University of Victoria
Engineering & Computer Science Co-op
Work Term Report
Summer 2022

Continuous Integration System for the Cloudscheduler VM Provisioning Service

High Energy Physics Research Computing Group
Department of Physics, University of Victoria
Victoria, BC, Canada

Victor Kamel
V00963333
W-2
Computer Science
vkamel@uvic.ca

August 26, 2022

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

Supervisor's Approval: To be completed by 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 and Computer Science Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be retained and available to the student or, subject to the student's right to appeal a grade, held for one year after which it will be deleted.

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

Signature Colson Driemel Position: Research Assistant Date: Aug 26th 2022

Name (print): Colson Driemel E-Mail: colsond@uvic.ca

For (Company Name) University of Victoria

Executive Summary

The High Energy Physics Research Computing (HEPRC) group at the University of Victoria manages cloud resources for high energy physics applications. To that end, they develop the cloudscheduler v2 service, which schedules the creation of virtual machines on clouds to run high throughput computing physics jobs. However, the current cloudscheduler development life cycle does not make full use of the continuous integration software development practices that are common in industry. Therefore, a continuous integration server is created in order to automate regular deployments of the cloudscheduler system to a virtual machine in the cloud, running the available test suites against it and reporting the test results. This allows the HEPRC group to make better use of the advantages that the continuous integration practice can provide such as preventing errors from accumulating by fixing them earlier in the development cycle, increasing the overall stability of the cloudscheduler v2 system by running regular tests and increasing developer productivity by reducing the difficulty of troubleshooting errors. In this report, several candidate software packages are tested, and a solution using Jenkins CI is selected and implemented.

Table of Contents

List of Figures and Tables

Figures

Tables

Glossary

CLI	*Command Line Interface.* Text-based user interface used in a computer terminal.
cloud	Computational resources available on-demand over the network.
cron	Linux utility for running commands periodically.
DSL	*Domain-Specific Language.* Specialized programming language targeted at a particular problem domain.
FOSS	*Free and Open Source Software.* Software distributed under a free license with publicly-available source code.
HTTP	*Hypertext Transfer Protocol.* Protocol for transmitting information over the internet.
hypervisor	Software that creates, manages and runs Virtual Machines.
SaaS	*Software as a Service.* Cloud-based software delivery strategy where the code is centrally hosted and licensed by subscription.
SCM	*Source Code Management.* Version control for source code.
self-hosted	Service running on a privately owned web server.
SSH	*Secure Shell.* Secure network protocol for remote machine access.
VM	*Virtual Machine.* An emulation and/or virtualization of a computer system.
XML	*Extensible Markup Language.* A hierarchical form of text-based data representation.
YAML	*YAML Ain't Markup Language.* Data-oriented language typically used for writing configurations files.

1 Introduction

The High Energy Physics Research Computing (HEPRC) group in the Department of Physics and Astronomy at the University of Victoria is a research group focused on the management of cloud resources for projects such as the Belle II experiment [1] at the KEK Laboratory in Japan and the ATLAS experiment [2] at the CERN Laboratory in Switzerland. They develop and maintain the cloudscheduler VM provisioning service, currently in its second major iteration (CSV2), that automatically starts VM instances on clouds in order to run physics jobs on demand [3]. The original version of cloudscheduler was created in order to streamline the use of infrastructure as a service clouds for high-throughput computing projects [4], and was fundamentally redesigned in its second version to make use of new technologies and software designs that have become available since its original conception [5]. Cloudscheduler v2's source code is now written primarily in Python 3 [6], and has a web interface written with the Django framework [7], along with a command line interface (CLI). Due to the complexity inherent in interfacing between the different cloud configurations, CSV2's database, and other services such as the HTCondor job scheduler [8], several test suites for CSV2 have been created. The original set of tests for cloudscheduler, the unit test suite, is written in Python 3 and tests interactions with CSV2, both via the CLI and directly with HTTP requests. However, there have since been additional tests written by past co-op students [9, 10] that directly test interactions with the web interface using the Selenium framework [11] along with the Python built-in library unittest [12]. These tests support the Chrome, Firefox, Opera and Chromium web browsers and will be referred to as the web test suite.

Currently, the cloudscheduler development life cycle consists of developing code on a test machine running CSV2, deploying this code using the standard upgrade process via the Ansible automated deployment software [13] to a secondary machine running the previous version of cloudscheduler. If the deployment is successful, the unit and web tests can be run. Once all of the planned changes have been implemented, a release tag is created with the most recent changes in

Git [14] and the new version is deployed into production.

However, there are some issues and inefficiencies inherent in the current process. Full deployments of the cloudscheduler software onto a new machine are rarely tested since the development servers are not deployed from scratch, and changes made during development are not always fully tested against the existing test suites until a test deployment can be made. In addition, the running of the test suites themselves is not automated to a satisfactory degree. Thus, the HEPRC group may benefit from adopting some subset of the continuous integration and continuous delivery processes.

1.1 Continuous software engineering

Continuous software engineering practices such as continuous integration, delivery and deployment are software development practices that involve increasing the frequency of the production of new features by shortening the cycle of developing, testing, and releasing code changes. In [15], researchers define the practices as:

1. **Continuous Integration (CI)**. The practice of integrating code into a central repository frequently. This involves testing that builds succeed, and pass the available tests. This is typically done automatically by a CI server;

2. **Continuous Delivery (CDE)**. The practice of keeping the application in a production-ready state;

3. **Continuous Deployment (CD)**. The practice of automatically deploying changes into production.

Professionals report that the benefits of continuous integration practices involve improved project predictability due to finding errors quickly, as well as improved developer productivity due to the increased parallelism and more effective troubleshooting [16]. In addition, open source projects that use CI were found to have more frequent releases, accept pull requests faster and enable developers to worry less about breaking the build [17].

Table 1: Run lengths of past 3 unit and web test runs.

Test Suite	Run 1	Run 2	Run 3	$\bar{x}$	s
Unit Tests	79.00 min	47.72 min	48.10 min	58.27 min	17.95 min
Web Tests (full)	19.00 h	17.00 h	18.17 h	18.06 h	0.820 h
Web Tests (Chrome & Firefox)	9.57 h	9.60 h	9.25 h	9.47 h	0.193 h

1.2 Problem definition

Due to the distributed nature of the cloudscheduler system, the design of the test suites and other factors such as network latency, the testing suites vary in how long they take to run. Provisioning VMs on the HEPRC group's hypervisors is also highly dependent on the volume of jobs currently running on the clouds. In addition, while the unit test and the web test suites can be run in parallel, two instances of either suite cannot run concurrently against a single cloudscheduler instance as they would interfere with each others' operation. As seen in Table 1, the unit tests only require about an hour to run, while the web tests can take as long as 19 hours for the full test suite. Therefore, running the complete testing suite for every commit made to source control—which would need to be done multiple times a day—is not practical in this case. Therefore, the objective is not to implement a complete CI/CD pipeline as reported in the literature, but rather to implement certain aspects.

1.3 Requirements

For the purposes of the HEPRC group, the important aspects of an automated testing system are the ability to:

1. run on a set schedule or automatically when a commit to the repository is detected;
2. provision a virtual machine to run the tests against;
3. deploy the latest development version of cloud scheduler to this VM;
4. automatically run the unit and web test suites;
5. and report results of successful and failing tests.

In addition, free, open source (FOSS) and self-hosted solutions are preferable where possible, since the HEPRC's infrastructure can support maintaining their own servers.

2 Automating Continuous Integration

To implement the desired functionality, a CI server must be created to provision a cloudscheduler instance, configure it to connect to a cloud suitable for testing and to automatically run the tests.

2.1 Options considered

There exist a large number of different software solutions for creating an automated CI server. Most code hosting websites such as GitHub [18] and GitLab [19] provide built-in options for CI. In addition, there are dedicated web-based services that provide CI as a service. Finally, there are some self-hosted options available that forgo centralized services.

A large differentiating factor between these different CI systems is their compatibility with the SCM system that is employed by the development team, as well as support for the build tools required by the software being tested. However, since CSV2 is written in Python, a language that is interpreted rather than compiled, compatibility with build systems (such as Make, Ant or Maven) are not an important consideration here. Due to the ubiquity of Git as an SCM system, most of the options under potential consideration support it.

2.1.1 GitHub Actions and GitLab CI

GitHub Actions [20] and GitLab CI [21] are features provided by GitHub and GitLab respectively for CI integrated into their platforms. They both offer a certain number of free CI minutes, depending on whether the repository is private or public and the subscription tier, that can be used to run build steps on their machines [22, 23]. Since the HEPRC group has their own OpenStack cloud, and the cloudscheduler deployment process requires a non-trivial amount of setup, there is a signif-

icant advantage to running the tests and deployments on their own cloud rather than the SaaS VMs provided by GitHub or GitLab. This can be done using a custom runner, and would also avoid the problem of limited CI minutes since time on self hosted runners does not count towards the time allotment [24, 25]. Since the cloudscheduler code base is currently hosted publicly on GitHub (see [26]), GitHub Actions would integrate well with the current development infrastructure. However, for the Actions service, there is still a risk of hidden fees, and the CI server itself is not self hosted, which makes it a difficult choice to recommend. Since GitLab can be self hosted, this is less of an issue. However, as this would involve the complete migration of the cloudscheduler code base, this option would only be considered if the HEPRC group would eventually want to explore hosting their own SCM system.

2.1.2 Travis CI

Travis CI [27] is a CI/CD SaaS platform for automating testing and building software hosted on Assembla, Bitbucket, GitHub or GitLab founded in 2011 [28]. It is configured with a `.travis.yml` YAML file located in the root of the code repository, and builds are triggered when code is pushed to the repository [29]. Travis CI used to provide free services to open source projects which made it popular for this purpose. However, since Travis is not fully self hosted and now only offers a 30-day free tier [30], it is not a good fit for our needs.

2.1.3 BuildBot

BuildBot [31] is a job scheduling system written in Python with the Twisted asynchronous networking framework [32]. The system consists of a master node that monitors a code repository, and a set of worker nodes that are available to run jobs as required and report results when they are completed [31]. Configuration is done on the master node with a Python script that combines built-in components. BuildBot is open source, self-hosted and available in the Python Package Index [33]. In addition, since BuildBot does not have many dependencies, it has a lighter resource footprint when compared to other candidate options.

2.1.4 Jenkins

Jenkins [34] is an open source, self-hosted automation server that implements all parts of the the continuous integration and continuous delivery processes. The Jenkins project was originally named Hudson, but the two projects diverged in 2011 and the Hudson project is no longer maintained [35]. Jenkins depends on the Java Runtime Environment and build steps are configured in the Apache Groovy Java DSL [36] or a declarative pipeline syntax based on the former. Jenkins also has many plugins that extend its built-in functionality and provide integration for additional build steps.

2.2 Test Configurations

Jenkins and BuildBot are selected for further testing, as these software systems are particularly well suited to the requirements due to the fact they are free, open source, self-hosted and compatible with the existing architecture.

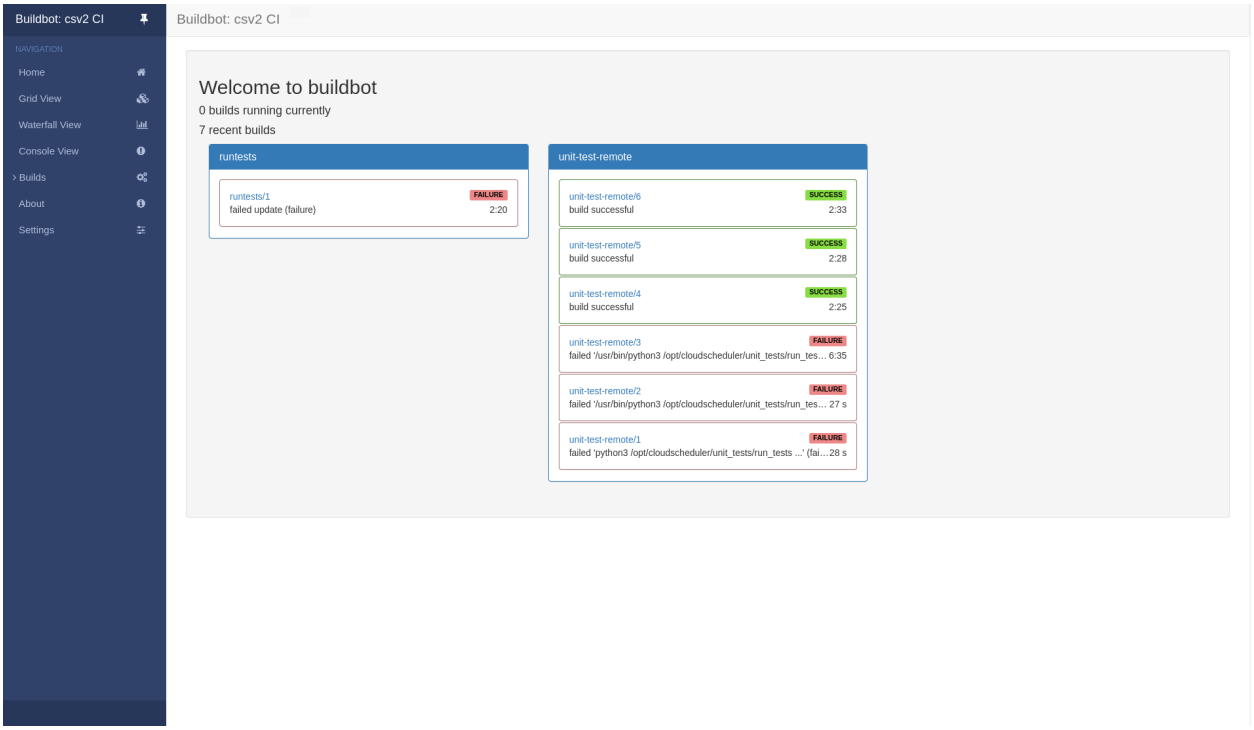

Figure 1: BuildBot web interface.

2.2.1 Buildbot

Setting up BuildBot is very simple as it only requires some packages to be installed with the Python package manager PIP. Once the BuildBot software is installed on the master and the node, it is possible to configure with the `master.cfg` file on the master node's file system. Build steps are also written into this file in Python. After starting the `master` service with the `buildbot` command, the BuildBot web interface can be accessed (as shown in Figure 1).

Although BuildBot is relatively easy to install, it is more difficult to configure relative to other options since most configuration options are not accessible through the web interface and must be changed in the configuration file. This includes build steps as well.

Another large area where BuildBot is lacking in comparison to Jenkins is in the number of plugins that support integration with services such as OpenStack, and the quality of the visualization and reporting tools available.

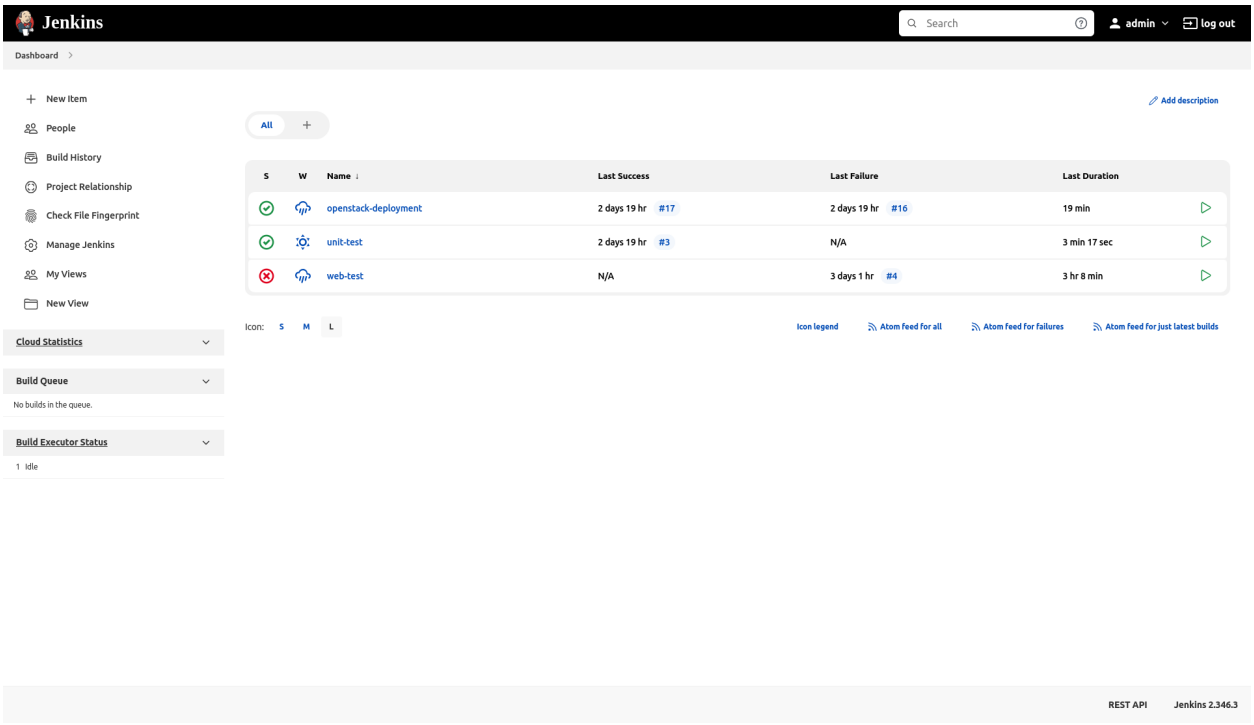

Figure 2: Jenkins web interface.

2.2.2 Jenkins

Since Jenkins is written in Java, first the Java Development Kit (JDK) must be installed as a dependency in order for the `jenkins` package to be installed from the Jenkins repositories. The Jenkins service can then be started and the Jenkins CI server configured using the web interface shown in Figure 2.

Everything further, including user accounts, build pipelines and plugins can be configured from the web interface. This makes Jenkins easier and more convenient to configure, as it does not require the developer to log in to the CI server from the command line via SSH in order to change settings. In addition, Jenkins has very strong support for plugins that make sending email alerts, collecting test data and interfacing with OpenStack clouds simpler that BuildBot.

Although Jenkins CI requires Java and more system resources to run in general, it was determined that this did not have any noticeable impact on the run time of the tests themselves. As seen

Table 2: Comparison of run lengths for methods of initiating tests, $n = 3$.

Method	Run 1	Run 2	Run 3	$\bar{x}$	s
Command Line	126 s	128 s	129 s	128 s (2 min 8 s)	1.53 s
Jenkins	148 s	140 s	136 s	141 s (2 min 21 s)	6.11 s
BuildBot	153 s	148 s	145 s	146 s (2 min 26 s)	7.13 s

in Table 2, both Jenkins and BuildBot have about the same overhead when compared to running the tests directly from the command line, with Jenkins having a slight edge of about five seconds when run against a small subset of the unit test suite (the `alias` tests). Therefore, since Jenkins was determined in general to be more feature complete than BuildBot, Jenkins was chosen for the final system.

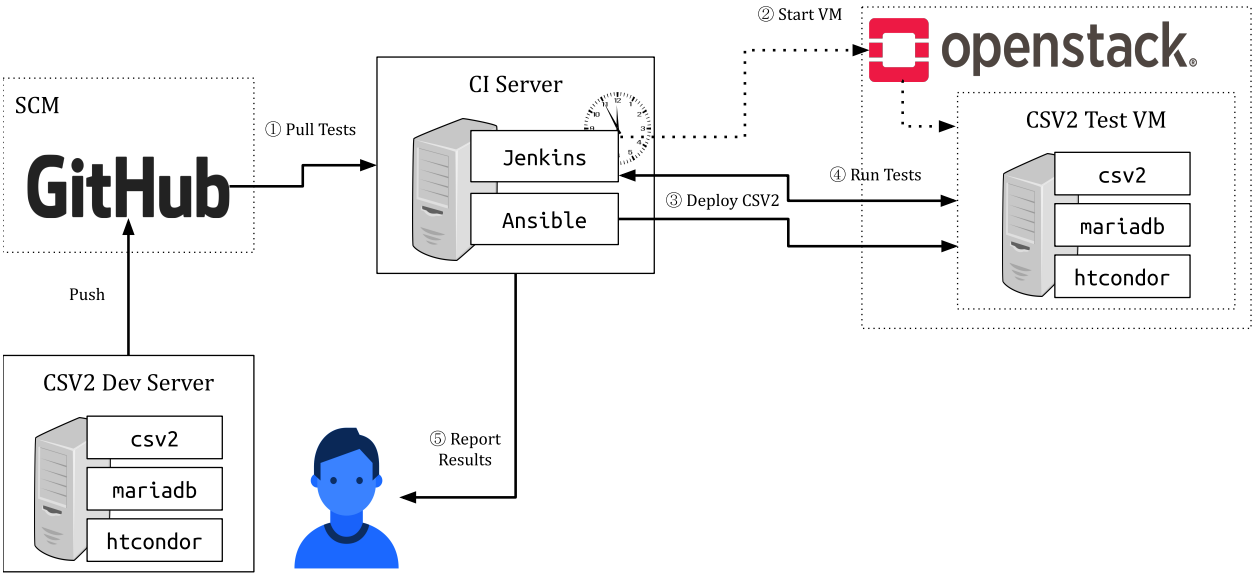

Figure 3: Diagram of CI system.

2.3 Final system design

In the finalized system, outlined in Figure 3, the central CI server orchestrates the entire automated testing system. Within the Jenkins CI interface, there are several pre-configured pipelines that can

either be run manually or on a schedule. The `openstack-deployment` pipeline (Appendix A) can provision a new virtual machine on an OpenStack cloud, and further deploy the CSV2 system to the virtual machine. Finally, the virtual machine is set as the target for the unit and web tests, which can each be run as their own separate `unit-test` and `web-test` pipelines (Appendices B and C). The unit tests are scheduled to run once every evening (Monday to Friday), while the web test run over the weekend since they take significantly longer to run to completion. In addition, either set of tests can be triggered manually, or per-commit if only a portion of the tests are run.

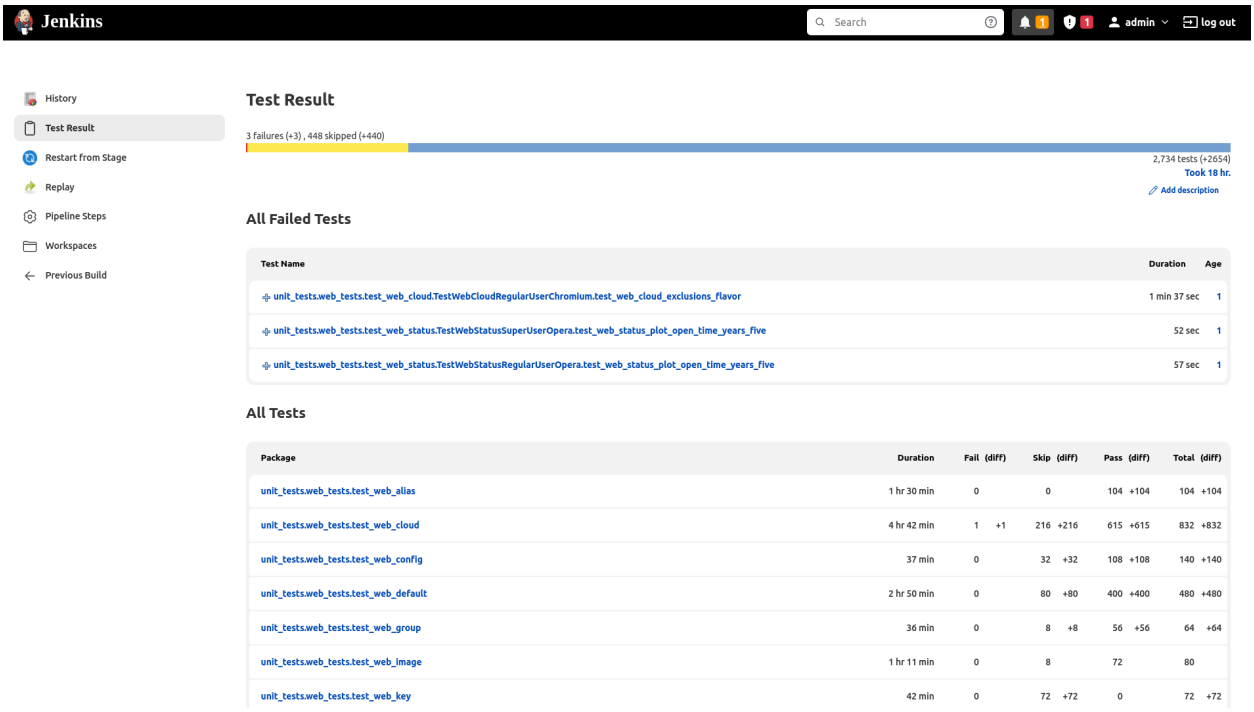

Figure 4: Jenkins test result interface.

Since Jenkins has support for collecting and displaying test data as JUnit-style XML [37], the unit test suite runner was extended to provide output in XML. The web tests were originally written and run with the Python unittest testing framework. However, it has no substantial support for exporting test output as XML. Therefore, web tests were transitioned to pytest [38], which is compatible with collecting and running unittest-style tests, in addition to generating XML test output. Making these changes allow Jenkins to display the test output in the web interface, as well as to track failing tests

and overall statistics as seen in Figure 4.

Due to the unstable nature of some of the web tests and the fact that some tests can be sensitive to network conditions, failed tests are rerun before they are considered to have failed using the `pytest-rerunfailures` plugin [39]. If failed tests are detected at the end of a build pipeline, the Jenkins CI server can notify the developers of the failing tests so that those tests can be fixed.

2.4 Deployment

Since many of the software systems managed by the HEPRC group are installed with the Ansible remote administration and deployment tool, an Ansible playbook was created to deploy the CI server to a newly-created virtual machine on a cloud or hypervisor. In the event of a failure of the CI server, it can be recreated with this deployment system.

2.5 Configuration

Virtual machine targets can be provisioned automatically by Jenkins with the OpenStack Cloud plugin [40], thus administrators only need to provide the CI server with credentials to the current OpenStack instance used for testing. The other parameters that can be configured are the branch used for deployment to the target cloud and the default passwords, which are specified in the OpenStack installation pipeline (Appendix A). In addition, each Jenkins pipeline can be scheduled to run at pre-determined intervals using a cron-like format in the web interface. Email notification settings are also configured with the web interface.

3 Conclusions

Several aspects of Continuous Integration and Continuous Delivery were integrated into the cloud-scheduler development life cycle by creating a CI server that can automatically provision an OpenStack virtual machine, deploy the latest CSV2 code and run the test suite against it at regular

intervals. This allows many advantages of automated CI/CD pipelines to become available during CSV2 development such as increased developer parallelism, earlier identification of errors, increased stability of cloudscheduler system and faster deployments.

Acknowledgements

I would like to thank Colson Driemel, Marcus Ebert and the other members of the HEPRC group for their consistent and ongoing support during my work term.

References

[1] Belle II, "Belle II Home Page," *belle2.org*. [Online]. Available: https://www.belle2.org/ [Accessed Aug. 8, 2022].

[2] CERN, "The ATLAS Experiment," *atlas.cern*. [Online]. Available: https://atlas.cern/about/ [Accessed Aug. 8, 2022].

[3] F. Berghaus, K. Casteels, C. Driemel, M. Ebert, F. F. Galindo, C. Leavett-Brown, D. MacDonell, M. Paterson, R. Seuster, R. J. Sobie, S. Tolkamp, and J. Weldon, "High-throughput cloud computing with the cloudscheduler vm provisioning service," *Computing and Software for Big Science*, vol. 4, no. 1, p. 4, Feb. 2020. [Online]. Available: https://doi.org/10.1007/s41781-020-0036-1 [Accessed Aug. 15, 2022].

[4] P. Armstrong, A. Agarwal, A. Bishop, A. Charbonneau, R. J. Desmarais, K. Fransham, N. Hill, I. Gable, S. Gaudet, S. Goliath, R. Impey, C. Leavett-Brown, J. Ouellete, M. Paterson, C. Pritchet, D. Penfold-Brown, W. Podaima, D. Schade, and R. J. Sobie, "Cloud scheduler: a resource manager for distributed compute clouds," *CoRR*, vol. abs/1007.0050, June 2010. [Online]. Available: http://arxiv.org/abs/1007.0050 [Accessed Aug. 15, 2022].

[5] R. Sobie, F. Berghaus, K. Casteels, C. Driemel, M. Ebert, F. Galindo, C. Leavett-Brown, D. MacDonell, M. Paterson, R. Seuster, S. Tolkamp, and J. Weldon, "Cloudscheduler: a vm provisioning system for a distributed compute cloud," *EPJ Web Conf.*, vol. 245, p. 07031, Nov. 2020. [Online]. Available: https://doi.org/10.1051/epjconf/202024507031 [Accessed Aug. 15, 2022].

[6] Python Software Foundation, *Python*. [Online]. Available: https://www.python.org/ [Accessed Aug. 8, 2022].

[7] Django Software Foundation, *Django*. [Online]. Available: https://www.djangoproject.com/ [Accessed Aug. 15, 2022].

[8] University of Wisconsin-Madison, *HTCondor*. [Online]. Available: https://htcondor.org/ [Accessed Aug. 15, 2022].

[9] G. Robertson, "Creating unit tests for cloudscheduler's web interface," Apr. 2020. [Online]. Available: http://heprcdocs.phys.uvic.ca/reports/robertson-wtr-2020.pdf [Accessed Aug. 17, 2022].

[10] E. Klassen, "Testing frameworks for the cloudscheduler web interface," Apr. 2021. [Online]. Available: http://heprcdocs.phys.uvic.ca/reports/klassen-wtr-2021.pdf [Accessed Aug. 17, 2022].

[11] Software Freedom Conservancy, *Selenium*. [Online]. Available: https://www.selenium.dev/about/ [Accessed Aug. 15, 2022].

[12] Python Software Foundation, "unittest — Unit testing framework," *unittest*. [Online]. Available: https://docs.python.org/3/library/unittest.html [Accessed Aug. 20, 2022].

[13] Red Hat, *Ansible*. [Online]. Available: https://www.ansible.com/ [Accessed Aug. 20, 2022].

[14] Software Freedom Conservancy, *git*. [Online]. Available: https://git-scm.com/ [Accessed Aug. 20, 2022].

[15] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, Mar. 2017. [Online]. Available: https://doi.org/10.1109/ACCESS.2017.2685629 [Accessed Aug. 23, 2022].

[16] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *ICSE 2013*, Mar. 2013, [Accessed Aug. 23, 2022].

[17] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 426–437, [Accessed Aug. 23, 2022].

[18] GitHub, Inc., *GitHub*. [Online]. Available: https://github.com/ [Accessed Aug. 20, 2022].

[19] GitLab B.V., *GitLab*. [Online]. Available: https://about.gitlab.com/ [Accessed Aug. 20, 2022].

[20] GitHub, Inc., "Automate your workflow from idea to production," *GitHub Actions*. [Online]. Available: https://github.com/features/actions [Accessed Aug. 20, 2022].

[21] GitLab B.V., "GitLab CI/CD," *GitLab CI/CD*. [Online]. Available: https://docs.gitlab.com/ee/ci/ [Accessed Aug. 20, 2022].

[22] GitHub, Inc., "About billing for GitHub Actions," *GitHub*. [Online]. Available: https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions [Accessed Aug. 20, 2022].

[23] GitLab B.V., "GitLab Pricing," *GitLab*. [Online]. Available: https://about.gitlab.com/pricing/ [Accessed Aug. 20, 2022].

[24] GitLab B. V, "GitLab Runner," *GitLab*. [Online]. Available: https://docs.gitlab.com/runner/ [Accessed Aug. 20, 2022].

[25] GitHub, Inc., "About self-hosted runners," *GitHub*. [Online]. Available: https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners [Accessed Aug. 20, 2022].

[26] HEPRC, *Cloudscheduler v2*. [Online]. Available: https://github.com/hep-gc/cloudscheduler [Accessed Aug. 20, 2022].

[27] Travis CI, *Travis CI*. [Online]. Available: https://www.travis-ci.com/ [Accessed Aug. 23, 2022].

[28] Travis CI, "About Travis CI," *Travis CI*. [Online]. Available: https://www.travis-ci.com/about-us/ [Accessed Aug. 20, 2022].

[29] Travis CI, "Travis CI Tutorial," *Travis CI*. [Online]. Available: https://docs.travis-ci.com/user/tutorial/ [Accessed Aug. 20, 2022].

[30] Travis CI, "Pricing — Travis CI," *Travis CI*. [Online]. Available: https://www.travis-ci.com/pricing/ [Accessed Aug. 20, 2022].

[31] The Botherders, *BuildBot*. [Online]. Available: https://buildbot.net/ [Accessed Aug. 23, 2022].

[32] Twisted Matrix Laboratories, *Twisted Python*. [Online]. Available: https://twisted.org/ [Accessed Aug. 20, 2022].

[33] The Botherders, "buildbot - PyPI," *BuildBot*. [Online]. Available: https://pypi.org/project/buildbot/ [Accessed Aug. 20, 2022].

[34] Jenkins, *Jenkins*. [Online]. Available: https://www.jenkins.io/ [Accessed Aug. 20, 2022].

[35] Eclipse Foundation, "About Jenkins," *Jenkins*, Feb. 2017. [Online]. Available: https://wiki.eclipse.org/index.php?title=Jenkins&oldid=414064 [Accessed Aug. 20, 2022].

[36] Apache, *Groovy*. [Online]. Available: https://groovy-lang.org/ [Accessed Aug. 20, 2022].

[37] Jenkins, *Junit*. [Online]. Available: https://plugins.jenkins.io/junit/ [Accessed Aug. 20, 2022].

[38] Holger Krekel, *pytest*. [Online]. Available: https://docs.pytest.org/en/7.1.x/ [Accessed Aug. 20, 2022].

[39] Leah Klearman, *pytest-rerunfailures*. [Online]. Available: https://github.com/pytest-dev/pytest-rerunfailures [Accessed Aug. 20, 2022].

[40] Oliver Gondža, *Jenkins – OpenStack Cloud*. [Online]. Available: https://plugins.jenkins.io/openstack-cloud/ [Accessed Aug. 20, 2022].

Appendix A CSV2 Deployment Pipeline

```
pipeline {
    agent { label 'ci-server' }

    stages {
        stage('Create and configure OpenStack VM') {
            steps {

                sh 'echo Creating new target instance'
                script {
                    def target = openstackMachine cloud: 'Beaver', template:
                    ↪  'csv2-target', scope: 'unlimited:'
                    echo 'Created new target ' + target.address

                    def host_number = target.address.tokenize('\\.').last()

                    echo 'Enabling root ssh'

                    withCredentials([sshUserPrivateKey(credentialsId: 'csv2_ci_target',
                    ↪  keyFileVariable: 'identity')]) {
                        def remote = [:]
                        remote.name = 'csv2-target'
                        remote.host = 'elephant' + host_number + '.heprc.uvic.ca'
                        remote.user = 'centos'
                        remote.identityFile = identity
                        remote.allowAnyHosts = true
                        remote.retryWaitSec = 30
                        remote.retryCount = 60
                        echo remote.toString()
                        sshCommand remote: remote, command: "sudo sed -ri
                        ↪  's/^.*((ecdsa|ssh)\\S*)/\\1/' /root/.ssh/authorized_keys"
                    }

                    echo 'Setting ci target'

                    def branch    = 'stable-2.10'
                    def db_backup = 'none'
                    def schema    = 'stable-2.10.3.rc1'

                    def default_pass = 'some_password'
                    def tester_pass  = 'some_password'
                    def other_pass   = 'some_password'

                    // Add to known hosts
                    sh 'ssh-keyscan -H ' + 'elephant' + host_number + '.heprc.uvic.ca' +
                    ↪  ' >> ~/.ssh/known_hosts'

                    def ansible_dir = '/opt/deployment/uvic-heprc-ansible-playbooks'
```

```
                    sh "sudo ${ansible_dir}/roles/csv2-ci/files/set_ci_target.sh
                    ↪  ${host_number} ${branch} ${db_backup} ${schema} ${default_pass}
                    ↪  ${tester_pass} ${other_pass}"


                }

            }
        }

        stage('Deploy csv2 to target') {
            steps {
                echo 'Beginning csv2 deployement'
                dir('/opt/deployment/uvic-heprc-ansible-playbooks') {
                    sh 'ansible-playbook -i inventory -u root main.yaml'
                }
            }
        }

    }

}
```

Appendix B Unit Test Pipeline

```
pipeline {
    agent { label 'ci-server' }

    stages {
        stage('Run tests') {
            steps {
                // Pull latest test code
                sh "sudo git --git-dir=/root/cloudscheduler/.git pull"

                 // Delete previous test results
                sh 'rm /var/lib/jenkins/workspace/unit-test/unit_results.xml | true'

                // Run unit tests
                dir('/root/cloudscheduler/unit_tests/') {
                    sh '''
                    export USER=tester
                    python3 run_tests -v alias cli cloud group job server user vm -x
↪   /var/lib/jenkins/workspace/unit-test/unit_results.xml
                    '''
                }
            }
        }
    }

    post {
        always {
            junit "unit_results.xml"
            sh "sudo git --git-dir=/root/cloudscheduler/.git reset --hard"
        }
    }
}
```

Appendix C Web Test Pipeline

```
pipeline {
    agent { label 'ci-server' }

    stages {
        stage('Run tests') {
            steps {
                // Pull latest test code
                sh "sudo git --git-dir=/root/cloudscheduler/.git pull"

                // Delete previous test results
                sh 'rm /var/lib/jenkins/workspace/web-test/web_results.xml | true'

                // Tests are selected using the -k flag, which pattern matches against
                 ↪  the test case names in the unit_tests/web_tests directory files.
                // All "Common" tests should not be run.
                // The --reruns sets the number of times to re-run a test before it is
                 ↪  considered to have failed.
                sh '''
                cd /root/cloudscheduler/unit_tests
                xvfb-run -a --server-args="-screen 0 1280x1024x24" python3 -m pytest
↪  web_tests/ --reruns 2 --ignore=web_tests/cloudscheduler -v -s -k "not Opera and not
↪  Chromium and not Common" --junitxml
↪  /var/lib/jenkins/workspace/web-test/web_results.xml
                '''
            }
        }
    }

    post {
        always {
            junit "web_results.xml"
            sh "sudo git --git-dir=/root/cloudscheduler/.git reset --hard"
        }
    }
}
```