Using Continous Deployment techniques to manage software change at a WLCG Tier-2

G Roy[1], **E Simili**[1], **G Stewart**[1], **S C Skipsey**[1] **and D Britton**[1]

[1] School of Physics and Astronomy, University of Glasgow, Kelvin Building, University Avenue, G12 8QQ, United Kingdom

E-mail: `gareth.roy@glasgow.ac.uk`

Abstract. Continuous Integration (CI) and Continuous Development (CD) are common techniques in software development. Continuous Integration is the practice of bringing together code from multiple developers into a single repository, while Continuous Development is the process by which new releases are automatically created and tested. CI/CD pipelines are available in popular automation tools such as GitLab, and act to enhance and accelerate the software development process. Continuous Deployment, in which automation is employed to push new software releases into the production environment, follows naturally from CI/CD, but is not as well established due to business and legal requirements. Such requirements do not exist in the Worldwide LHC Compute Gird (WLCG), making the use of continuous deployment to simplify the management of grid resources an attractive proposition. We have developed work presented previously on containerised worker node environments by introducing continuous deployment techniques and tooling, and show how these, in conjunction with CI/CD, can reduce the management burden at a WLCG Tier-2 resource. In particular, benefits include reduced downtime as a result of code changes and middleware updates.

1. Introduction

As budgets within the Worldwide LHC Compute Grid (WLCG) become more constrained due to a flat-cash funding environment sites are being forced to operate with reduced or limited manpower. At the same time these sites are required to support large and complex stacks of Grid Middleware essential for the daily operation of the WLCG experimental users. There is ongoing efforts within the WLCG to reduce the amount of software needed to configure and maintain a Grid site, with software such as CVMFS [9] used to simplify software distribution or Singularity [11] being used to normalise the production environment. Additionally more effort is being made to reuse industry standard tooling such as CEPH [5] for distributed storage, OpenStack [8] for resource provision, or through the use of commercial cloud resources to supply compute. There has also been work in novel methods for providing ephemeral resources such as VAC which is a VM lifecycle manager that implements the vacuum model on a group of autonomous worker nodes [10].

However, with nearly all these approaches System Administrators are still required to deploy, upgrade and patch the computing infrastructure regularly to ensure a safe and trusted environment. While configuration management tools such as Puppet [6] or Ansible [7] can simplify the process and ensure repeatability software updates can often lead to downtimes. To make sure systems are truly robust and minimise downtimes that may arise from failed software updates the setup and maintenance of staging or test systems is required, increasing the overall workload and cost of running these systems. Additionally as automated attacks on computational resources increase, System Administrators frequently have to upgrade and patch systems quickly sacrificing availability for safety.

An alternative to this approach is to leverage the now industry standard use of containers and couple this with the use of modern Continuous Integration (CI) and Continuous Development (CD) techniques commonly found in the software development lifecycle. Continuous Integration is the practice of bringing together code from multiple developers into a single repository, while Continuous Development is the process by which new releases are automatically created and tested. Continuous Deployment, in which automation is employed to push new software releases into the production environment, follows naturally from CI/CD, but is not as well established in industry due to business requirements where features are released on a fixed, controlled schedule to customers, or legal requirements where software releases may need to be verified or audited (potentially by a third party) for security compliance before release. Such requirements do not exist in the WLCG, making the use of continuous deployment to simplify the management of grid resources an attractive proposition.

We have developed work presented previously [1] on containerised worker node environments by introducing continuous deployment techniques and tooling, and show how these, in conjunction with CI/CD, can reduce the management burden at a WLCG Tier-2 resource.

2. Continuous Deployment Workflow

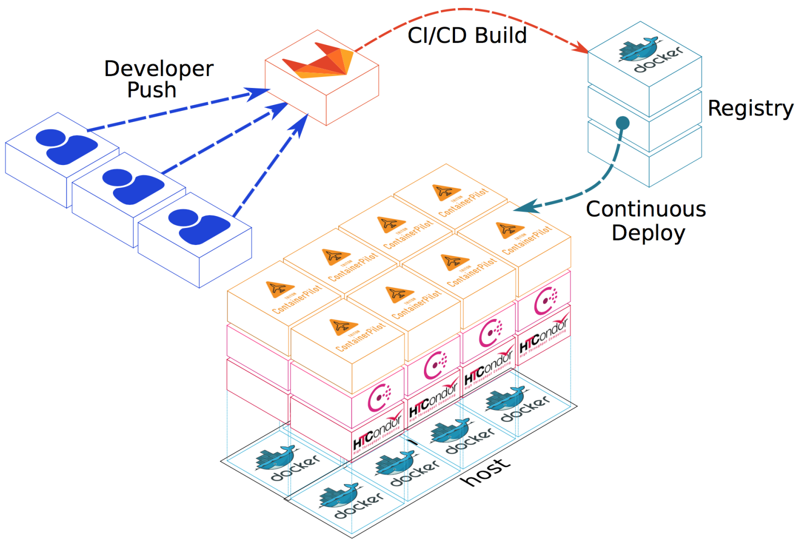


Figure 1: Schematic of a continuous deployment work flow based on containers.

The containerised worker node (CWN) that was developed previously [1] has been used as a representative software application for our continuous deployment workflow. In that previous work a container was developed that contained an HTCondor [4] worker node that communicated with the existing batch farm running at the Uki-Scotgrid-Glasgow site to run ATLAS payloads. The continuous deployment workflow proposed in this paper uses containers to snapshot particular, tested combinations of software releases along with their associated configuration. These snapshots can then be used as a single, atomic unit for deployment simplifying the upgrade procedure. The workflow itself can be split into three parts:

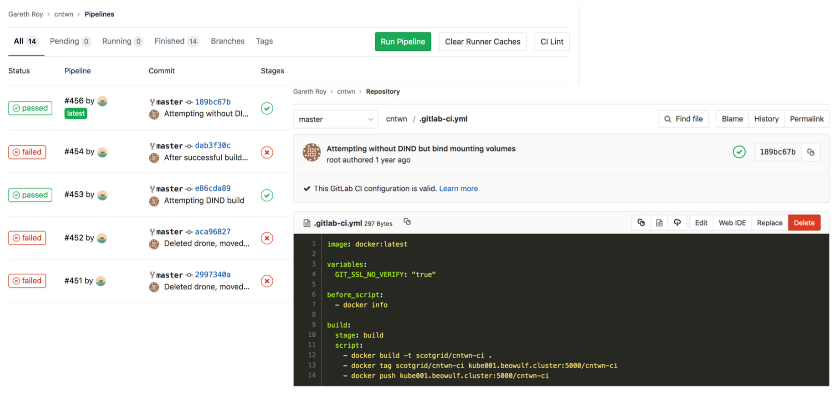
(i) Developers or System Administrators update configurations based on requirements.

(ii) Containers are automatically built, tested and pushed to a container registry ready for deployment to production.

(iii) Containers are deployed to execution hosts, replacing existing, older instances where necessary.

A schematic showing how the components interact is show in Figure 1. In this example Developers (or System Administrators) push changes into a GitLab [2] instance. This instance is used to implement the second part of our continuous deployment workflow by automatically building the container and running tests as soon as any changes take place in the master branch of a repository. If tests pass the containers are then pushed into a container registry (provided as part of the GitLab instance) for execution on a worker node. On each worker node host containers are run based on the images held within that registry, before a new container is instantiated on the host the registry is checked for a newer version and if one exists it is pulled to the host and all subsequent containers are started with the newer image.

3. GitLab-based CI/CD



Status	Pipeline	Commit	Stages
passed	#456 by latest	master 189bc67b Attempting without DI...	✓
failed	#454 by	master dab3f30c After successful build...	✗
passed	#453 by	master e86cda89 Attempting DIND build	✓
failed	#452 by	master aca96827 Deleted drone, moved...	✗
failed	#451 by	master 2997340a Deleted drone, moved...	✗

Attempting without DIND but bind mounting volumes
root authored 1 year ago
189bc67b

✓ This GitLab CI configuration is valid. Learn more

```
image: docker:latest

variables:
  GIT_SSL_NO_VERIFY: "true"

before_script:
  - docker info

build:
  stage: build
  script:
    - docker build -t scotgrid/cntwn-ci .
    - docker tag scotgrid/cntwn-ci kube001.beowulf.cluster:5000/cntwn-ci
    - docker push kube001.beowulf.cluster:5000/cntwn-ci
```

Figure 2: Output and configuration of a GitLab CI/CD Pipeline

As described in the previous section, in our example workflow production versions of the containerised worker node are built using a GitLab CI/CD pipeline. When changes are made to the master branch of the repository, a new container is automatically built and pushed to a self-hosted container registry. Figure 2 shows an example of a pipeline automatically building a worker node container.

In our workflow a feature-branch model is employed, in which new features are added as new branches in the repository; these branches can then be tested in isolation through manual or automated testing. Once the feature-branch has been shown to be working correctly, it can be merged into the master branch, at which point the pipeline is activated and the new container is produced. In the future, it is planned that changes to feature-branches will also trigger the production of a container. These can then be used in automated "canary" deployments, allowing unattended live testing before the feature is rolled into production at scale. Additionally, if a problematic deployment made it to production a particular branch could be reverted and a new container generated. This in conjunction with the process of updating containers described in the next section would allow the automated correction of any problematic deployment.

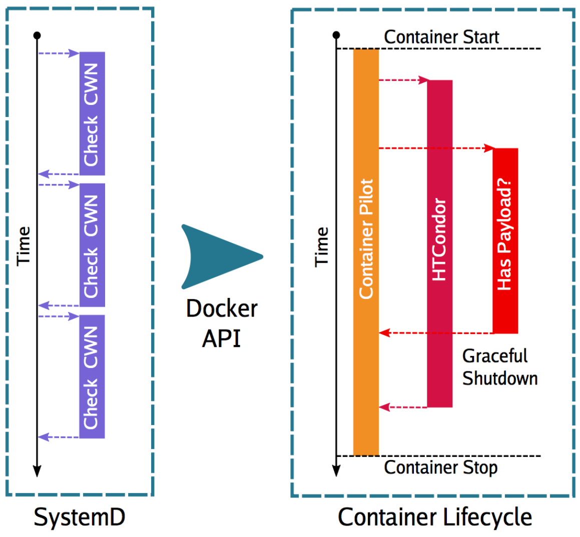


Figure 3: Schematic showing the lifetime of a self terminating container, and operation of the watchdog script.

4. Self-termintaing container

In order to ensure that containers are constantly updated, each container is configured with a specific lifetime, after which the containers self terminate with a watchdog script restarting and checking for updated containers to pull. The overall operation and lifetime of the container and watchdog script is shown in Figure 3.

In this work ContainerPilot [3] is used as an initialisation mechanism within the running container in order to create the self-terminating HTCondor worker nodes. The JSON listing shown in Figure 4 describes two jobs run by ContainerPilot. The first starts a Condor master daemon which connects to the local batch farm. The second is only triggered after the first job is determined to be healthy, and waits 30 minutes before forcing a peaceful shutdown of the Condor startd daemon, allowing the job to complete before all the daemons terminate. Along with self-termination ContainerPilot is also used to register the container with a local information system for monitoring purposes and in future it is planned to package up logging information and send it to a central system. It is still possible to get self-termination by only using Condor within the container. This is achieved by setting the `STARTD_NOCLAIM_SHUTDOWN` parameter to terminate the startd daemon if no job was present after some period.

Along with the self-terminating container a simple watchdog script, built as a SystemD timer, runs outside the container. This script checks to see which containers are running and, if a worker node container is found to be missing, pulls the latest version of the container from the registry and starts a new instance. This simple mechanism ensures that as each container terminates it is replaced by the most up to date version. This allows the almost seamless upgrade of worker nodes within the cluster without the need to drain the system as the lifetime of each container is only that of the running job.

5. Conclusions

In this paper we have outlined an example continuous deployment workflow for a WLCG Tier-2 based on a containerised worker node. We have described how changes to configuration information can trigger an automated building process using GitLab provided CI/CD tooling. How this build process can push a new container to a central registry and how this coupled with

```
jobs: [
   {
       name: "condor-master",
       exec: "condor_master -f",
       port: 9621,
       health: {
           exec: "condor_config_val -startd StartJobs",
           interval: "60",
           ttl: "120",
           timeout: "60",
  },
  {
           name: "has-payload"
           exec: "condor_off -daemon startd -peaceful
           when:
                 source: "condor-master",
                 once: "healthy",
                 interval: "30m",
   }
  },]
```

Figure 4: JSON configuration for ContainerPilot showing the healthcheck and self-termination stanza.

a lifetime and ability to self-termination can lead to a constant update process across the cluster without the need to drain systems for rebuilding.

In the future it is hoped to extend this work to include the automated monitoring and collection of logs from within each container, the exploration of different batch systems (or other novel methods of resource allocation), and it is hoped to extend the model of a fixed lifetime and a mechanism for self-termination to other services such as a CE.

References

[1] *G. Roy et al.*, 2018 J. Phys.: Conf. Ser. 1085 032026, "A container model for resource provision at a WLCG Tier-2"
[2] *giltlab.com*, "Gitlab CE" [software], version 11.6.5, 2019. Available from https://www.gitlab.com [accessed 29 May 2019]
[3] *ContainerPilot*, "ContainerPilot" [software], version 2.7.8, 2017. Available from https://www.joyent.com/containerpilot [accessed 29 May 2019]
[4] *Michael Litzkow, Miron Livny, and Matt Mutka, "Condor - A Hunter of Idle Workstations", Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.*
[5] CEPH, "CEPH" [software], version Mimic, 2019. Available from https://ceph.com/ [accessed 29 May 2019]
[6] Puppet, "Puppet" [software], version 6.4.2, 2019. Available from https://puppet.com/ [accessed 29 May 2019]
[7] Ansible, "Ansible" [software], version 2.4, 2019. Available from https://www.ansible.com/ [accessed 29 May 2019]
[8] OpenStack, "Openstack" [software], version 14 Stein, 2019. Available from https://www.openstack.com/ [accessed 29 May 2019]
[9] J Blomer et al.; 2011 J. Phys.: Conf. Ser. 331 042003, "Distributing LHC application software and conditions databases using the CernVM file system"
[10] A McNab; "Running jobs in the vacuum" (A McNab et al 2014 J. Phys.: Conf. Ser. 513 032065)
[11] Singularity, "singularity" [software], version 2.3.2-dist, 2017. Available from http://singularity.lbl.gov/ [accessed 18 Oct 2017]