Version 1.0 (17 February 2017) https://github.com/mazurov/chep2012/compare/v1.9...v1.10

Microservices for systematic profiling and monitoring of the refactoring process at the LHCb experiment

A Mazurov[1] **and B Couturier**[2]

[1] University of Birmingham, Birmingham, United Kingdom
[2] CERN, European Organization for Nuclear Research, Geneva, Switzerland

E-mail: `alexander.mazurov@cern.ch`

Abstract. Any time you modify an implementation within a program, change compiler version or operating system, you should also do regression testing. You can do regression testing by rerunning existing tests against the changes to determine whether this breaks anything that worked prior to the change and by writing new tests where necessary. At LHCb we have a huge codebase which is maintained by many people and can be run within different setups. Such situations lead to the crucial necessity to guide refactoring with a central profiling system that helps to run tests and find the impact of changes.

In our work we present a software architecture and tools for running a profiling system. This system is responsible for systematically running regression tests, collecting and comparing results of these tests so changes between different setups can be observed and reported. The main feature of our solution is that it is based on a microservices architecture. Microservices break a large project into loosely coupled modules, which communicate with each other through simple APIs. Such modular architectural style helps us to avoid general pitfalls of monolithic architectures such as hard to understand a codebase as well as maintaining a large codebase and ineffective scalability. Our solution also allows to escape a complexity of microservices deployment process by using software containers and services management tools. Containers and service managers let us quickly deploy linked modules in development, production or in any other environments. Most of the developed modules are generic which means that the proposed architecture and tools can be used not only in LHCb but adopted for other experiments and companies.

1. Introduction

In LHCb, as in all High Energy Physics (HEP) experiments, complex software is used to process the data recorded by the detectors. Whenever developers change or modify their software, even a small tweak can have unexpected consequences. Regression and performance testing is testing existing software applications to make sure that a change does not break or degrade any existing functionality. The purpose of these tests is to catch issues that may have been introduced into a new build and to ensure that previously eradicated issues continue to stay fixed. By re-running testing scenarios that were originally scripted when known problems were first fixed, you can make sure that any new changes to an application have not resulted in a regression, or caused components that formerly worked to fail. Such tests can be performed manually on small projects, but in the HEP software repeating a suite of tests each time an update is made is too time-consuming and complicated to consider, so an automated testing tool is required.

The LHCb Performance and Regression (LHCbPR) [1] project is a key component in the LHCb Software [2] development which provides support to conduct systematic profiling and allows comparing the results of performance and regression tests run on the LHCb applications. In this paper we introduce a new architecture and implementation of LHCbPR system which allows to avoid some major pitfalls of the previous version [1].

Section 2 gives a brief overview of the previous LHCbPR version, its advantages and disadvantages, following by the description of the new architecture. In Section 3 the implementation details are outlined.

2. LHCbPR Software Architecture

Figure 1 presents a general sequence diagram of the interaction between the tests running service, LHCbPR and users. The test service requests from LHCbPR information on how to run tests, then actually runs the tests and finally saves the results back to LHCbPR. Users have an ability to retrieve and analyse these tests results.

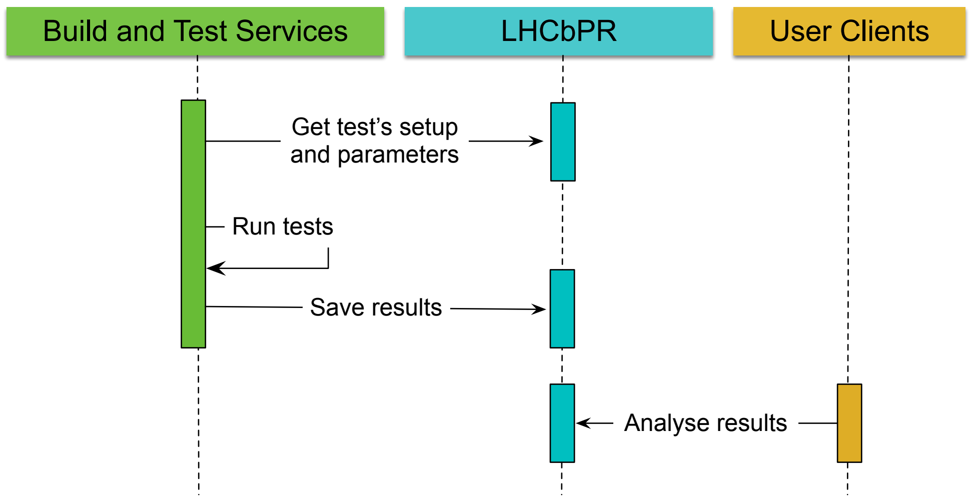


Figure 1: General sequence diagram of interaction between test running services, LHCbPR and users

2.1. Monolithic architecture

The first version of the LHCbPR had a monolithic architecture, in which all functionality is integrated into one application. All components of such application are stored in one codebase that leads to the common inherent issues like:

- The application can be difficult to understand and modify. That intimidates new developers and slow down the development process.
- There are usually no hard boundaries between components and these can be easily broken.
- Continuous deployment is difficult since in order to update one component you have to redeploy the entire application. This is especially an issue in components that require rapid and frequent redeployment, like user interfaces.
- A monolithic application forces you to be coupled to the certain technology stack. That leads to the situation where, for example, it can be difficult to adopt a newer technology stack.

The monolithic architecture was chosen initially since it has benefits at the prototyping stage of the software products:

- Simple to develop a small application since you are working with one codebase and all developers use the same technology stack.
- Simple to deploy application since you usually have to run the application in the same runtime and environment.
- Simple to scale — you can run multiple copies of the application behind a load balancer.

But the advantages of such architecture works only for small applications, usually at the time when you only start to develop it. In the growing monolithic application the disadvantages become more annoying and slow down the development process. One of the solutions is to use the microservices architecture.

2.2. Microservices Architecture

In opposite to the monolithic architecture, the microservices architecture applications are built from the smaller components, which are usually stored in separate codebases and are independent of each other. In the new version of LHCbPR components communicate through simple Application Programming Interfaces (APIs). Such components are commonly named microservices.

Microservices help to avoid general pitfalls of the monolithic architecture. Smaller components are easier to understand and modify. There are strict boundaries between components and it is almost impossible to break one component by modifying the other. Only changes of API can lead to some disruptions.

Solutions based on microservices have a number of drawbacks:

- Complexity of creating distributed system.
  - We need to have a good coordination between each component's team. In the development process APIs can be modified which can lead to the rewriting of some component parts.
  - In distributed systems we have to use the communication technologies that are usually not needed in monolithic applications. For example, microservices can be run on different nodes and communicate with each other through synchronous HTTP interfaces or through asynchronous messaging systems.
- Deployment complexity. Each of the components can be run in different environments and use various technologies, which leads to additional efforts for supporting and deploying such systems.

The next section shows details of the new LHCbPR implementation and outlines how major microservices pitfalls were solved.

3. LHCbPR implementation

Figure 2 details the implementation of the new LHCbPR microservices architecture.

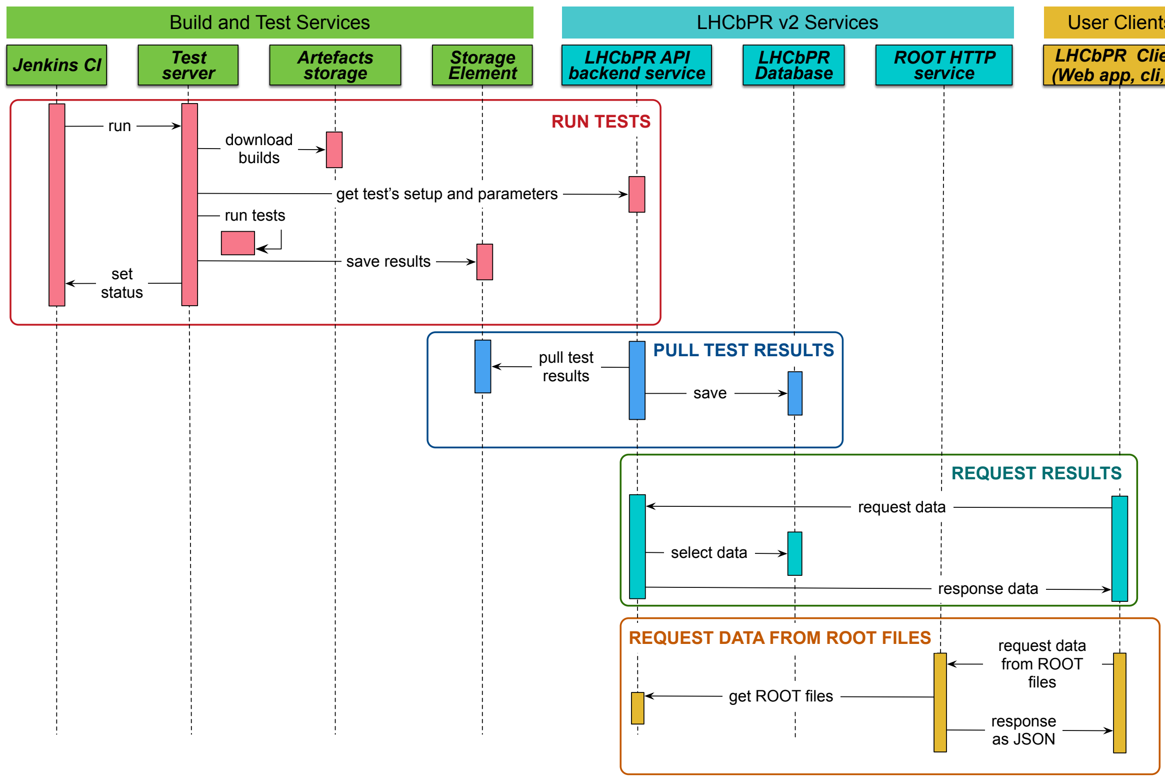


Figure 2: LHCbPR v2. Sequence diagram of interaction between test running services, LHCbPR's microservices and users.

The test service communicates with LHCbPR for the following purposes:

- **Retrieve test parameters**. The test manager configures the schedule for running tests. Each record of the schedule provides information about the build environment: application under the test, name of the test, name of the special software handler for test results postprocessing, operating system, compiler version and machine architecture. At the scheduled time the test service requests additional information from LHCbPR application: the name of the program for running the test and the path to the test script. After the testing finishes the required test results are saved to LHCbPR. All information about the tests and its results are stored at MySQL database. The other components can access the database through the LHCbPR API backend microservice. This service is the REST web service that allows clients to retrieve and update information in the database be making HTTP requests. The advantage of the service is that it hides database implementation — clients can retrieve only information that the service provides and could not directly access the database. Most of the changes in the database schema or in the database engine are almost transparent for the service clients.
- **Save test results**. Test results are packed by test service into an archive file and uploaded to the network storage. LHCbPR backend service pull the results into the database by executing the corresponding import script. Results become immediately accessible through the REST API.

3.1. API service

The API service is open for any program that supports HTTP requests. The common use of the service is to retrieve test results which are provided in JSON. For example, the important part of the LHCbPR infrastructure is a web application (Section 3.3), which allows users to search and analyse results. This application is built on top of two services: (1) the LHCbPR API service and (2) LHCbPR ROOT service (Section 3.2).

3.2. ROOT service

In many tests that are used in HEP the output contains histograms and graphs. One of the possible regression testing analysis is to compare histograms. Such comparison can be done by presenting histogram ratios or by evaluating the Kolmogorov probability. In LHCbPR we developed a web microservice that can convert almost any ROOT object into JSON and output histogram comparison results. This functionality is based on the TBufferJSON class in ROOT. Any user client can retrieve results by making simple HTTP requests and analyse the output without installing ROOT.

3.3. Web application

The regression tests do not make sense if there are no tools for analyzing the results. For that reason a web frontend was created that uses the API and ROOT services described above.

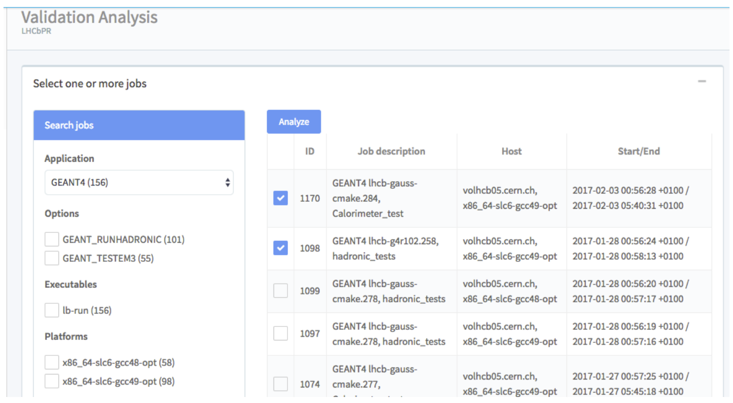
This frontend has the following functions:

- **Find test results** by their parameters like name, application, platform. Figure 3(a) shows an example of the corresponding interface.
- **Compare and present the results** of the selected tests in the user-friendly interface.

The frontend is a single-page AngularJS [6] web application that loads a single HTML page and dynamically updates that page as the user interacts with the application. We use web server only for handling static pages and all operation with the related microservices are done with AJAX requests.

The main part of the web application is an analysis module — a set of javascript code and HTML that is responsible for combining and presenting the selected tests. Analysis modules work like plugins — each module is stored in the dedicated folder and does not depend on the other modules. The application is structured so people can work independently on modules. For the developers convenience the main library contains a set of web components that implements the functionality common to most of the analysis modules. For example, the common components are the form for searching test results and component for displaying histograms retrieved from the ROOT service.

The application can have as many analysis modules as number of tests or even more. But it is not necessary to create analysis for each test because the application includes generic modules that can handle results of most of the tests. For example, it includes a module that can plot trend analysis of any numeric value that was probed in different versions of some application. Another generic module can compare histograms from any ROOT files provided in the test results as shown at Figure 3(b).


Validation Analysis
LHCbPR

Select one or more jobs

Application
GEANT4 (156)

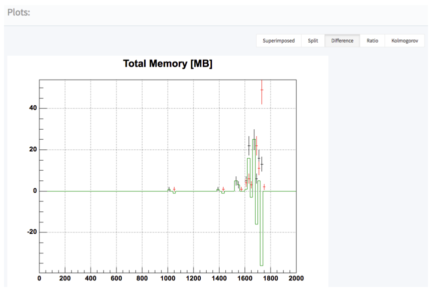
Options
☐ GEANT_RUNHADRONIC (101)
☐ GEANT_TESTEM3 (55)

Executables
☐ lb-run (156)

Platforms
☐ x86_64-slc6-gcc48-opt (58)
☐ x86_64-slc6-gcc49-opt (98)

	ID	Job description	Host	Start/End
✓	1170	GEANT4 lhcb-gauss-cmake.284, Calorimeter_test	volhcb05.cern.ch, x86_64-slc6-gcc49-opt	2017-02-03 00:56:28 +0100 / 2017-02-03 05:40:31 +0100
✓	1098	GEANT4 lhcb-g4r102.258, hadronic_tests	volhcb05.cern.ch, x86_64-slc6-gcc49-opt	2017-01-28 00:56:24 +0100 / 2017-01-28 00:58:13 +0100
☐	1099	GEANT4 lhcb-gauss-cmake.278, hadronic_tests	volhcb05.cern.ch, x86_64-slc6-gcc48-opt	2017-01-28 00:56:20 +0100 / 2017-01-28 00:57:17 +0100
☐	1097	GEANT4 lhcb-gauss-cmake.278, hadronic_tests	volhcb05.cern.ch, x86_64-slc6-gcc49-opt	2017-01-28 00:56:19 +0100 / 2017-01-28 00:57:16 +0100
☐	1074	GEANT4 lhcb-gauss-cmake.277,	volhcb05.cern.ch, x86_64-slc6-gcc49-opt	2017-01-27 00:57:25 +0100 / 2017-01-27 05:45:18 +0100

(a) Search tests' results. Search form (left) and search results table (right).



(b) Show difference of the histograms from the selected tests

Figure 3: Web application. Example of analysis module.

4. Deployment

The microservices pitfall appears to be the difficulty in developing components in different runtimes and in different technology stacks.

For example, LHCbPR uses the following tools and libraries:

- **API service**. *Django REST framework* [5], *MySQL* database, *Gunicorn* [7] HTTP server
- **ROOT service**. *Flask* [10] framework, *Gunicorn* HTTP server, *ROOT*.
- **Web frontend**. *NGINX* [8] HTTP server, *Node.js* [9] for developing.

It will be hard for developers to test and deploy such set of technologies — they need to be sure that services are run in the same environment, at the same operating system and use the same versions of libraries. The most challenging problem is to make all services communicate with each other in the proper way.

These issues are resolved in LHCbPR by packaging services into *Docker [3]* containers and orchestrating them with the *Docker Compose [4]* tool. *Docker* containers guarantee that the operating system and software will always run the same regardless of its environemnt. *Compose* is a tool for defining and running multi-container *Docker* applications. In our case, such applications are LHCbPR's services. *Compose's* configuration can be shared between developers, so they can be sure that the services communicate in the same environment. The deployment into production becomes smooth since the same *Docker* images are used in development and production environments. The major difference between these environments is only *Compose's* configuration.

Figure 4 shows the current deployment diagram of LHCbPR. At the moment all services run at the dedicated virtual machine. All user requests go through the proxy server that redirects them to the corresponding service. All files, produced by tests, are stored at the network *CEPH* volume which is accessible from all services. *MySQL* database is managed by the CERN's *Database On Demand* service. Our plan is to scale deployment to several virtual machines that can be done easily since we can use *Docker* tools for that.

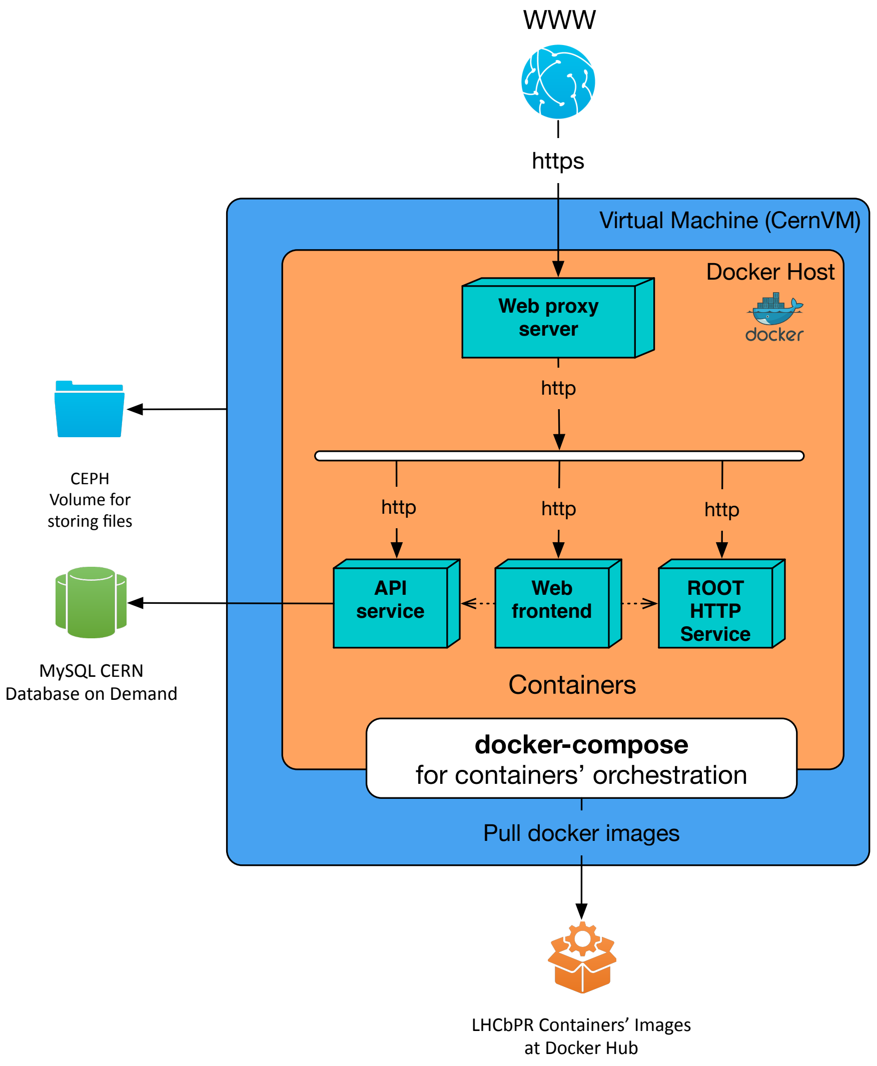


Figure 4: LHCbPR deployment diagram

5. Summary

In this paper we presented a new version of the LHCb performance and regression testing framework. This version demonstrates the advantages of microservices approach for building complex applications.

The new LHCbPR is currently used in the production at LHCb experiment and handle test results of simulation and trigger software. LHCbPR is not coupled with the features specific to LHCb. It is possible to reuse LHCbPR in other experiments or companies. For using LHCbPR, test services should only know how to retrieve tests information from the LHCbPR API service and pack test results to the format specific for this service.

We are currently focusing on improving generic analysis modules in LHCbPR web application and always welcome any developers that would like to adopt LHCbPR for their needs.

References

[1] "Systematic profiling to monitor and specify the software refactoring process of the LHCb experiment", Ben Couturier et al 2014 J. Phys.: Conf. Ser. 513 052020

[2] ”Software for the LHCb experiment”, Corti, G. and Cattaneo, M. and Charpentier, P. and Frank, M. and Koppenburg, P. and Mato, P. and Ranjard, F. and Roiser, S. and Belyaev, I. and Barrand, G., 2006 IEEE Trans. Nucl. Sci.
[3] Docker, https://www.docker.com/
[4] Docker Compose, https://docs.docker.com/compose/
[5] Django REST framework, http://www.django-rest-framework.org/
[6] AngularJS, https://angularjs.org/
[7] Gunicorn, http://gunicorn.org/
[8] NGINX, https://www.nginx.com/
[9] Node.js, https://nodejs.org
[10] Flask, http://flask.pocoo.org/