# Optimizing Provisioning of LCG Software Stacks with Kubernetes

**Authors:** J Heinz[1,3], R Bachmann[2,3], G Ganis[3], D Konstantinov[4], I Razumov[4]
[1]Karlsruhe University of Applied Sciences [2]NTNU [3]CERN [4]NRC Kurchatov Institute - IHEP

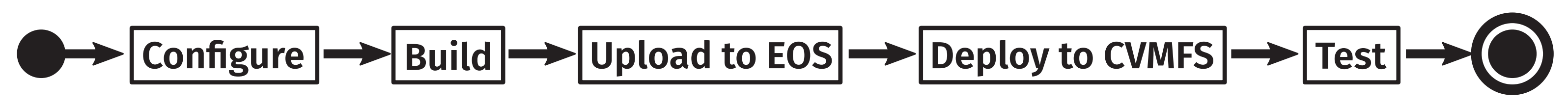
**Email:** ep-sft-spi@cern.ch

## The LCG software stacks

The LCG software stacks[1] contain almost 450 packages available for several compilers, operating systems, Python versions and hardware architectures. Among these packages are Monte Carlo generators, machine learning tools, Python modules and HEP specific software. Some of its users are:

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, Geant4, etc. It also provides the possibility to test new compilers and configurations.

The typical workflow of a LCG stack build pipeline contains the following steps:

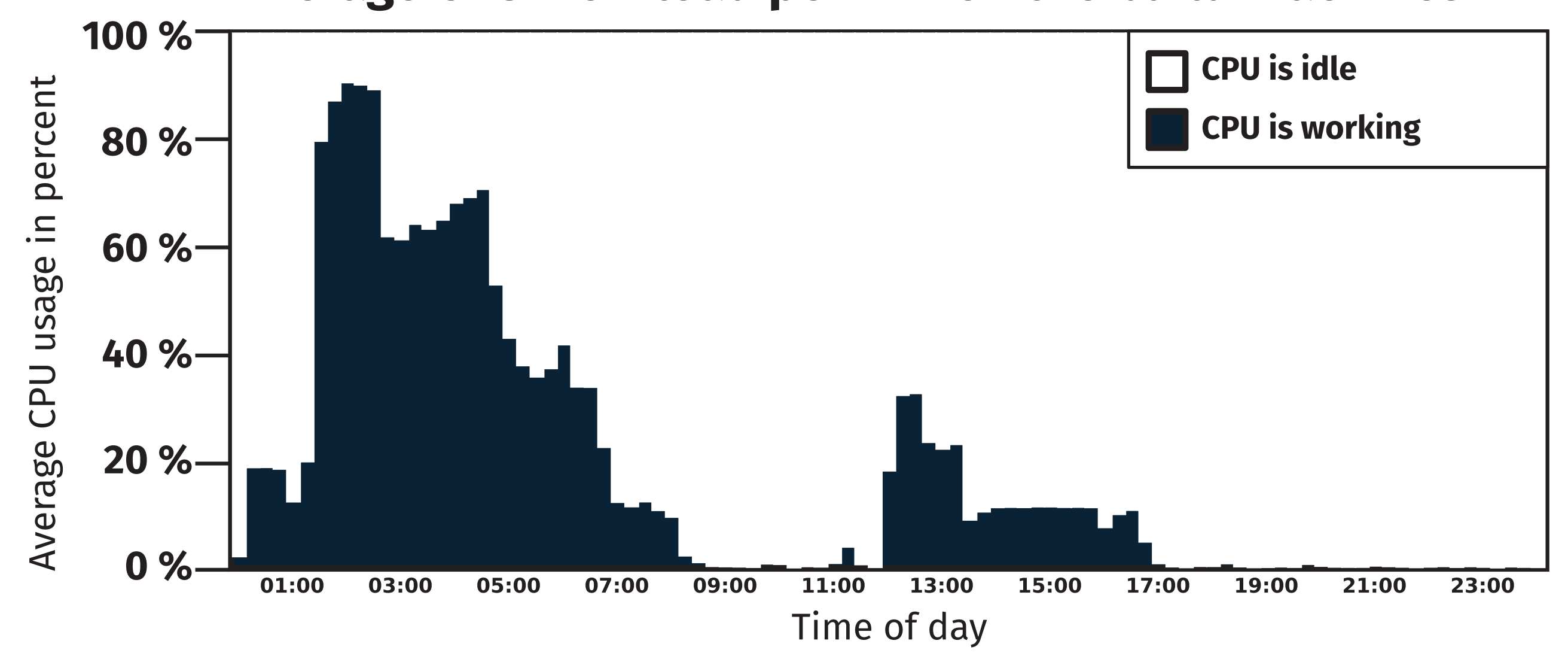


This process is currently automated using a ***Jenkins*** continous integration server and virtual machines that function as build slaves for the Jenkins master. These machines are taken from ***OpenStack***, CERN's in-house cloud provider[2]. They run CERN CentOS 7 which is configured using ***Puppet*** to provide a Docker daemon for the build process as well as ***Kerberos*** and ***CVMFS*** integration. The CVMFS file system and the Kerberos keytab are bind-mounted into the Docker build containers.

## Saving resources through container orchestration

The goal of moving from VMs to Kubernetes is to ***increase the efficiency of resource usage*** and better respond to ***peaks*** of resource demand. This can be realized by the cluster auto-scaling feature of Kubernetes that can automatically spawn and delete worker nodes as needed. The current CPU usage is displayed in the plot below:

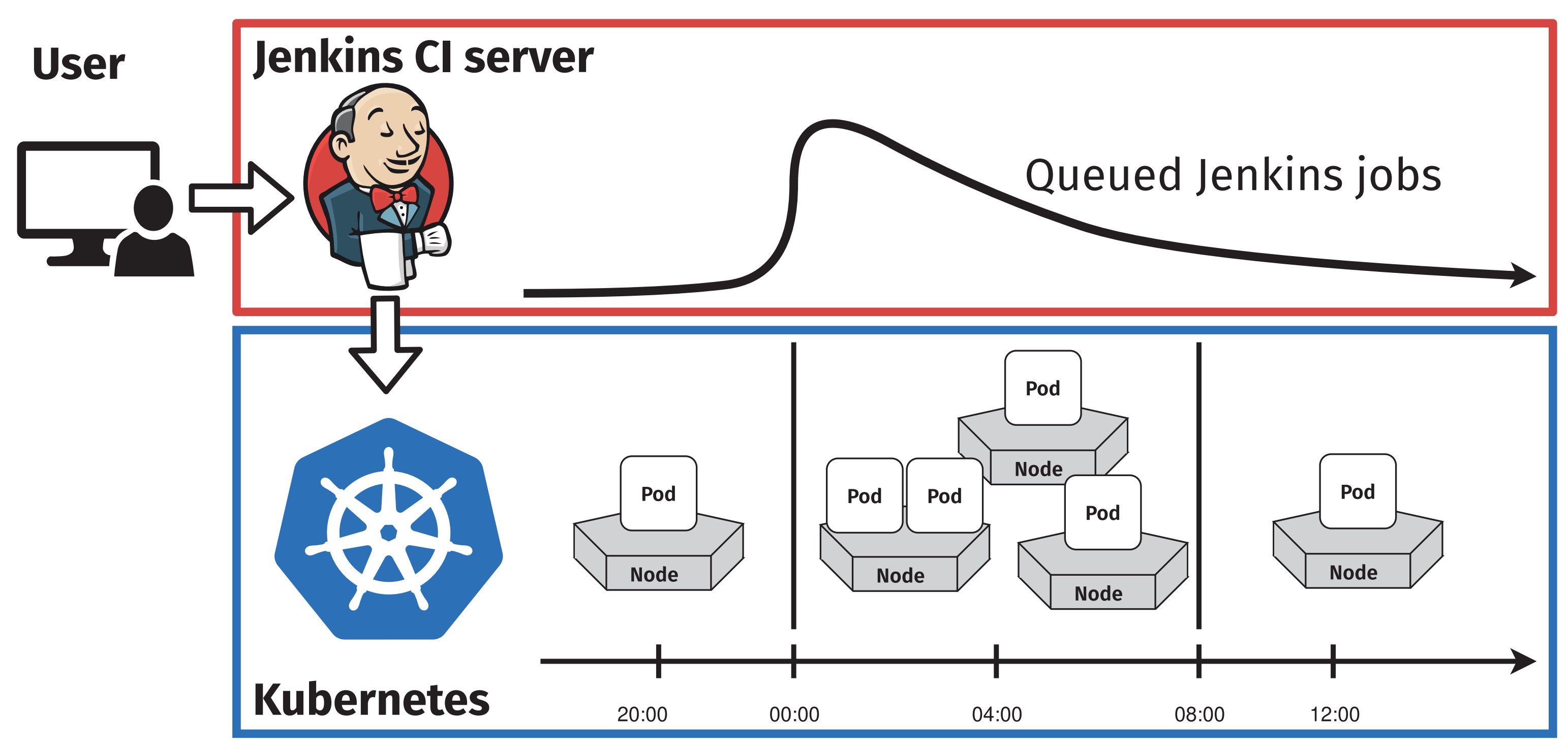
**Average CPU workload per 24h on the build machines**



Kubernetes also serves as an additional ***abstraction layer*** that takes care of the underlying operating system, scheduling and cluster management. A user of Kubernetes only needs to provide a suitable container image which is launched inside a ***pod***. A pod is a group of one or more ***co-scheduled*** containers with shared storage and network.

## Vision for a new build infrastructure

- Keep a Jenkins CI server as a single point of entry to trigger build jobs, manage configurations via variables and store all necessary secrets such as passwords.
- Use Kubernetes like a batch system similar to HTCondor to provide the necessary resources on demand and scale down after the builds are done

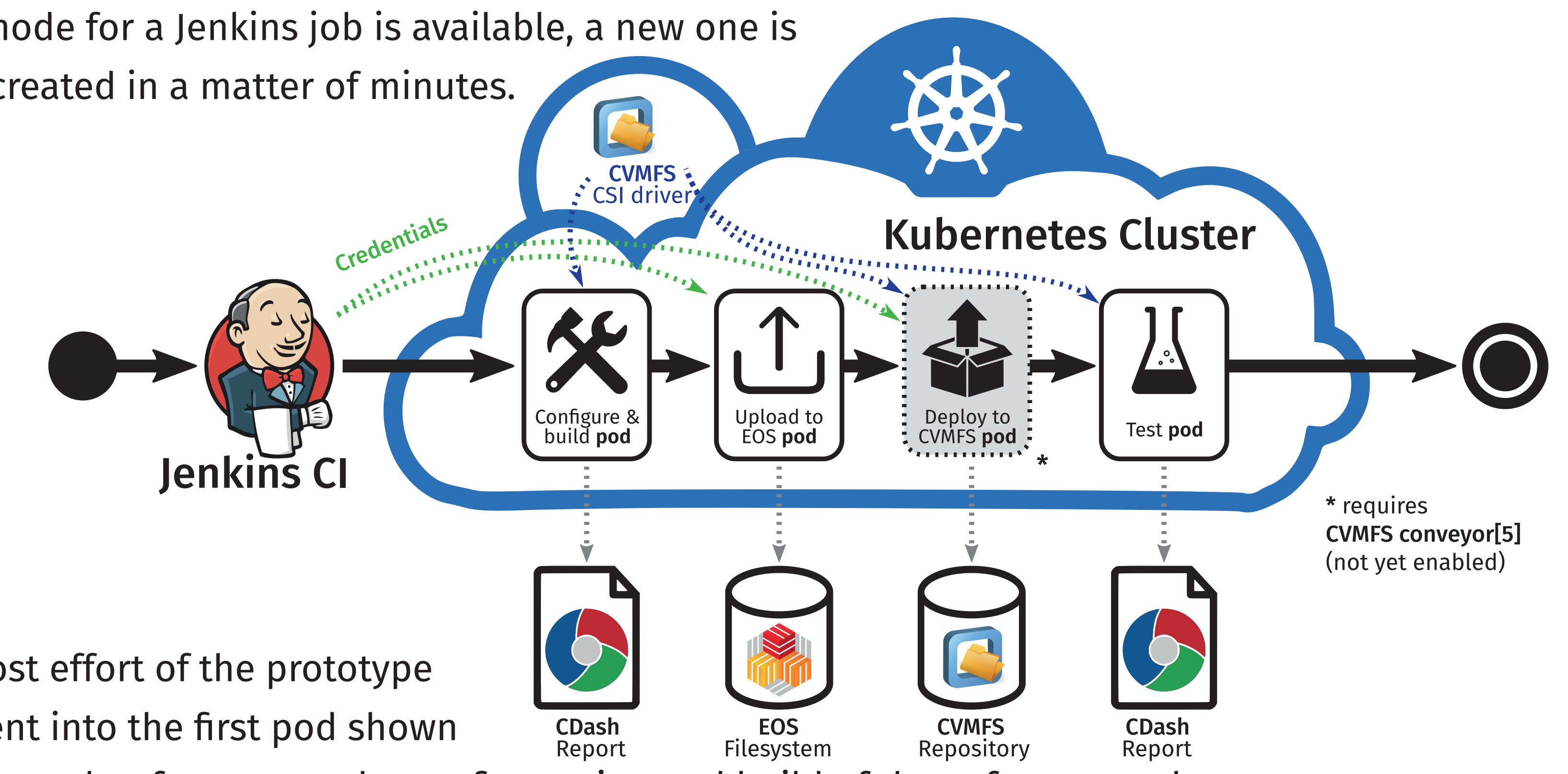


- Move environment definitions from the Docker host system inside the container images, for example the Kerberos authentication.
- Follow Red Hat's ***single concern principle***[3] to use specialized container images for building, testing and deployment instead of general purpose images

## State of the prototype

**CERN Infrastructure**

To realize the prototype we use the CERN cloud container service that relies on on ***OpenStack Magnum***[2]. CVMFS read access is needed for most pods to make incremental builds and tests possible. This feature is implemented by the ***CernVM-FS CSI driver***[4]. ***Auto-scaling*** of the cluster works within the quota of the OpenStack project. If no free node for a Jenkins job is available, a new one is created in a matter of minutes.



Most effort of the prototype went into the first pod shown above that focuses on the configurration and build of the software stack.

**Jenkins integration**

The integration into the Jenkins CI landscape is the defining challenge of this prototype because of the need to adapt to complex pre-existing workflows. This includes the configuration of jobs, getting their log files and sending Kerberos tickets.

There are two different approaches to solve this problem:

| | Kubernetes plugin for Jenkins | Kubernetes API (HTTP or kubectl) |
|---|---|---|
| **Jenkins integration** | Integrates with existing Jenkins instance ⊕ | Requires mechanism for returning status and logs to Jenkins synchronously ⊖ |
| **Ease of use** | Requires definition of Groovy pipeline ⊖ | Can be used from any shell environment ⊕ |
| **Development workload** | Dependence on external project and developers ± | Full control but increased internal development workload ± |

## Experience with the prototype

The software build inside the first pod, as well as reading from CVMFS, sending a report to CDash and auto-scaling have been successfully tested.

| | |
|---|---|
| x86-64 | ✓ |
| ARM | ✗ |
| PowerPC | ✗ |

| | |
|---|---|
| Linux | ✓ |
| Windows | ✗ |
| macOS | ✗ |

**Known limitations**

- The Kubernetes cluster is not managed by IT directly. Therefore the responsibility for maintaining and monitoring the cluster falls to the SPI project as an additional workload.

**Shortcomings**

- Deployment to CVMFS cannot be done easily within Kubernetes because it still relies on an external pre-configured release manager VM.
- Streaming the log files from the pods to Jenkins turned out to be complex.

## Future plans and outlook

- Further testing and evaluation of the Jenkins integration technologies
- Finalization of the CVMFS deployment pod by enabling and configuring the CVMFS conveyor technology[5].
- A reduction of the maintenance workload is expected from scheduled service improvements by CERN IT.

## References

**[1]** *Bulding, testing and distributing common software for the LHC experiments*, https://doi.org/10.1051/epjconf/201921405020
**[2]** *CERN OpenStack service*: https://cern.ch/clouddocs
**[3]** Bilgin Ibryam (Red Hat), 2017: *Principles of container-based application design*
**[4]** *CernVM-FS CSI driver:* https://github.com/cernops/cvmfs-csi
**[5]** *CernVM-FS conveyor:* https://github.com/cvmfs/conveyor

**Scan to download this poster as PDF**