Optimizing Provisioning of LCG Software Stacks with Kubernetes

Richard Bachmann[1,2,*], *Gerardo* Ganis[1,**], and *Dmitri* Konstantinov[3,]*Ivan* Razumov[3,]*Johannes Martin* Heinz[4,]

[1]CERN, Experimental Physics Department, 1211 Geneva 23, Switzerland

[2]NTNU, Department of Computer Science, NO-7491 Trondheim, Norway

[3]NRC Kurchatov Institute - IHEP, Akademika Kurchatova pl. 1, 123182 Moscow, Russia

[4]Karlsruhe University of Applied Sciences, Moltkestr. 30, 76133 Karlsruhe, Germany

Abstract. The building, testing and deployment of coherent large software stacks is very challenging, in particular when they consist of the diverse set of packages required by the LHC[***] experiments, the CERN Beams department and data analysis services such as SWAN. These software stacks comprise a large number of packages (Monte Carlo generators, machine learning tools, Python modules, HEP[****] specific software), all available for several compilers, operating systems and hardware architectures. Along with several releases per year, development builds are provided each night to allow for quick updates and testing of development versions of packages such as ROOT, Geant4, etc. It also provides the possibility to test new compilers and new configurations.

Timely provisioning of these development and release stacks requires a large amount of computing resources. A dedicated infrastructure, based on the Jenkins continuous integration system, has been developed to this purpose. Resources are taken from the CERN OpenStack cloud; Puppet configurations are used to control the environment on virtual machines, which are either used directly as resource nodes or as hosts for Docker containers. Containers are used more and more to optimize the usage of our resources and ensure a consistent build environment while providing quick access to new Linux flavours and specific configurations.

In order to add build resources on demand more easily, we investigated the integration of a CERN provided Kubernetes cluster into the existing infrastructure. In this contribution we present the status of this prototype, focusing on the new challenges faced, such as the integration of these ephemeral build nodes into CERN's IT infrastructure, job priority control, and debugging of job failures.

[*]e-mail: richard.bachmann@cern.ch
[**]e-mail: gerardo.ganis@cern.ch
[***]Large Hadron Collider
[****]High Energy Physics

1 Introduction

The *Software Process Integration* (SPI) team, which is part of CERN's *Experimental Physics - Software* group, provides a comprehensive collection of software for scientific computing. This collection is known as the *LCG*[1] *software stacks*[1]. A sizable amount of time and computing resources are used to maintain, update and expand these stacks. This, combined with a growing scope, creates a need to make the process more efficient. Data collected from the current build system indicates that a significant portion of the available computing resources, such as CPU time, memory and storage, is not actively utilized. An example of the observed CPU use can be seen figure 1. The ongoing deployment of *Kubernetes*[2] at CERN presents an opportunity to resolve this issue.

In this paper we present the results of our investigations of ways to use Kubernetes in order to provide the LCG software stacks. This paper is organized as follows: In this section we summarize the main aspects of the LCG software stacks and the Kubernetes system. In section 2 the requirements for the new system are laid out. Section 3 lists approaches and tools for building this system with Kubernetes, as well as their perceived advantages and disadvantages. Finally section 4 presents the project's current status and future work.

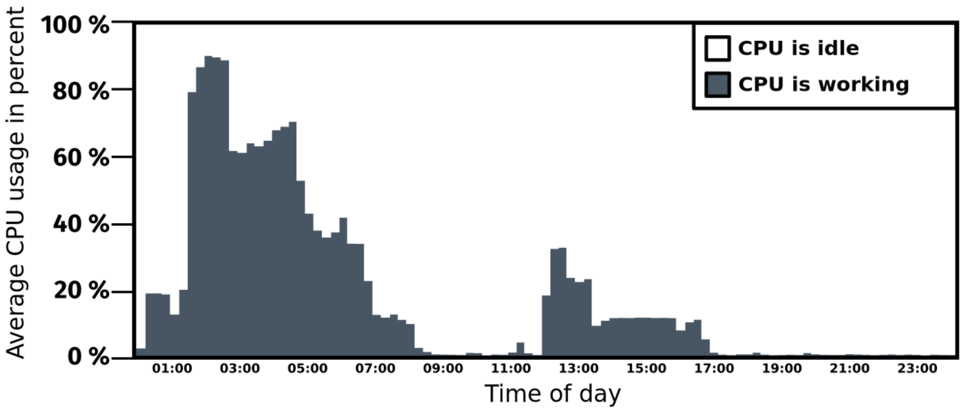

Figure 1. The average CPU utilization of build nodes active during the nightly build process. After the initial peak most of the compute resources go unused, and could be made available to other processes or user groups.

1.1 The LCG Software stacks

The LCG software stacks contain almost 450 packages compiled with several compilers, and made available for multiple operating systems, Python versions and hardware architectures. Among these packages are Monte Carlo generators, machine learning tools, Python modules and HEP-specific software. Along with several releases per year, about 30 development builds are provided each night to allow for quick updates and testing of new versions of ROOT[3], Geant4[4], etc. The system also provides the possibility to test new compilers and configurations. The two experiments ATLAS[5] and LHCb[6], as well as the SWAN[7] platform and the Beams Department[8] are the largest users at CERN. Additionally, the stacks have a number of users outside the listed projects and even outside of the HEP community.

[1]LHC Computing Grid

1.1.1 Build process

Figure 2 outlines the different stages of the build, testing and distribution process which creates the LCG stacks. The process is currently automated using a Jenkins[9] continuous integration server [2]. This server delegates most of the computation work to a number of different build workers, most of which are virtual machines. These virtual machines, in turn, are provided by the CERN OpenStack[11] cloud. The work of transitioning to a fully containerized workflow is already well underway, and the majority of the builds now run in containers.

A standard build interacts with a number of external services in order to make the output available to users. Among these are:

- **Kerberos**[12], which provides the means of secure authentication.
- **EOS**[13], a disk-based low-latency storage service.
- **CVMFS**[14], the primary distribution platform.
- **CDASH**[15], which displays data about the overall health of the system.

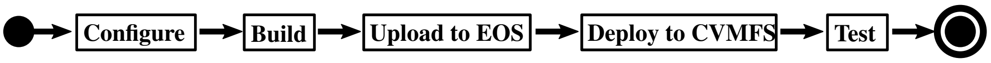

Figure 2. A typical workflow of an LCG stack build pipeline contains these steps.

1.2 Kubernetes

Kubernetes is an open-source container orchestration system, designed to enhance service scalability by replicating individual components. It features a detailed REST API, which is used to interact with the system. Kubernetes operates on *pods*, which function as the smallest schedulable unit. A pod is a group of one or more co-scheduled containers with shared storage and networking. These are scheduled to run on worker *nodes*, which can be either physical or virtual machines. In general, pods can be launched on any available node which fulfills a set of requirements, such as the amount of available resources. This makes the process of adding computing power to the whole system as simple as adding new nodes to the cluster.

2 Vision for a new build infrastructure

The Kubernetes-based build system will have to work in unison with existing systems, in addition to following new design principles. The most important top level requirement is for it to function with Jenkins as the controlling interface and agent, since Jenkins is used to operate all existing build processes. This means that it must allow for Jenkins to trigger build jobs, manage and supply configuration options and credentials, and also view the result of each job. If possible, rewrites of existing scripts should be kept to a minimum.

In order to execute the build jobs, as shown in figure 2, the solution must be able to run the project's CMake[16]-based build system *LCGCMake*[1] and supply it with the environment variables which define each build. The resulting artifacts must then be uploaded to EOS and deployed to CVMFS, which requires credentials and further configuration options. Both the build- and test-stage interact with CDash, which is used to monitor the status of the builds.

[2]In time Gitlab CI/CD [10] may replace Jenkins as the main tool for running the build processes, but the investigation of this possibility will be conducted independently of this project.

Furthermore it is seen as desirable to build the solution as a combination of purpose-specific components. Ideally each stage seen in figure 2 should be an independent container which can be replaced without affecting the others. In order to achieve this, the solution should follow Redhat's *Principles of Container-based Application Design*[17] to the degree which other constraints allow. Containers of the same stage should also be prevented from affecting each other, ensuring that each job is executed independently.

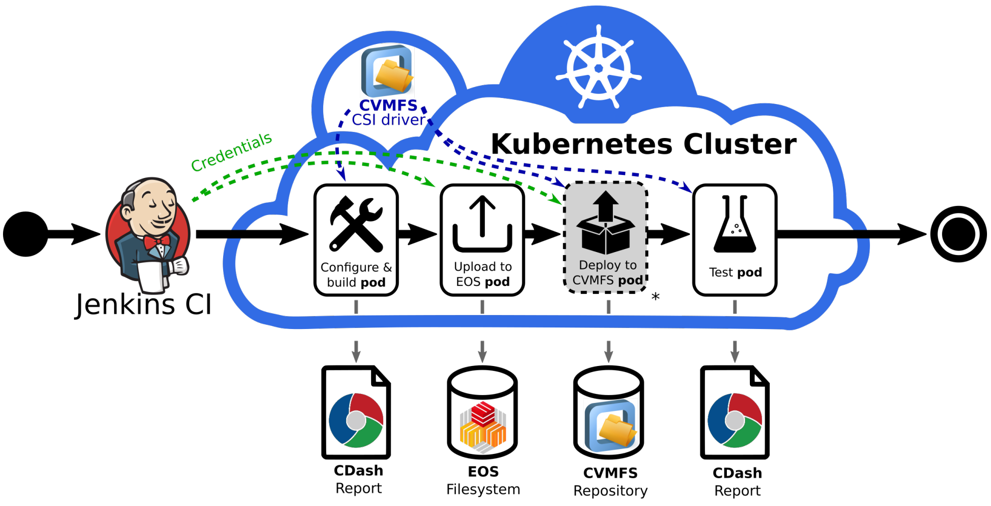

Figure 3. Interactions of the Kubernetes solution with different system components. The Configure step is at the time of writing not complex enough to warrant a separate pod, but this may change in the future.

2.1 Kubernetes as a batch system

Kubernetes is traditionally known as a platform for running long-lived services in a scalable and fault tolerant manner. The LCG stack use case differs from this model, as build jobs are ephemeral, usually lasting only a handful of hours. However, Kubernetes offers a number of mechanisms which are of particular interest for the realization of the project:

- The Kubernetes *batch job* is a concept which wraps a pod in additional logic, granting the ability to control job timeout, number of required completions, etc.[18]. These are designed for short-lived actions, not too different from the LCG stack build jobs.
- Kubernetes' *secret* objects allow for the secure management of credentials, access tokens and similar sensitive resources [19]. In the absence of a Jenkins client running on the pod, this system can be used to supply pods with sensitive data without having to include it in the container image or pod definition. A mechanism for synchronizing the Kubernetes secrets with the Jenkins secrets will still have to be created.
- *Auto-scaling* allows Kubernetes clusters to adapt to varying workloads and sudden peaks of resource demand. Different ways of scaling exist, but *cluster auto-scaling* is of particular interest for this use case. This feature, which is implemented on Kubernetes and the underlying provider OpenStack, allows for the automatic spawning and deletion of worker nodes based on demand. Nodes are added to the cluster when the sum of required resources exceeds what currently exists in the cluster. Likewise, nodes can be deleted when resources

go unused for a period of time. Worker nodes are returned to a shared resource pool when deleted, allowing others to make use of them. At this point in time other internal projects, as well as the ROOT and Geant4 development teams, share this resource pool. In the future, both pool size and demand may increase with the maturation of Kubernetes use at CERN. With the adoption of auto-scaling computing resources, a noticeable flattening of the graph in figure 1 is expected.

2.1.1 Known limitations

- The LCG Stacks are currently built for a set of x86_64 Linux platforms, as well as MacOS and ARM64. Of these only x86_64 Linux is officially supported. For now MacOS and ARM64 would therefore have to be built as before.
- Managed Kubernetes clusters are at the time of writing not offered, which means that its operation and maintenance represents an additional burden for the SPI team. This might, however, change in the future.

3 Evaluation and comparison of potential solutions

3.1 JenkinsCI Kubernetes Plugin

The first prototype was developed using the *Jenkins plugin for Kubernetes*[20]. While initially promising it proved to lack development in certain areas which are needed for this use case. For instance, the *container statement*, which allows for the execution of commands within specific containers, was marked as an alpha feature at the time of testing and did not function as expected. The fact that the plugin requires all scripts to be rewritten to Jenkins' pipeline syntax represents another major roadblock.

3.2 Working with the Kubernetes API

As each Kubernetes cluster exposes an intricate and powerful API to the web it can be used as a more direct way to interact with its resources. To manage the complexity of the needed requests two methods of interaction have been evaluated:

3.2.1 Scripted kubectl

Kubectl[21] is the primary command-line tool used to interact with clusters. It is used primarily for development and manual operations on a cluster. Nonetheless kubectl can also be used in scripts to perform the desired actions in an automated fashion. Such a solution, however, would require a complete rewrite of the existing scripts. These rewrites would also carry an additional burden of introduced complexity, making the code difficult to maintain.

3.2.2 REST API Wrapper

Instead of making the jobs directly interact with the cluster, an intermediate system could be introduced between these components. This system can take the shape of a feature-reduced API exposed to Jenkins, a set of project-specific logic and system to communicate with the cluster. An in-house developed prototype based on this concept, called *Lodesman*[22], was created, but has been put aside in favour of a more mature platform. The latter is described in section 4.

3.3 Others

In addition to the approaches mentioned so far, the technologies shown in table 1 were briefly examined to see whether or not they fulfill the group's requirements. Note that HTCondor is a system which is unrelated to Kubernetes.

System	Assessment
HTCondor[23]	Running jobs may be terminated, resulting in loss of build artifacts. The average build size makes this highly undesirable. Queued jobs may have unpredictable long wait times, which some users counter by reserving resources in advance. The former is incompatible with the regularity expected by users who rely on the builds for their own pipelines, the latter results in inefficiencies similar to, or worse, than what is experienced now.
Kube-batch[24]	Different focus: AI/ML, BigData, HPC. Does not solve Jenkins integration.
Jenkins-x[25]	Focused on microservice development. The opinionated approach could be limiting.

Table 1. Other technologies which were briefly explored. These are deemed to be less desirable than the approaches presented so far.

4 Status and future work

At the time of writing the possibility of using *Argo*[26] to implement the pipelines is being explored. For the purpose of this project Argo provides two particularly useful toolsets: *Workflows*[27] and *Events*[28].

- Workflows describe an ordered list of process steps, which can be used to describe the builds similarly to what is shown in figure 3. Individual workflow stages and their interactions can be specified using YAML files. Each stage represents an individual pod. Argo extends workflow YAML syntax with parameters, loops and more, allowing the expression of certain logic without having to resort to developing custom applications for this purpose.
- The Events dependency manager provides functionality for webhooks and logic systems which can trigger these workflows. This can be used to create an interface for Jenkins.

A prototype has been set up: It is able to perform a standard nightly build, producing the desired artifacts and uploading a report to the monitoring tool CDash.

The technical challenges we currently face mostly concern the movement of data between pods, due to the large size of build artifacts. Argo handles the transfer of control of *persistent volumes*[29] by use of *persistent volume claims* (PVC). Persistent volumes are abstractions of physical storage, which can be mounted within the file system of pods. Claims can be used to dynamically provision volumes from OpenStack, but a system for the management of the PVCs still has to be found. The number of PVCs should scale to the number of needed concurrent workflows and must be constrained by a potentially varying storage quota. A system for long term storage of specific files, such as logs, has to be integrated as well.

The interface with external storage is also being actively worked on. We are at this point in time evaluating the solutions provided by CERN IT for the purpose of interfacing with EOS and CVMFS storage systems via *CSI drivers*[30]. As a potential alternative to EOS we are also considering OpenStack object storage using the S3 protocol, which has the advantage of already being integrated in Argo.

References

[1] J. C. Villanueva. "*Building, testing and distributing common software for the LHC experiments*". In: EPJ WoC **214**.05020 (2019).

[2] *Kubernetes*. Kubernetes. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 01/15/2019).

[3] R. Brun & F. Rademakers. "*ROOT - An Object Oriented Data Analysis Framework*". In: Nucl. Inst. & Meth. in Phys. Res. A **389** (1997), pp. 81–86. URL: https://root.cern.ch/.

[4] S. Agostinell et al. "*GEANT4-a simulation toolkit*". In: Nucl. Inst. & Meth. in Phys. Res. A 506 (July 2003), p. 250.

[5] G. Aad et al. "*The ATLAS experiment at the CERN large hadron collider*". In: *Journal of Instrumentation* **3** (Aug. 2008), S08003.

[6] *The LHCb experiment*. CERN. 2019. URL: http://lhcb.web.cern.ch/lhcb/ (visited on 02/11/2019).

[7] D. Piparo et al. "*SWAN: a Service for Interactive Analysis in the Cloud*". In: Future Gener. Comput. Syst. 78 (2018), pp. 1071–1078.

[8] *The Beams Department*. CERN. 2019. URL: https://beams.web.cern.ch/ (visited on 02/11/2019).

[9] *Jenkins open source automation server*. 2019. URL: https://jenkins.io/doc/#what-is-jenkins (visited on 12/18/2019).

[10] *Gitlab CI/CD*. 2020. URL: https://docs.gitlab.com/ee/ci/ (visited on 07/02/2020).

[11] *CERN OpenStack service*. CERN. 2019. URL: https://clouddocs.web.cern.ch/ (visited on 12/17/2019).

[12] *Kerberos*. 2019. URL: https://web.mit.edu/kerberos/#what_is (visited on 01/15/2019).

[13] *EOS storage*. 2019. URL: https://eos-docs.web.cern.ch/eos-docs/ (visited on 01/15/2019).

[14] Jakob Blomer et al. "*New directions in the CernVM file system*". In: Journal of Physics: Conference Series **898** (Oct. 2017), p. 062031.

[15] *CDash testing server*. Kitware. 2019. URL: https://www.cdash.org/ (visited on 12/18/2019).

[16] *CMake build process management system*. 2020. URL: https://cmake.org/overview/ (visited on 07/08/2020).

[17] Bilgin Ibryam. *Principles of Container-based Application Design*. Redhat. 2017. URL: https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf (visited on 01/15/2019).

[18] *Kubernetes batch jobs*. Kubernetes. 2019. URL: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/ (visited on 01/15/2019).

[19] *Kubernetes secrets*. Kubernetes. 2019. URL: https://kubernetes.io/docs/concepts/configuration/secret/ (visited on 01/15/2019).

[20] *Kubernetes Continuous Deploy Plugin*. Jenkins. 2019. URL: https://github.com/jenkinsci/kubernetes-plugin (visited on 12/18/2019).

[21] *Kubectl*. Kubernetes. 2019. URL: https://kubernetes.io/docs/reference/kubectl/overview/ (visited on 01/15/2019).

[22] *Lodesman project repository*. 2019. URL: https://gitlab.cern.ch/rbachman/lodesman (visited on 01/15/2019).

[23] *HTCondor*. 2019. URL: https://research.cs.wisc.edu/htcondor/description.html (visited on 01/15/2019).

[24] *kube-batch*. 2019. URL: https://github.com/kubernetes-sigs/kube-batch (visited on 12/18/2019).

[25] *Jenkins X*. 2019. URL: https://jenkins-x.io/docs/overview/ (visited on 12/18/2019).

[26] *Argo project*. 2019. URL: https://argoproj.github.io/ (visited on 03/09/2020).

[27] *Argo Workflows*. 2019. URL: https://argoproj.github.io/argo/ (visited on 03/09/2020).

[28] *Argo Events*. 2019. URL: https://argoproj.github.io/projects/argo-events (visited on 03/09/2020).

[29] *Kubernetes Persistent Volumes*. Kubernetes. 2019. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (visited on 02/10/2019).

[30] *Container Storage Interface*. Kubernetes. 2019. URL: https://kubernetes-csi.github.io/docs/#kubernetes-container-storage-interface-csi-documentation (visited on 02/11/2019).