    port: 31961
    targetPort: 31961
  type: NodePort


---
apiVersion: v1
kind: Pod
metadata:
  name: testcondor
  labels:
    app: test
spec:

  containers:
  - name: condor
    image: mfens98/csv2-htcondor:0.6
    env:
    - name: CONDOR_COLLECTOR_PORT
      value: '31961'
    ports:
    - name: condorport
      containerPort: 31961
    securityContext:
      privileged: True
```

Appendix II: Sample DockerFile

Below is a sample Dockerfile which builds a MariaDB container along with the extra files and packages to build the cloudscheduler database. Comments are preceded by a '#' symbol.

Dockerfile:

```
FROM mariadb/server:10.3         #Which image to build on top of

RUN apt-get update -qq; apt-get install -qq \ #Packages to install
   libmysqlclient-dev \
   python3-mysqldb \
   python3-pip \
   software-properties-common \
   sudo \
   systemd; \
   apt-get update -qq \
   add-apt-repository ppa:deadsnakes/ppa; \
   apt-get update -qq; \
   pip3 install -q \
      sqlalchemy \
      pyyaml \
      pymysql; \

   #Edit configuration file
   sed -i '/max_connections/s/100/10000/g' /etc/mysql/my.cnf

#Copy files to build cloudscheduler database
COPY init.sh /docker-entrypoint-initdb.d/init.sh
COPY db_upgrade_files.tgz /
COPY localbackup.tgz /
COPY mysql-sudo /etc/sudoers.d/mysql

#Expose necessary ports
EXPOSE 3306
EXPOSE 22

# extract files and give mysql sudo access
RUN tar -xzf db_upgrade_files.tgz; chmod 600 /etc/sudoers.d/mysql

#When container is started, run this file with the CMD as an
option to this file
ENTRYPOINT ["docker-entrypoint.sh"]

CMD ["mysqld"]
```

Appendix III: Sample Ansible Playbook Definition YAML

Below is a sample YAML file which defines the playbook used to build a cloudscheduler container using Ansible-bender. The playbook runs several tasks which do the setup of the container image. Below can be seen the base image definition along with extra arguments for the playbook also a name for the target container image, which ports to expose and the entrypoint and container's main command.

```
csv2-container-build.yaml
---

- hosts: csv2
  vars:
    ansible_bender:
        base_image: docker.io/centos/systemd
        layering: False
        ansible_extra_args: "-e addenda=addenda -u root"

        target_image:
          name: csv2-container:newerVersion
          ports: 0-65000
          entrypoint: '["/init.sh"]'
          cmd: "/usr/sbin/init"

  roles:
  - csv2
```

Appendix IV: Configurable Error States for Mimic

Below is a few of the options for configuring error states for mimic, they are expressed as mimic would read them, in a dictionary. The first entry 'creates' a VM which takes 30s to build. The second 'creates' a VM in the ERROR state, and the third responds to a boot request with error code 500 and the message "No available hypervisors."

```
{"metadata" : {"server_building" : "30"}}
{"metadata" : {"server_error" : "true"}}
{"metadata" : {"create_server_failure" : "{\"message\" : \"No
available hypervisors\" , \"code\" : 500}"}}
```