Journal of Physics: Conference Series **513** (2014) 052017 doi:10.1088/1742-6596/513/5/052017

The ATLAS data management software engineering process

M Lassnig[1], V Garonne[1], G A Stewart[1], M Barisits[1], T Beermann[2], R Vigne[3], C Serfon[1], L Goossens[1], A Nairz[1], A Molfetas[4], on behalf of the ATLAS Collaboration

[1] European Organization for Nuclear Research (CERN), 1211 Geneva, Switzerland
[2] Bergische Universität Wuppertal, 42119 Wuppertal, Germany
[3] University of Vienna, 1010 Vienna, Austria
[4] University of Melbourne, 3010 Victoria, Australia

E-mail: mario.lassnig@cern.ch

Abstract. Rucio is the next-generation data management system of the ATLAS experiment. The software engineering process to develop Rucio is fundamentally different to existing software development approaches in the ATLAS distributed computing community. Based on a conceptual design document, development takes place using peer-reviewed code in a test-driven environment. The main objectives are to ensure that every engineer understands the details of the full project, even components usually not touched by them, that the design and architecture are coherent, that temporary contributors can be productive without delay, that programming mistakes are prevented before being committed to the source code, and that the source is always in a fully functioning state. This contribution will illustrate the workflows and products used, and demonstrate the typical development cycle of a component from inception to deployment within this software engineering process. Next to the technological advantages, this contribution will also highlight the social aspects of an environment where every action is subject to detailed scrutiny.

1. Introduction

The ATLAS experiment creates and manages non-trivial amounts of data [1]. Since the detector started taking data, the experiment's distributed data management system Don Quijote 2 (DQ2) [2] is responsible to manage petabytes of data on the Worldwide LHC Computing Grid [3]. DQ2 also manages the entire lifecycle of experiment data, from raw detector data up to derived data products from physics analysis. The governing technical policies are defined by the ATLAS Computing Model [4].

With the upgrade of the Large Hadron Collider (LHC) in progress, the rate of data in the experiment will increase drastically in the next years of operation. A study on the performance and usability metrics of DQ2 showed that it will not likely be suitable to

cope with the increasing demands [5]. For this reason, and to cope with new use cases put forth by the physics community, a new version of the data management system has been developed, called Rucio.

This contribution gives an overview of the software engineering process that has been used to build Rucio, and how it differs from the commonly employed process in the ATLAS distributed computing community. First, Section 2 describes the requirements engineering process that was used to establish the required use cases and scenarios. Section 3 describes the design phase, where the requirements were translated into functional pieces for development. Section 4 describes the actual implementation process and how Rucio is tested. Section 5 details the non-technical factors in the whole process, such as human interaction, and Section 6 concludes with a summary and an outlook.

2. Requirements engineering

Rucio development follows a waterfall model in principle – that is, in order: specification, design, construction, integration, testing, installation, maintenance. However, the actual construction and integration phases follow an agile test-driven model. This has two reasons, primarily to mitigate potential conflicts with third-party software as efficiently as possible, and secondly, to make better use of available personpower due unforeseen extended waiting times. For example, such waiting times can include prolonged procurement of hardware, or remote grid site administration and configuration. The testing step in the waterfall model thus becomes almost obsolete because the construction and integration steps have already thoroughly tested the full system.

The first step in building Rucio was to gather all existing requirements of DQ2 from the experiment, to collect possibly new requirements for the upcoming LHC and experiments activities, and to distil this information into a conceptual model. This included sending out surveys to the main clients of the data management system, the Tier-0, the metadata service AMI, data preparation groups, reprocessing groups, the workload scheduler PanDA, the physics groups, and the end users, that is, the physicists themselves. Separate sessions where then held with each group, where the main points of their written responses were discussed. This lengthy process took about 4 months in the latter half of 2011. In the meantime, technical meetings with the other LHC experiments, CMS, ALICE, LHCb, were setup, in which the general future directions of the data management for all experiments were laid out. This allowed to find overlaps and differences easily, and in turn influenced also the question-and-answers in the ongoing sessions from the user surveys. Finally, a workshop on the evolution of ATLAS data handling was held, in which the main directions were discussed, and the final use cases were confirmed. The results from all this work have been condensed into the first iteration of the Rucio conceptual design document [6]. The first step of the waterfall model – specification – was not completed, however. The conceptual model was distributed for wider review to the ATLAS community to ensure that the information was correctly understood and described by the Rucio team. It only took three iterations to come to a conclusion of the document, with only minor changes and one additional use case added. The document was finally extended to include specific descriptions of each component within Rucio for the design stage, and has been approved by the experiment.

Since the whole specification process was very lengthy, the intermediate time was

used to setup an infrastructure to support the later stages of the process. This included testing of various software packages for suitability and scalability with educated guesses, including the database management system, the database abstraction layer, communication protocol libraries, or monitoring tools.
The design phase followed with a component oriented view. As the basic system functionality was already laid out by the design document, it was easy to devise components that fulfil the use cases. One engineer was assigned to each component to write a design proposal that was discussed with the rest of the team using shared workspaces. This allowed the team to work on the full system design in parallel, coming to a rapid conclusion within less than 2 months. The downtime between component design revisions was used to further test various software packages and setting up a development environment for the actual implementation.

3. Implementation and test

The implementation of Rucio follows one guiding principle: it must be always releasable from the master branch. This requires that the master branch is fully functional at all times. To ensure this, the following actions have been taken:

- No engineer is allowed to modify the master branch directly.
- New features are only implemented in separate *feature branches*.
- Feature branches are only merged into the master after approval by at least two other engineers.
- Approval by an engineer requires
  - visual scanning of the source code,
  - local test of the source code,
  - explicit vote.
- Feature branches are rejected, in case of
  - missing test cases,
  - missing documentation or comments,
  - wrong code style.

After the feature branch has been merged into the master an automated test suite runs on the master. If successful, the suite builds a release package, ready for deployment.

The distributed version control software *git* [7], and the code review server *Gerrit* [8] are used to support this. The basic workflow is illustrated in Figure 1. If a new feature is to be developed, an engineer forks the master into a feature branch. If there is a second new feature they want to work on, the engineer has to fork a second feature branch off the master. After developing the code, the engineer uploads the differential to the master, called a patch, into the code review system, and everyone on the team is notified that there is a new patch ready for approval. The patch upload only succeeds if the automated code checker *flake8* approves the patch [9]. This checker requires that every source file follows the same stylistic guidelines, for example, no whitespaces at the end of lines or unused imports of libraries. If the check fails, the engineer has to rewrite the source code until it is accepted. There is no way around this and it enforces a common coding style. Every engineer in the team now pulls the patch into their local

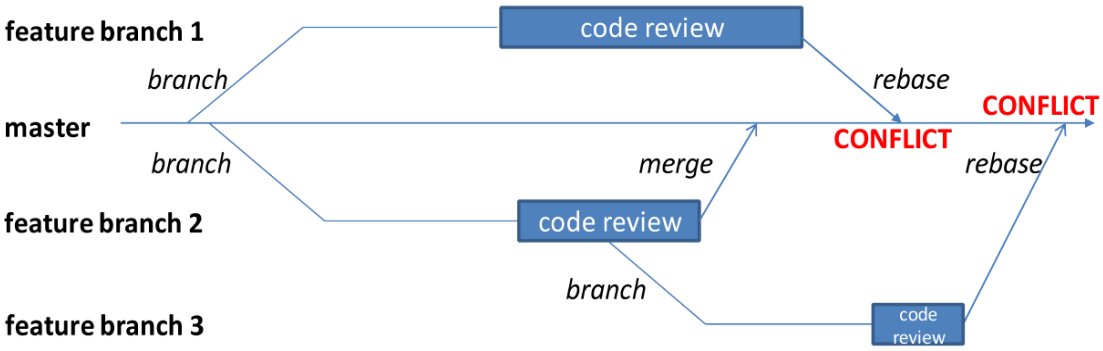

Figure 1: Development model: Feature branches are reviewed separately and merged into the always releasable master. Conflicts must be rebased.

development environment and begins testing the patch. First, this requires that the source code is read. This can be done on the web frontend of Gerrit, which displays the modifications in a split view. Comments on the source code can be inserted directly in the web frontend. Second, the testing itself is based on unit tests, against the central libraries, the API, and the clients. There are no unit tests for the daemon processes, but they can optionally run in single iteration mode, such that they can be tested manually. In all, there are 190 unit tests, with an average full serial test duration of 105 seconds against the slowest database backend, SQLite. The faster supported databases, MySQL, PostgreSQL, and Oracle, can run the full test in 90 seconds. If the test is successful, the engineer can vote +1, 0, -1 on the patch. If a patch gathers at least +2 votes, anyone can merge the patch into the master. If there is at least one negative vote, regardless of the amount of positive votes, the patch cannot be merged. In this situation, the engineer modifies the source code accordingly and uploads a new patch set. Gerrit automatically deduces that the new patch set is only a modification of an old patch and handles this appropriately. If the engineer wants to continue working on a feature, while their patch is in code review, they can fork a second feature branch of the already existing feature branch. This is sometimes needed, for example, if an API has to be provided for another part of the system, but the implementation of the API is not yet ready. This will eventually lead to conflicts in the master, as the same source files are changed in two different ways, with different reviews, or possibly by two different persons. This requires rebasing the patch. This means that the engineer gets the differential of the new state of the master, and their own patch, and has to select which parts of the patch to apply

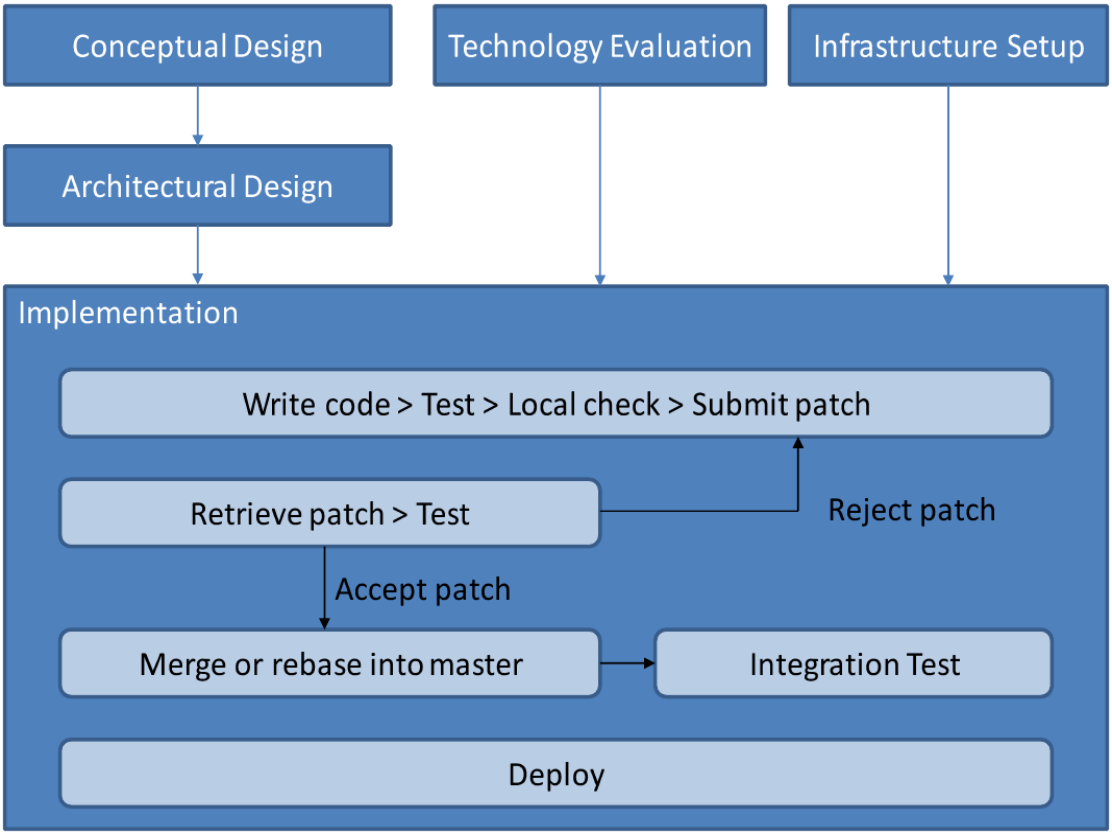

Figure 2: The full process: requirements engineering goes in parallel with technology evaluation and infrastructure setup. Implementation follows an independent and parallel test-driven approach with always-releasable source code.

to the master. This is necessary, for example, if a problem has been solved in the master that existed at the time of forking the feature branch. If the engineer temporarily fixed the problem in their feature branch, they can now select the proper solution from the master as the authoritative solution. If the patches are non-overlapping inside the same source file, then git can automatically rebase the patch without user interaction.

Once a patch is merged or rebased into the master, an automated test is run in addition. This downloads the source code from the master, prepares a fresh installation with a newly initialised SQLite database, and runs all unit tests. In principle, this automated test should never raise an error. If it does raise an error, notifications are sent out to all engineers that the master is broken, with the according log files. Until now, this has only happened twice. Once through accidental corruption of the Gerrit backend due to a malfunction of the hosting machine, and once on purpose while adding support for a second database because two engineers were working on different database backends with conflicting interfaces.

A summary of the whole process is illustrated in Figure 2. The requirements engineering phase occurs in parallel to the Technology Evaluation and Infrastructure

Setup. The implementation itself consists of three independent blocks, development, approval, and deployment, where each block can be tackled by any engineer at any time.

4. The human factor

This strict model of development has several social implications. Before introducing this model, several critical points came up, most importantly:

- Won't this take a long time to get my changes into the source code?
- What happens if I need to quickly fix something?
- I really don't like that someone else is judging my work.
- Writing test cases is bothersome, and no one uses them anyway.
- We will never agree on a common coding style – everyone is different.
- Branching? Merging? What is this?
- How do we deal with new members in team? How will we educate them?
- How can we deal with outside contributions?

The most important fear was that this is actually a slow process. This was true in the beginning as the branching strategy was not clear. Only after a few introductory sessions on the usage of the git and Gerrit tools, the feature branch strategy was adopted. In this strategy, the notion of *slow process* essentially disappeared. Submitted code was already written, but comments and reviews were ultimately helpful in only getting the highest quality code accepted into the master. It took about 1 month to converge on this workflow with the most basic problems evened out, for example, committing patches from a wrong branch, accidentally voting positively on bad patches, or not having the automated code checker enabled. After this introductory period, newly submitted patches rarely contained semantically or syntactically wrong code, which in turn improved the throughput of patch submission, revision, and acceptance. Even temporary team members, for example, summer students, were able to be productive in this environment within a day.

The issue of code review itself was surprisingly well-received afterwards. Every team member judged objectively the code, and never the person writing the code, ultimately leading to every team member knowing everything about the code base. This allows every team member to work on every component, even if it was originally not written by them. Additionally, due to the conceptual design document and architectural design document, the *big picture* was clear, therefore every engineer can potentially take over any component for future developments.

The last critical issue was stressful periods, where parts of the code had to be changed quickly in an already deployed testbed, for example, to fix a critical bug that would cause a database lockup. In this case self-approval of the patches in the Gerrit, circumventing the code review, was used.

Summarised, out of more than 1400 patches, less than 1 percent was self-approved. The mean time to merge a perfect patch is approximately 4 hours, for patches requiring one iteration approximately 1 day, and for patches requiring more than one iteration approximately 7 days. Only 12 patches were rejected, however more than 100 patches were abandoned. These patches mostly represent temporary features when integrating with third-party software, and are not needed anymore.

5. Conclusion

The development of Rucio has followed a different strategy than other software projects in the ATLAS distributed computing community. Instead of an iterative *discuss and approve-by-committee* process, a more strict test-driven process has been enforced. These tests have been derived from the initial requirements engineering and design phases. The results show that it is possible to provide high-quality software from the beginning, with clear testing scenarios to fulfil all required use cases. The initial delays due to social uncertainties and undefined conduct of the engineers were quickly dismissed, and proper procedures and conducts were defined. This allowed high throughput when developing source code, easy injection of new engineers, including temporary ones, into the team, and eventually a working software system.

Bibliography

[1] ATLAS Collaboration, *ATLAS Technical Proposal*, 1994 Technical Report CERN/LHCC/94-93
[2] M Branco, et al., *Managing ATLAS data on a petabyte-scale with DQ2*, J. Phys.: Conf. Ser. 119, 062017, 2008
[3] WLCG Collaboration, *Worldwide LHC Computing Grid*, http://lcg.web.cern.ch/, 2011
[4] ATLAS Collaboration, *ATLAS Computing: Technical Design Report*, 2005 Technical Report ATLAS-TDR-017
[5] V Garonne, et al., *The ATLAS Distributed Data Management project: Past and Future*, J. Phys.: Conf. Ser. 396, 032045, 2012
[6] M Barisits, et al., *Rucio: Conceptual Model*, Technical Report, ATL-COM-SOFT-2011-030, CERN, 2011.
[7] L Torvalds, et al., *git*, http://git-scm.com/, 2005
[8] S Pearce, et al., *Gerrit*, https://code.google.com/p/gerrit/, 2009
[9] T Ziade, et al., *Flake8*, https://bitbucket.org/tarek/flake8/, 2010