PROCEEDINGS OF SCIENCE

Modernisation of the LHCb continuous integration build system

Maciej Szymański[a,*] **and Marco Clemencic**[a]

[a] *CERN,*
Esplanade des Particules 1, P.O. Box 1211 Geneva 23, Switzerland

E-mail: maciej.szymanski@cern.ch, marco.clemencic@cern.ch

In the context of the LHCb upgrade for LHC Run 3, the experiment software builds and release infrastructure are being improved. In particular, we present the LHCb nightly builds pipelines which are modernised to provide a faster turnaround of the produced builds. The revamped system organises tasks of checkouts of the sources, builds and tests of the projects in LHCb software stacks on multiple architectures in a directed acyclic graph of dependencies, with the artifacts of each task cached and reused whenever possible, and distributes the jobs to the workers in the build farm. This work describes the implementation of the new system.

41st International Conference on High Energy physics - ICHEP2022
6-13 July, 2022
Bologna, Italy

*Speaker

PoS(ICHEP2022)229

1. Introduction

1.1 LHCb nightly builds

Nightly builds pipelines are a critical service for software development in LHCb, serving the purpose of a centralised software quality monitoring hub. The system is used by developers and maintainers to validate and test the ongoing development efforts.

LHCb software is composed of about 30 C++ interdependent projects. Due to this dependency chain, we need to build a consistent stack of projects, which we call a *slot*. At the time of writing, we build around 50 slots which correspond to different use cases, e.g. state-of-the-art developments, back-ports of features and fixes to legacy versions of the software, specific merge requests, simulation developments, etc. We include in the configuration of the slots also the versions of external libraries.

The goal of the nightly builds pipelines is to checkout[1], build and test the slots regularly (every night, unless there are no updates with respect to the previous iteration) and on demand for merge request validation workflows. The slots are built typically for several *platforms*, that is combinations of architectures, operating systems, compilers and optimisation levels.

The main requirement for the nightly builds pipelines is to provide fast feedback on the impact of their developments on builds and tests. Such feedback is presented in form of tables summarising the state of builds and tests, accessible through web pages. The *artifacts* are also deployed to a shared file system as soon as possible so that developers can use them for testing and further development.

1.2 Motivation for modernisation

While the *nightly build system* used in LHCb so far [1, 2] still works, it reached the scalability limits of its design. Moreover, the software development workflow implemented in preparation for Run 3 and beyond increased significantly the load on the system, because of the *on-demand* builds used to validate specific merge requests [3]. That evolution makes the system a continuous integration one rather than just *nightly*.

One of the main limitations of the old system is that the build of a slot for a given platform is run in a single task, which prevents us from parallelising builds when the dependency graph would allow it and forces a full rebuild in case of infrastructure–related problems (e.g. network failures). In addition, we often duplicate work because one cannot reuse artifacts between slots that share parts of the stack definition.

Finally, the legacy LHCb nightly build system is based on `Jenkins` [7], but, because of our specific needs, we do not use its features and we just rely on it as a way to execute tasks on remote machines. Moreover, the way our `Jenkins` instance is deployed (via the `OpenShift` [8] service provided by CERN IT [5]) seems to affect its performance and stability. Trying to extend the nightly build system the way we wanted looked impossible on the old infrastructure, so we decided to design the new continuous integration build system tuned to our use case.

[1] By *checkout* we mean a task getting the requested version of the project to be built. In practice, it is essentially `git clone` and `git fetch`.

2. Design of the new continuous integration build system

The design of the new system revolves around two basic principles: maximise parallelism and minimise work.

The key to improving the parallelism is to split the whole process into small independent tasks, so where the old system used monolithic jobs (checkout and build), the new system relies on smaller jobs, one per project instead of one for the whole slot. For example, instead of checking out all projects in a slot one by one and waiting for completion before starting the build, we can check out all projects at the same time and start the build of a project as soon as possible.

Minimising the work means in the first instance avoiding duplication and that can be done at multiple levels. For example, multiple jobs are using the same version of a given project so it is enough to run the corresponding checkout job only once and use its artifact multiple times. It is also possible to reuse the checkout artifacts from one day to the next if there are no changes in the code. Build artifacts can be reused too, although one has to take into account more factors to identify equivalent build tasks, like the version of the project and its dependencies, the version of the compiler and build tools, and details about the environment of the job.

The work to be undertaken is organised in a directed acyclic graph (DAG), where the nodes are actual tasks (checkout, build, and test) and the edges denote dependencies between types of the tasks (e.g. test of `LHCb` depends on build of `LHCb`) and the projects themselves (e.g. build of `LHCb` depends on build of `Gaudi`). Once the workflow is resolved, the CPU-intensive workload is distributed to a build farm.

The goal of the new system's design is to achieve a simpler and cleaner solution compared to the legacy one, intending to provide easy deployment for development and production environments.

3. Implementation

The language of choice for the CI system is `Python` thanks to its versatility, fast development workflow, and rich collection of powerful open-source frameworks. A family of `Python` packages with focused responsibilities was developed [9]. The high-level architecture of the system is shown in Fig. 1.

3.1 Scheduler

We implemented the task scheduler, responsible to coordinate the execution of tasks taking into account their dependencies, on top of the `Luigi` [10] framework. `Luigi` takes care of dependency resolution, workload management and failure recovery policies, while we provide, via the `Python` package `lb-nightly-scheduler`, the description of the tasks (what they do) and their dependencies taking into account that we share work between slots and that in our builds the dependencies are dynamic (i.e. we know the interdependencies between projects only after the checkout task, once we can inspect the project code).

3.2 Remote execution

The scheduler decides what to do and when, but the actual tasks are performed remotely on a dedicated pool of machines. To run the job on remote resources, we implemented, in the `Python`

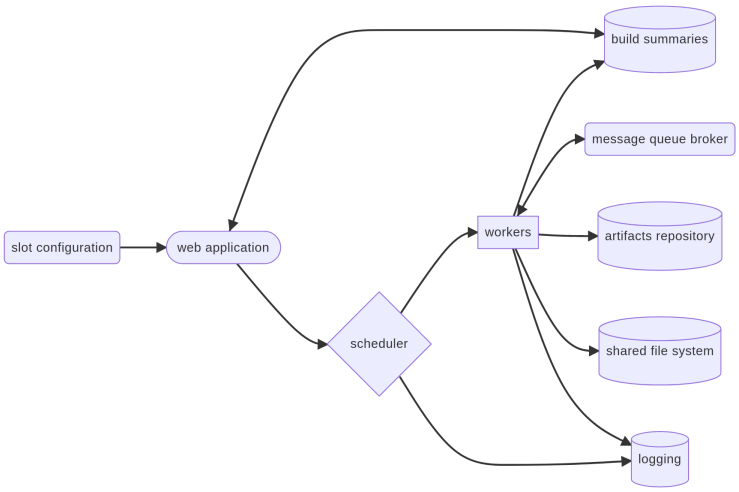

Figure 1: High-level architecture of the new CI system

package `lb-nightly-rpc`, an RPC (Remote Procedure Call) service using `Celery` [11] as the transport layer. `Celery` itself is an abstraction layer on top of a messaging system, that in our configuration is `RabbitMQ` [12] for initiating tasks and `MySQL` [13] as the results backend.

The RPC system we developed is responsible for routing the tasks to build machines depending on the CPU architecture (e.g. `x86_64` or `aarch64`, `AVX2` or `AVX512` capable CPUs), but it also provides mechanisms to tune job priorities, retry policies and other parameters.

3.3 Functions

To separate business logic from infrastructure details, we developed a dedicated `Python` package (`lb-nightly-functions`) to host the code for the actual checkout, build and test tasks. The package relies on another `Python` package (`lb-nightly-configuration`) to decode and interpret the configuration of a slot to know what to do in the specific tasks (e.g. which version of a software project to check out or the details of the build environment). `lb-nightly-functions` is also responsible for the bookkeeping related to the tasks (uploading of artifacts and logs to dedicated repositories, recording tasks states in a database, etc.) and for wrapping the execution of the tasks in `Singularity` containers [15].

3.4 Support services

A number of related services exist to support the main functionality of the system. The tables summarising the status of the builds and detailed reports on specific checkout, build and test tasks are brought to the developers by a `Flask`-based [16] web frontend, which is also used as an entry point to trigger builds on demand. The main backend database of the system is `CouchDB` database [17] instance. Artifacts are stored on an `S3` [18] server hosted at CERN (optionally with a Sonatype Nexus Repository layer [19]) and deployed to the `cvmfs` distributed file system [20]. Thanks to the recent improvements to `cvmfs` publication rates [4], we can rely on the deployment of checkout and build artifacts to `cvmfs` as a way to distribute them to all the machines in the build farm to be used by the dependent build and test tasks. Logs from checkout, build, and test tasks as well as scheduler

and worker services are published to `OpenSearch` [21] using `Fluent Bit` [22], and are available for inspection through the web application with `OpenSearch Dashboards` [21].

3.5 Deployment

All packages in the `lb-nightly` family (see the dependency graph in Fig. 2) are managed with the `Python` tool `Poetry` [25] and published to the `Python Package Index` [26] and the `conda-forge` channel of the `conda` package manager [27].

Most services rely on the CERN IT-provided infrastructure, like DataBase on demand [6] or `OpenShift`, but for the scheduler and the RPC workers we use dedicated machines where these processes are run as `systemd` services.

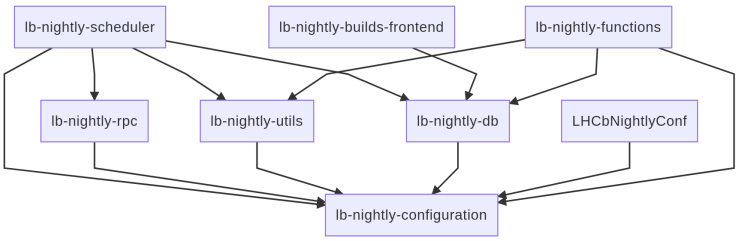

Figure 2: The dependency graph of `lb-nightly` packages.

The deployment of scheduler and RPC workers to the machines of the build farm is achieved via `conda` environments installed on `cvmfs`, which allows us to also control the versions of auxiliary applications needed for the tasks, like `Git`, `CMake`, `Ninja` etc.

It is possible to envisage a more organic deployment of the system on a `Kubernetes` [24] cluster, however, it is beyond the scope of this contribution.

4. Summary

It is crucial to provide a robust continuous integration system for building LHCb software stacks. The newly designed system is much more efficient and cleaner than the legacy one. Initial studies show increased stability, throughput, and overall performance. This was achieved by splitting and parallelising the tasks, as well as by caching and reusing the artifacts to save resources. The new CI system provides better control than the previous system based on `Jenkins`. At the time of writing, we are working on the commissioning of the system.

References

[1] M. Clemencic and B. Couturier, J. Phys. Conf. Ser. **513** (2014), 052007 doi:10.1088/1742-6596/513/5/052007

[2] M. Clemencic and B. Couturier, J. Phys. Conf. Ser. **664** (2015) no.6, 062008 doi:10.1088/1742-6596/664/6/062008

[3] R. Currie, R. Matev and M. Clemencic, EPJ Web Conf. **245** (2020), 05039 doi:10.1051/epjconf/202024505039

[4] E. Bocchi, J. Blomer, B. Couturier, C. Burr and D. van der Ster, EPJ Web Conf. **251** (2021), 02034 doi:10.1051/epjconf/202125102034

[5] A. Lossent, A. Rodriguez Peon and A. Wagner, J. Phys. Conf. Ser. **898**, no.8, 082037 (2017) doi:10.1088/1742-6596/898/8/082037

[6] R. G. Aparicio, D. Gomez and I. Coterillo Coz, J. Phys. Conf. Ser. **396**, 052034 (2012) doi:10.1088/1742-6596/396/5/052034

[7] https://www.jenkins.io/

[8] https://www.redhat.com/en/technologies/cloud-computing/openshift

[9] https://gitlab.cern.ch/lhcb-core/nightly-builds

[10] https://luigi.readthedocs.io/en/stable/

[11] https://docs.celeryq.dev/en/stable/

[12] https://www.rabbitmq.com/

[13] https://www.mysql.com/

[14] https://gitlab.cern.ch/lhcb-core/LHCbNightlyConf/

[15] https://apptainer.org/

[16] https://flask.palletsprojects.com

[17] https://couchdb.apache.org/

[18] https://aws.amazon.com/s3/

[19] https://www.sonatype.com/products/nexus-repository

[20] https://cvmfs.readthedocs.io/en/stable/

[21] https://www.opensearch.org/

[22] https://fluentbit.io/

[23] https://docs.docker.com/compose/

[24] https://kubernetes.io/

[25] https://python-poetry.org/

[26] https://pypi.org/

[27] https://conda.io/

[28] https://docs.gitlab.com/ee/ci/