Testing and Deployment Strategies - Their Impact on Accelerator Performance

J-C. Garnier, CERN, Geneva, Switzerland

Abstract

The Accelerator Control system is developed by the Controls Group in collaboration with numerous software developers from equipment groups to provide equipment experts and operation crews with general and specific control and supervision software. This presentation proposes a review of the different strategies applied for testing, validation and deployment of the software products. The risk and impact of these strategies on the LHC [1] performance and availability will be studied, and possible evolution to continuously improve this performance will be discussed.

INTRODUCTION

This paper intend to present a review of common software engineering activities which are testing and deployments, and how it impacted the performance of the LHC in 2016.

Software testing is a way to prove that the software is performing in the way it is required by the stakeholders. It is also a way to inform both stakeholders and other developers of the behavior and the quality of the software.

Software deployment is the process of ultimately using the software in a production environment. This process can consist of few or many steps, depending on the choices of the software engineers and users of the software.

As any software engineering activity, numerous ways to perform them have been advertised, demonstrated, accepted or denied by the software engineering community.

This paper focuses on Java, C++ and Python technologies and on high level software, ranging from Front-End Controller software (e.g. FESA [2]) to User Interfaces, Services and Analysis Tools. This papers references multiple times to controls software or controls environment, which means the software and infrastructure produced by the Controls Group and the Equipment Groups together.

This study was done following a top-down approach, where the ultimate upper goal is the performance of the Accelerators. From the estimated impact on the performance, one could identify issues in the deployment strategies and the testing strategies.

The first section describes the investigation performed to review the impact of control software on LHC performance. The second section describes the testing strategies applied during development, release and deployment of the control software. The third section describes the deployment strategies that are used for software in the control system. The fourth and last section gives an outlook of possible directions for the future.

REVIEW OF TEST AND DEPLOYMENT IMPACT ON LHC PERFORMANCE

Tracking Tools

The accelerator control system relies on a multitude of tools to track events, faults and actions: logbook, AFT [3], central tracing service, etc. Coordination initiatives complete this list: Exploitation Meetings and Smooth Upgrade Working Group (SUWG).

The most visible and exploitable tool is the logbook where every step of the accelerator operations are recorded. Entries can be added manually by operators or experts, or automatically by software tools such as the LHC Sequencer. The logbook will typically record every software deployment that was announced to the operators. The fact that these recordings are written by hand, and only when announced, proves the analysis to be difficult as some deployments are indeed not referenced, and the referenced deployments are reported in very different ways. Nonetheless, if a quantitative evaluation was impossible for 2016, a qualitative evaluation has been performed and one can see that deployments have had a negative impact on the performance of the LHC, and multiple hours of operations were compromised after software deployments. There were different causes:

- lack of tests and validation before the deployment to operation
- validation could only be performed with beams
- impact on dependent systems was underestimated

This last point is very significant, as it is frequently mentioned, not only during deployment and integration in operation, but also during the development phases when the contribution of other software teams is often underestimated.

Correlating data from the logbook with other tools proved to be difficult and extremely time consuming. A deployment tracking system is clearly required in order to supervise this activity and allow the collection of performance metrics. It could help identifying all deployments: major deployments, bug fixes and rollbacks.

Coordination of deployments

Another source of useful information was the Smooth Upgrade Working Group. It reviews and coordinates the deployment of software product for each technical stop. It evaluates the risk for operation, how the software can be validated, and if it can be rolled back. In addition it tries to identify the outgoing and incoming dependencies

of the software. In spite of all the dedication and expertise of its members, it is difficult if not impossible to oversee all dependencies and prevent all issues, and it keeps track of them as they raise. The amount of upgrades that are reported to the SUWG is very important. This brings difficulties in identifying the risk of all of them while it relies on manual analysis. The SUWG could take advantage of some software support, like a dependency tracking system that would report that a software A is used by software B and C, and that an upgrade of Software A needs to be reviewed with software B and C developers.

From this investigation, one can see that some software is required to better support the deployment of software product in the LHC environment. Software could:

- help planning the deployments, by providing an accurate and exhaustive understanding of the controls environment and the dependencies on the deployed software.
- collect metrics about deployments, improving the identification of shortcomings induced by this activity.

REVIEW OF TESTING STRATEGIES

Testing is a wide term and numerous type of software testing can be identified. The following ones are considered in this paper:

- Unit tests that focus on an isolated unit of code, typically a function or a class.
- Integration tests that validate the interactions of multiple components together.
- User Acceptance Tests to confirm that the user specifications are fulfilled.
- Testing on a Staging environment, which is ideally as close as possible to the production environment.
- Final Validation in production, the time devoted to this last validation is reduced if other testing strategies are applied.

If writing tests has an initial costs, it also brings numerous advantages to the product, hence a valuable return on investment, particularly on long term projects like in the controls environment. They can be seen as documentation for a function, a class or a feature. They ease the long term maintenance of the software applications as they help engineers new to the project understand the source code and prove that it behaves as expected, and make them confident in refactoring - e.g. modifying the existing code - when fixing bugs or adding new features. The automation and repeatability of the tests also helps understanding the impact of any modification in the source code or in a third party dependency on the application behavior.

Unit testing and integration testing are two kind of tests that must be considered at design time. All software is not unit testable, but all software can be written in a testable way. The ability to test the source code of an application also highly depends on the third party software components on which the application relies. In the controls environment, different strategies have been applied through time, and depending on the interests of the providers of these third party tools. It results in a discrepancy of the ability to test controls software. For instance the FESA framework makes it complicated to write unit or integration tests as it is very difficult to isolate the logic under test from the FESA framework. This is a common issue with frameworks that force the user code to be coupled with them. On the other hand, when using a library like CMW, JAPC, Post Mortem, it is rather simple to add an abstraction between the source code of the application and the third party library, which in turn makes the application source code testable without depending on the third party library.

In a machine like the LHC, where the objectives are ambitious and the machine protection risks significant, it is important to consider that core components used by many applications should not impair the testability of these applications.

Software testing is supported by metrics that helps understanding the presence and quality of the tests. Two very important metrics are the coverage of lines of code and of branches. A line of code is covered if it is executed once by a test. Branch coverage highlights the fact that all conditions of a branch (e.g. if statement) are tested. The controls environment registers 654 Java projects. 365 of these projects have less than 50% of line of code coverage. This means that more than half of the software that runs the controls environment is not tested. These metrics only concerns Java because it was simple to integrate the entire controls software stack thanks to an homogeneous development strategy. C++ and Python remain not monitored at the moment, except on the initiative of the developers. These metrics are exposed publicly at CERN via SonarQube [4]. Exposing these metrics to users of the product could encourage the developers to provide high quality software and improve the user confidence in the product.

While unit and integration testing are very good tools to validate the way the source code behaves, user acceptance tests are of tremendous significance to highlight that the software is acting the way it should. Combined with automation, they guarantee that a feature stays available for the lifetime of the software product. These tests usually need to run in a staging environment in order to be effective. They however often must rely on third party systems that are not available in the staging environment, for instance beams. It is usually possible to compensate this with simulated or mock components. A successful design of user acceptance tests in a staging environment is for instance the Orbit Feedback system, which relies on automated tests performed without beams. This testing strategy helped in understanding the legacy software system, and reduced the testing time in operation with beams, that requires both operators and experts to be there to validate

the software. Many initiatives exist to provide test environments, for CMW, Timing and FESA, or for FGCs. Machine Protection is currently implementing a project of a test bed for magnet protection and interlock equipment.

At the moment, controls and equipment groups usually all have their own staging environment, which is usually limited to their area of concern and rarely provides possibilities of integration with systems from other equipment groups. Based on the experience gathered during numerous hardware commissioning, and foreseeing the work necessary to further automatize the machine commissioning and beam checkout, the MPE group is currently working on a staging environment that will enable performing integration testing on entire powering systems, comprising FGCs, QPS, interlocks systems and controls software.

The controls environment currently brings some limitations to the proper implementation of staging environments: there are too many dependencies on the Technical Network infrastructure, where the operational systems are also running. A proper staging environment would require a correct separation between itself and the operational environment. Numerous attempts had been performed in this direction but they were always unsuccessful. The Controls Group under the CO3 impulse is currently reviewing their numerous project and will come back with recommendations on providing core services to staging environment out of the Technical Network.

The more time is spent on unit testing, integration testing, user acceptance testing and staging, the less time is required for validation in the production environment. There is of course a lack of metrics illustrating this in our environment. Such metric could be provided by a deployment service, that would help comparing the time required to validate software in operation with the software code coverage.

REVIEW OF DEPLOYMENT STRATEGIES

The fact of deploying an application consist of following a process. Two types of applications must be considered here: Graphical User Interfaces (GUIs) and services. Both do not follow the same deployment strategies. Services can then be split into two categories: FECs with e.g. FESA, and Java services.

GUIs

In the controls environment, deploying a GUI consists of one step: a release. A release will deliver the GUI product to a shared file system with its version number. The delivered product is a JNLP (Java Network Launch Protocol) that can be started anywhere provided that a Java Runtime Environment is available.

A PRO release will update a symbolic link that is usually referred by numerous users, as it is the link to the latest production ready GUI. When a PRO release is performed, the previous PRO release is aliased with the PREVIOUS alias. This simple mechanism ensures that either users can rollback to the previous valid version of the application by modifying the link they use, either developers can modify the PRO link to the previous stable version.

The particularity of the deployment area is that it is mutable, and modifications of the deployed production GUIs are sometimes performed in particular situations where a core service or library needs all its clients to be updated. The danger in this operation is that production GUIs are not tested after this modification and it could lead to unexpected behaviors.

Some alternative processes are implemented by other equipment groups. For instance, BI is relying on a single and common classpath for all their applications. It guarantees the compatibility of their application between each others, while all applications are modified if new releases of core components are performed. In this environment, a strong validation step is required to ensure validity of the applications. Another example of alternative process is used in MPE, where the GUI deployment process is composed of three steps: a release, an automated validation, and an immutable deployment. This process guarantees that the application is validated by automated user acceptance tests before it can be used in operation, and that the application used in operation will never be modified. Any modification even a simple modification of dependencies must go through the automated validation step.

Services

Java servers deployments in the controls environment consists of two steps: a release, and a deployment on the server that hosts the service. In order to deploy, the developers authenticate to the operational server and elevate their privileges to run some deployment commands and start-up commands. With their elevated privileges, the developers can unintentionally perform actions that might impact other services, there is no protection against that at the moment. After deployment, previous versions of the service are kept to be used for roll back.

Rolling back a service doesn't rely on commands, but on file system and symbolic link manipulations. There is currently no consistent solution offered for multi-layer applications that consist of server, database and GUI. It can be a challenge to rollback all these at once.

Here MPE and BE-CO-DS are using Continuous Integration and Continuous Delivery in order to support fast delivery of features to their services. These implementations rely on scripts that automate the deployment based on the authentication and privilege elevation aforementioned. It removes the risks that a human does not interact anymore directly with remote machine, but the underlying mechanism is still the same and can still be error prone.

In addition, a link must exist between the General Purpose Network and the Technical Network in order for developers to deploy their services. A clean deployment strategy could provide capabilities to developers to release and deploy their applications without having to act on themselves on the operational infrastructure. This would help

reducing the risks of human errors.
Deploying a FESA class is yet another but very similar procedure. The difference mainly consists in the fact that the API of the FESA class is exposed in a database.
In the current environment, there is no restrictions on the deployment of operational software, whether GUIs or servers. A deployment can be performed at anytime without any constraints coming from the beam presence or operation planning. Indeed responsible people are aware of these constraints and coordinate as best as possible with their interested colleagues for a deployment. The risk of deploying anytime and the entanglement between development environment and operational environment could be avoided with an integrated deployment tool.

Deployment Service

This paper identified the following possible areas of improvements in the testing and deployment of software packages:

- Testing out of the production environment in order to minimize the validation time in production
- Simple and safe deployments and rollbacks that can be performed by Operation Crews when they judge that there is a time window to do so
- Tracking of the time spent to validate a software product in production, until it is accepted or rejected
- Proper separation of the development environment from the operational environment
- Tracking of dependencies between services and API versions

Such a deployment service is an investment that would make the software deployment step safer and more transparent. It cannot be built and provided at once, but should rather grow step by step. A first milestone would simply be the tracking of the deployed versions and the incoming versions that operators could deploy and validate. A valuable extension that could come afterward is the tracking of the dependencies between the software services, and their versions.

Numerous open source tools can support parts of such a deployment service.

OUTLOOK

This paper has explained that it could not rely on any quantitative metrics of the impact of testing and deployment strategies of control software on the accelerator performance. This illustrates however a scarcity in the controls environment that can be fulfilled. This paper has illustrated that most of the Java software stack remains uncovered by unit and integration tests, and that such metrics are missing for other programming languages. It shows however that controls and equipment groups are equipped with testbeds that allow them to validate their software in a closed staging environment. Larger testbeds can be implemented to validate the integration of software services together. This paper illustrates that the ways deployment are performed are not optimal and comport some risks, and that a technical solution to improve this could actually provide valuable metrics in order to better understand the performance of software deployments.

ACKNOWLEDGMENT

I would like to express my gratitude to Kajetan Fuchsberger and Grzegorz Kruk for allowing me to work on this vast, motivating and challenging topic.

REFERENCES

[1] LHC Study Group et al., “The Large Hadron Collider, Conceptual Design”, CERN, Geneva, Switzerland, Rep. CERN/AC/95-05, Oct. 2012.

[2] M. Arruat et al., “Front-End Software Architecture”, ICALEPCS07, Knoxville, Tennessee, USA, 2017.

[3] A. Apollonio et al., “LHC Accelerator Fault Tracker - First Experience”, IPAC2016, Busan, Korea, May 2016.

[4] http://sonar.cern.ch