University of Victoria
Engineering & Computer Science Co-op
Work Term Report
Spring 2018

Continuous Integration for a Cloud Management System

Department of Physics
University of Victoria
Victoria, BC

Tahya Weiss-Gibbons
V00832231
Work Term 3
Computer Science
tahyaw@uvic.ca

April 20, 2018

In partial fulfillment of the academic requirements of this co-op term

Supervisor's Approval: To be completed by the Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering Computer Science/Math Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be either returned to the student or, subject to the student's right to appeal a grade, held for one year after which it will be destroyed.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature: _____ Position: _____ Date: _____

Name (print): _____ Email: _____

Company Name: _____

Abstract

Cloudscheduler is a cloud management system developed by HEPRC which schedules batch jobs and manages virtual machines. There is currently a production version of cloudscheduler in use, as well as an updated version in development. It was desired for a continuous integration system be developed for both versions of cloudscheduler. Continuous integration is a setup of continually testing and integrating code as it is developed, to allow for easier deployment and detection of errors. To this end, a Jenkins pipeline was used. In order to maintain a fully isolated test, a cloud environment was simulated using libvirt, a virtualization API. This allowed cloudscheduler to run test job on virtual machine without having access to an external cloud.

Contents

List of Figures

Continuous Integration for a Cloud Management System

Tahya Weiss-Gibbons, tahyaw@uvic.ca

April 20, 2018

1 Report Specification

1.1 Audience

This report is intended for anyone with an interest in continuous integration. For users, it provides an introduction to continuous integration methods and cloud management systems. For developers, this provides an overview into the design an implementation of a continuous integration setup for cloudscheduler.

1.2 Prerequisites

Some basic knowledge of Jenkins, container software, virtual machines, cloud computing, libvirt, and database software is required.

1.3 Purpose

The purpose of this report is to document the process of creating a continuous integration setup for cloudscheduler. This includes setting up a local cloud, for testing purposes without having access to an external cloud.

2 Introduction

The High Energy Physics Research Computing group (HEPRC) at the University of Victoria is actively engaged in a variety of projects for the analysis of data from particle physics experiments as well as providing advice to researchers in other fields [1]. Activities include high speed networking, virtualization and cloud computing.

Cloudscheduler is a cloud management system create and maintained by the HEPRC group. Cloudscheduler launches and manages virtual machines (VMs) for jobs on different computational clouds. Currently an updated version of cloudscheduler is being developed.

One goal of the HEPRC group was to develop a continuous integration setup for cloudscheduler. Continuous integration is a method of integrating and testing code worked on by multiple people continuously throughout development. This allows for continual testing of the project and easier deployment.

3 Continuous Integration for Cloudscheduler

3.1 Jenkins Pipeline

Jenkins is a widely used method of continuous integration. A Jenkins Pipeline is a suite of plugins which create a Continuous Delivery Pipeline [2]. This is an automated process of compiling and testing code from source, simulating building the software in a reliable, repeatable and testable process. The definition of a Jenkins pipeline is written to a file, called a Jenkinsfile, and added to the repository for a project. This allows for automated builds of the code as it is updated for all branches and pull requests of the project, as well as code reviewing and version control of the pipeline. There are two different syntax's which can be used for Jenkins pipelines, declarative or scripted. For more information on syntax and use for this project, see Section 3.4.

3.2 Testing Setup for Cloudscheduler

Using Jenkins, the most effective way to perform basic tests on cloudscheduler is to setup a mock cloudscheduler deployment and try and run a test job. For both versions of cloudscheduler, this involves having a running condor manager, setting up and installing cloudscheduler on the same machine, submitting a test job to condor, launching a virtual machine to run the test job and then ensuring that the job runs and finishes correctly on the VM. To do this, a docker container was used as a node in Jenkins.

3.3 Cloudscheduler Container

Docker is a container management software, used for developing Linux container systems. Jenkins allows the use of docker containers as nodes in a pipeline. This can be done either using publicly accessible container images on Docker hub, giving the pipeline a Dockerfile for it to run and compile it's own image or using a local image on the host which Jenkins has access to. For this project, a local Docker image was used. For both versions of cloudscheduler, condor was installed in the container image, as well as all the necessary python modules and libvirt. For the updated version of cloudscheduler, python 3 and python 2 were both added to the image. This was due to cloudscheduler updating to python 3 for most of it's code, other than the pollers, which still ran with python 2. Due to the localhost cloud setup, for more information see Section 4, the cloudscheduler containers had to be run with the privileged flag. While this then compromised the isolation of the Docker container, it was the only solution available for running a localhost cloud setup.

```
###Descriptive Pipeline Syntax###
pipeline {
    agent {
        docker image {centos:7}
    }
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}
```

Figure 1: Basic example of descriptive syntax for a Jenkins pipeline. An agent is given to run the stages on, which are run in sequence. A stage defines a conceptually distinct set of tasks [2], in this example Build, Test and Deploy.

```
###Scripted Pipeline Syntax###
node {
    checkout scm
    docker.image('centos:7').inside(){
        stage('Build') {
            //
        }
        stage('Test') {
            //
        }
        stage('Deploy') {
            //
        }
    }
}
```

Figure 2: Basic example of scripted pipeline syntax for a Jenkins pipeline. A node defines what will do the core work for a pipeline and defines a workspace for the work to be done. Checkout scm will import the project from git to the workspace, and then inside of a docker image stages can be run in a similar manner to the descriptive pipeline.

3.4 Scripted vs Declarative Pipeline

There are two different syntax's which can be used to write a Jenkins pipeline definition, scripted or declarative. Declarative is a more recent feature of Jenkins which allows for easier code readability and specific syntax features. See Figure 1 for a basic example of this syntax. Initially, the pipeline for the older version of cloudscheduler was developed using declarative syntax. While initially this was easier for running a simple pipeline, there were some limitations with the declarative syntax. This included problems with retrieving the log files upon failure of the pipeline, and complications with running multiple parallel containers. Scripted syntax was used in the end for both version 1 and 2 pipelines as the syntax was less restrictive. Scripted syntax is based off of a Groovy compiler, and as such is a fully featured programming environment. Groovy is a dynamic programming language, which is designed to be integrated with any Java program, such as Jenkins. For an example of scripted syntax, see Figure 2.

4 Simulating a Cloud Environment

In a typical production environment, cloudscheduler runs on the same machine as the condor central manager, collects jobs and clouds on which these jobs should be run from the manager and then manages the VMs on specified clouds to match the requirements of the jobs. This required access and communication with an external cloud, which was not a desired require-

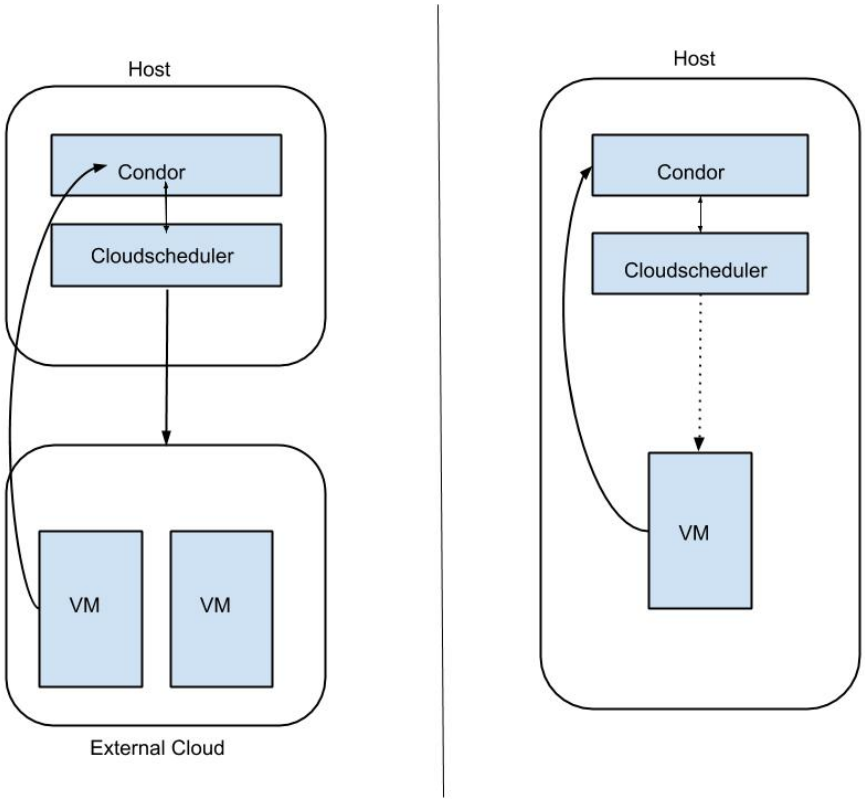

Figure 3: In a typical cloudscheduler setup, as seen on the left, condor and cloudscheduler run on the same host, while cloudscheduler connects to an external cloud to manage VM's for condor jobs. Simulating this setup for the Jenkins pipeline, cloudscheduler was extended to use the libvirt module in order to create VM's directly on the host.

ment for testing purposes. An alternate solution was used, where libvirt was used to simulate a cloud environment locally, see Figure 3. Libvirt is an opensource, hypervisor-independent virtualization API [3]. With libvirt included within a container, cloudscheduler could start, poll and remove VMs locally and no externally cloud was needed, allowing for a more isolated testing environment. This solution was used for both versions of cloudscheduler.

4.1 Localhost Cloud Type Version 1

In the first version of cloudscheduler, the clouds which cloudscheduler has access to are specified in a configuration file, along with information needed for connecting and utilizing that cloud. When a job is submitted to condor, it will have a target cloud specified. If the target cloud matches a cloud cloudscheduler has knowledge of and can connect to, cloudscheduler will attempt to run the job on that cloud. To simulate this behaviour with libvirt, a new cloud type was added called localhost. See Figure 4 for an example localhost cloud entry. A module was written using the libvirt python extension for cloudscheduler to perform the

```
[container-cloud]
cloud_type = localhost
vm_slots = 10
cpu_cores = 1
memory = 8000
networks = default
```

Figure 4: Example of a cloud_resources entry for a local cloud in cloudscheduler version 1. Cloudscheduler will read the the cloud_resources entries and manage jobs on these clouds.

same tasks as an external cloud locally. The limitations of this 'cloud' were then the same as the limitations of the host and its virtualization capabilities.

4.2 Localhost Version 2

For the new version of cloudscheduler, the principal of creating a localhost cloud was similar, though some specifics were changed. This was due to a both cloudscheduler using a database to store information, see Section 5, as well as the expanded use of pollers. Pollers are services running with cloudscheduler which check the current state of the system and update the database as needed. This gave current and accurate information to cloudscheduler about the status of condor, as well as any clouds it was connected to. A localhost poller was needed for the newer version of cloudscheduler. This was done in a similar manner to already existing pollers and ran as a service inside of the container. Multiple process threads were started in the poller, and updated the database as required. This allowed for cloudscheduler to have current, accurate information on domain states, running domains and to verify that any commands requested are indeed executed. The other expansion for the localhost cloud needed for version 2 of cloudscheduler was in connection with the database. Specific configuration files needed for localhost were now stored in the database, such as the user data for launched VM's and with the creation of groups in the newer version of cloudscheduler, a local group had to be added to the database.

4.3 Network Connections & Contextualization

Once a VM had been created by cloudscheduler using libvirt, it needed to be able to communicate back to the condor central manager. Libvirt provides a default virtual network, which was automatically enabled and allows for connections with a created domain and the host [5]. Virtual machines launched using cloud images by libvirt still have to be contextualized, to contain the correct user data and network connections. When this is done by a cloud, such as Openstack or Amazon, this is done using a meta data server, which the VM connects with on boot and is passed the correct meta data and user data. When simulating this environment, this can also be done using cloud-init and reading configuration information off of a simulated CD-ROM device [6]. For this method, a meta data and user data file are

```
general:
    clouscheduler_log_file: "/var/log/cloudscheduler/cloudscheduler.log"

database:
    db_user: "csv2"
    db_password: "password"
    db_host: "db"
    db_port: 3306
    db_name: "csv2"
```

Figure 5: Example configuration file for cloudscheduler version 2 connecting to an linked database container. When run the cloudscheduler container links the database container, with db as the linked name. This file is stored centrally in the container, and is the only configuration file not stored in the database.

required. The user data file includes all information needed for the VM to start condor and connect to the central manager. The meta data file provides host information to the VM, including host name and instance id. Using these two files, a configuration drive is created, and attached to the domain definition. SSH keys can also be added into the user data file, to allow a user to log into the VM to test the contextualization and connection.

5 Database for Cloudscheduler Version 2

5.1 Setting up Database

For the newer version of cloudscheduler, a database was used to store current state information, as well as configuration. MariaDB was used for this project, which is an opensource database software based off of MySQL [4]. To emulate this in a testing environment, parallel containers were started within the pipeline, with one container running the condor central manager and cloudscheduler, the other running mariaDB. The mariaDB container was configured by having the schema for all of the tables saved in the repository and importing the table definitions upon starting the container. As some table definitions were dependant on other tables, a list was created of the table names, providing an order for the tables to be added to the new database. Some necessary information then needed to be inserted into the database, such as a localhost cloud definition, a local group and standard yaml configuration files. Both containers could then run in parallel with the cloudscheduler container linked to the database container, allowing cloudscheduler to fetch information from and update the database as needed. Connection settings for the database were provided through a configuration file stored on a central location on the cloudscheduler host. For an example of this file, see Figure 5.

5.2 Database Connection

There was one difficulty with establishing a connection between the two containers was with the driver used by cloudscheduler to connect with the database. The original connection used by cloudscheduler, when both cloudscheduler and the database were on the same host is shown below.

```
create_engine("mysql:/// + csconfig.config.db_user + ":" +
    csconfig.config.db_password + "@" + csconfig.config.db_host + ":" +
    str(csconfig.config.db_port) + "/" + csconfig.config.db_name)
```

When the mariaDB server was on a separate host, even with the mariaDB client installed on the cloudscheduler host, this connection was not able to be made through python. This was solved by installing the python MySQL module and changing the database connection call in cloudscheduler.

```
create_engine("mysql+pymysql:/// + csconfig.config.db_user + ":" +
    csconfig.config.db_password + "@" + csconfig.config.db_host + ":" +
    str(csconfig.config.db_port) + "/" + csconfig.config.db_name)
```

6 Running Test Jobs

Using the localhost cloud, and with cloudscheduler properly configured, a test job could then be run in a Jenkins pipeline to test cloudscheduler. With all services running, a pipeline would submit a test job to condor. Then it would monitor for the job showing up in cloudscheduler, the VM being launched and showing up in both cloudscheduler and virsh, the libvirt command line interface, for the VM registering to condor, the job matching to the VM and finally for the job running and finishing on the VM. If at any point the behaviour is not expected, a failure occurs or the VM does not register within the wait time, the pipeline will pull the log files and exit, raising a failed flag. By handling an error as it occurs, it allows for a clean exit and extracts needed information to debug the failure at a later time.

While this pipeline tests the basic functionality of cloudscheduler, it is a small scale test of the capabilities of the cloud management. For future development, it would be desirable to expand this pipeline to test cloudschedulers management of multiple batch jobs. This would better test the system as it would more closely mimic a production environment. This pipeline could also be expanded to test more advanced features of cloudscheduler and aid in their development.

7 Conclusion

Continuous integration allows for earlier detection of potential bugs in code, as well as easier deployment of a project. Using a Jenkins pipeline, continuous integration was setup for

both versions of cloudscheduler. This was done by running a mock deployment setup. A docker image was used as as a node in the pipeline, and cloudscheduler was installed from the most current branch of the git project. Then a test job would be submitted to condor, and cloudscheduler would try and process the job, launch a VM, wait for the job to run and remove it. In order to achieve this, a cloud was simulated locally. This allowed for a fully contained testing environment, which was not reliant on an external cloud. This similar setup was done for both versions of cloudscheduler. The main differences between the two versions included adding a database and pollers to cloudscheduler version 2. For the Jenkins pipeline, the database was run a separate container running in parallel with the cloudscheduler container, with the two containers linked together. This required the database connection to be changed within cloudscheduler to use the python mysql driver. If the test job failed at any point, the log files of cloudscheduler, and condor would be pulled from the container and stored for future debugging. This pipeline only provides basic tests of the cloudscheduler functionality and provides a starting pointing for expanding the pipeline to include testing multiple, more complicated jobs as well as testing additional features within cloudscheduler.

8 Glossary

cloud-init Multi-distribution package to handle early contextualization of cloud images

Container A virtualization technique which can create multiple isolated Linux environments using the same host kernel

Docker A container management software for developing and sharing container systems

Domain A running virtual machine, or a configuration which can be used to launch a virtual machine using libvirt

Jenkins An automation engine for continuous integration

MariaDB An opensource fork of MYSQL database

Node A machine or container which is capable of executing a Jenkins pipeline

VM Virtual Machine, an instance of a machine running in software

yaml Data serialization language

References

[1] "High Energy Physics Research Computing", *High Energy Physics Research Computing* [Online] Available: `http://heprc.phys.uvic.ca/`

[2] "What is Jenkins Pipeline?", *Jenkins User Documentation* [Online] Available: `https://jenkins.io/doc/book/pipeline/`

[3] "Chapter 1. Introduction", *Libvirt Application Development Guide Using Python* [Online] Available `https://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/index.html`

[4] "About MariaDB", *MariaDB Foundation: Supporting Continuity and Open Collaboration* [Online] Available `https://mariadb.org/about/`

[5] "Networking", *Libivrt Wiki* [Online] Available `https://wiki.libvirt.org/page/Networking`

[6] "Booting Cloud Images with Libvirt", *Odd Bits* [Online] Available `http://blog.oddbit.com/2015/03/10/booting-cloud-images-with-libvirt/`