Testing of complex, large-scale distributed storage systems: a CERN disk storage case study

Jozsef Makai1,*, Andreas Joachim Peters1,**, Georgios Bitzes1,***, Elvin Alin Sindrilaru1,****, Michal Kamil Simon1,†, and Andrea Manzi1,‡

1CERN

Abstract.
Complex, large-scale distributed systems are frequently used to solve extraordinary computing, storage and other problems. However, the development of these systems usually requires working with several software components, maintaining and improving a large codebase and also providing a collaborative environment for many developers working together. The central role that such complex systems play in mission critical tasks and also in the daily activity of the users means that any software bug affecting the availability of the service has far reaching effects.
Providing an easily extensible testing framework is a pre-requisite for building both confidence in the system but also among developers who contribute to the code. The testing framework can address concrete bugs found in the codebase thus avoiding any future regressions and also provides a high degree of confidence for the people contributing new code. Easily incorporating other people’s work into the project greatly helps scaling out manpower so that having more developers contributing to the project can actually result in more work being done rather than more bugs added.
In this paper we go through the case study of EOS, the CERN disk storage system and introduce the methods and mechanisms of how to achieve all-automatic regression and robustness testing along with continuous integration for such a large-scale, complex and critical system using a container-based environment.

1 Introduction

EOS [1] is a multi-protocol disk-only storage system developed at CERN since 2010 to store physics data for LHC and non-LHC experiments. The system became operational in 2011 and the total storage capacity has considerably increased, reaching more than 250PB in 2018. Besides physics data, EOS is also providing the back-end for a sync & share solution developed at CERN, CERNBox, and a Notebook solution, SWAN [2], which inevitably add new requirements towards the system.

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).
In addition to the already distributed architecture of EOS, starting from 2017 CERN has developed a new key-value pair store, QuarkDB, used as an EOS namespace back-end alternative in order to scale out its performance and capacity. These new additions along with new access patterns contribute to increased complexity of the overall system architecture. Given the distributed nature of the system and the growing development team, it is crucial for the future of the project to have ways to perform automatic system, performance and regression tests, possibly linked to every change committed to the code base. Automatically building a new package release based on the latest commits, deploying this new version on a newly spawned test cluster and running the full battery of tests against this new deployment are part of the continuous integration and testing strategy that we have put in place.

2 Continuous Integration for EOS

This section describes the activities performed to enable Continuous Integration for the EOS project.

2.1 CI in EOS until 2017

This process started by analyzing the build infrastructure already available at the beginning of 2017 for EOS, Jenkins CI [5], and understanding the missing features and possible alternatives. EOS was making use of a Jenkins CI server and slaves, a setup which was also shared with the Xrootd [3] project. Jenkins was used only to build RPMs for a couple of platforms (SL6 and CC7 mostly) using mock, without any automatic testing activity performed (only regular static analysis was implemented). There was no integration with other build facilities at CERN, therefore the release process was also manually driven and error prone.

In addition, the technology used was not the state-of-the-art anymore, and while the Jenkins server was supported centrally by CERN IT, the various plugins needed had to be maintained by the users. On top of that, CERN IT started at the beginning of 2017 to provide the users a GitLab CI solution [6], which has been a natural choice for the EOS team given the usage of GitLab as code management system.

2.2 Migration to Gitlab-CI

GitLab CI is an extension to GitLab implemented to offer building and testing facilities on top of code management. Everything is available via the same GitLab Web Interface while the build and testing processes are executed on separate nodes (runners) allowing scalable, parallel builds. The build and testing scripts are part of the project codebase, allowing different builds per branch and tags which can be aggregated in Job stages and Pipelines. The specification of the different stages and actions to be performed at each step are stored in the .gitlab-ci.yml file together with the rest of the codebase. The YAML format of the file makes it trivial to add new testing stages or to modify existing ones, even for new contributors. In addition, GitLab CI offers native support for Docker [7] which can be used to simplify both the building and testing phases given its built-in multi-platform support.

The project therefore started to evaluate GitLab CI for providing at first the build infrastructure. The build scripts can be executed in different docker images depending on the OS and this represented a natural replacement for mock and chroot builds. Therefore, the build scripts have been migrated to the new platform, starting from the various depending projects (e.g. Xroottd as the framework used by EOS). While migrating the scripts to GitLab CI, we also decided to implement compiler caching in order to speed up builds and make use of Google Testing as a framework for Unit Tests.
In particular, the usage of compiler caching (ccache) is crucial in a big project like EOS (650k LOC) with lots of executables, shared objects and libraries: we could not afford to recompile the whole codebase for each small change to the code every time. Therefore using compiled objects from earlier builds for unchanged code and recompiling only changed files and dependencies was the solution to go forward. The usage of ccache [8] in conjunction with GitLab CI caching has been implemented for regular commits, while tag releases are still built from scratch to avoid any issues. As seen in Figure 1 in some cases the compiler caching gave almost 40% less compilation time.

![Figure 1. Compiler cache gain build time](image)

Moving to Docker based builds has eased the support of new platforms for the EOS client by using the same build environment. Support for Ubuntu (Artful and Bionic) has been added from scratch while MacOSX builds are better supported. As far as Fedora builds are concerned, it becomes easier to add new release builds (like Fedora29 or Rawhide) as the related Docker images become immediately available for testing. Given the new platforms supported, we decided to have separate package repositories stored directly on EOS and GPG signed for each distribution flavor.

While integrating with the GitLab Pipeline mechanism we also restructured the way repositories used for package distributions were done. We now have different repositories for builds done on each commit and builds that correspond to official tag release. Moreover, we tried to simplify a previously manual step done by the operations team, which was responsible for pushing new releases to the CERN official buildsystem called Koji.

### 2.3 Coverity and Sonar

Static analysis via Coverity [10] for EOS source code was also available in the old Jenkins CI. With the migration to GitLab CI, a new Coverity Job has been added to the Pipeline in order to run once a day (see Figure 2) with a special instrumented compilation.

On the other hand, Sonar [9] analysis has been added for each commit to the master and dev branches. The cppcheck runs only at source code level, with no compilation needed. (see Figure 3) After integrating with these two static analysis tools, we put considerable effort into fixing most of the errors and warnings reported. This translated into a reduction of more than 80% of the reported issues, thus making it much easier to spot newly introduced latent bugs in the code.
The integration of Coverity and Sonar in the EOS Gitlab CI pipeline has been quite straightforward as described in the .gitlab-ci.yml code snippet in Figure 4.

**2.4 Docker based testing**

The main change with respect to the previous building system, is the usage of Docker to build EOS for various platforms and generation of package artifacts. Docker images are also a way to easily distribute the software and indeed one of the novelties added when moving to GitLab CI has been the generation and publication of EOS docker images to the CERN GitLab Docker Registry [12]. Once an EOS Docker image is built the next step is to use Docker also as a means to perform both functional and stress testing.

A new project was created in order to setup a Docker based EOS cluster (https://gitlab.cern.ch/eos/eos-docker) that is used in all the testing stages of the pipeline. The project, which can also be used outside GitLab CI, allows spawning an EOS cluster made of several containers that replicate a real production instance. The default containers are: MGM, MQ, FSTs (by default 6), EOS client, KDC for authentication and optionally a QuarkDB container (see Figure 5).

The practical advantage of this Docker-based setup is that anyone can easily deploy a fully functional EOS cluster on their laptop in a matter of seconds. This proved very convenient during targeted debugging sessions and also serves as a reference point for anyone interested in the various configuration steps needed when starting from scratch with EOS. Furthermore this has been easily adapted or served as a building block for people deploying EOS in a Docker-based environments outside CERN.
The integration of Coverity and Sonar in the EOS Gitlab CI pipeline has been quite straightforward as described in the .gitlab-ci.yml code snippet in Figure 2.

### 2.4 Docker based testing

The main change with respect to the previous building system, is the usage of Docker to build EOS for various platforms and generation of package artifacts. Docker images are also a way to easily distribute the software and indeed one of the novelties added when moving to GitLab CI has been the generation and publication of EOS docker images to the CERN GitLab Docker Registry [12]. Once an EOS Docker image is built the next step is to use Docker also as a means to perform both functional and stress testing.

A new project was created in order to setup a Docker based EOS cluster (https://gitlab.cern.ch/eos/eos-docker) that is used in all the testing stages of the pipeline. The project, which can also be used outside GitLab CI, allows spawning an EOS cluster made of several containers that replicate a real production instance. The default containers are: MGM, MQ, FSTs (by default 6), EOS client, KDC for authentication and optionally a QuarkDB container (see Figure 5).

The practical advantage of this Docker-based setup is that anyone can easily deploy a fully functional EOS cluster on their laptop in a matter of seconds. This proved very convenient during targeted debugging sessions and also serves as a reference point for anyone interested in the various configuration steps needed when starting from scratch with EOS. Furthermore this has been easily adapted or served as a building block for people deploying EOS in a Docker-based environments outside CERN.

### 2.5 Functional and stress testing

Apart from investing in this new build and deployment architecture, considerable work was put into restructuring the testing framework available in EOS. First of all, we migrated all CppUnit tests to GTest and created a single binary that contains all the unit tests grouped together. The end-to-end system test are grouped into a single home-grown bash script that gets updated regularly with test case scenarios coming from bugs or incidents encountered in production.

The system tests that require a fully functional EOS instance are run for both Red Hat and Debian based containers. In this way, we can ensure proper EOS client support for these two major distributions. The matrix of tests executed from the GitLab CI pipelines gets even more...
complicated since we are also testing the different types of namespace back-ends that are now available in EOS: the legacy in-memory namespace and the new namespace in QuarkDB.

Another type of tests that are run directly from the GitLab CI pipeline are stress tests. For these we are using a tool developed within our group called grid-hammer [13] which is capable of generating different types of load tests against the targeted back-end. It supports multiple protocols but for our use-case we are only interested in xrootd. These type of tests have proven very efficient in detecting issues that don’t manifest themselves in other more simple types of tests. Load tests are the closest thing that we have in the pipeline that match real production workloads.

A recent valuable addition to the testing infrastructure is the use of the Address and Thread Sanitizer [11] for all the unit-tests. These have been instrumental in finding subtle memory corruptions, data races and thread deadlocks in the newly developed FUSE code and not only.

3 Summary and outlook

Moving from a Jenkins build setup to GitLab CI provided us with the opportunity to expand support for different platforms and distributions while at the same time gave us the chance to improve our testing infrastructure. The entire “user- experience” when working with the GitLab CI interface is much better than before (see Figure 6).

The developers can easily follow up on the builds they are responsible for, the automatic notification mechanism in case of failures provides a baseline level of confidence in recently committed code and more importantly accountability.

We plan to extend the current setup that we have by integrating GitLab CI with Kubernetes. This would allow us to overcome the limitations brought by the fact that all containers making up an EOS cluster must now run on the same physical machine. Using a Kubernetes setup we could easily scale up or down the size of the cluster thus being able to reproduce realistic failure scenarios that only happen at scale. Last but not least, we also consider expanding and refining our testing framework to achieve better coverage of the code base and also to introduce different types of tests like American Fuzzy Lop testing.
3 Summary and outlook

Moving from a Jenkins build setup to GitLab CI provided us with the opportunity to expand support for different platforms and distributions while at the same time gave us the chance to improve our testing infrastructure. The entire "user experience" when working with the GitLab CI interface is much better than before (see Figure 6).

Figure 6. EOS Gitlab CI pipelines

The developers can easily follow up on the builds they are responsible for, the automatic notification mechanism in case of failures provides a baseline level of confidence in recently committed code and more importantly accountability.

We plan to extend the current setup that we have by integrating GitLab CI with Kubernetes. This would allow us to overcome the limitations brought by the fact that all containers making up an EOS cluster must now run on the same physical machine. Using a Kubernetes setup we could easily scale up or down the size of the cluster thus being able to reproduce realistic failure scenarios that only happen at scale. Last but not least, we also consider expanding and refining our testing framework to achieve better coverage of the code base and also to introduce different types of tests like American Fuzzy Lop testing.

References

[12] Gitlab Project, "Gitlab Container Registry" [Software], available from https://docs.gitlab.com/ee/user/project/container_registry.html