CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY AT FRIB*

M. Konrad†, D. Maxwell, G. Shen
Facility for Rare Isotope Beams, Michigan State University, East Lansing, MI 48824, USA

Abstract

Development of many software projects at the Facility for Rare Isotope Beams (FRIB) follows an agile development approach. An important part of this practice is to make new software versions available to users frequently to get feedback in a timely manner. Unfortunately building, testing, packaging, and deploying software can be a time consuming and error prone process. We will present the processes and tools we use at FRIB to standardize and automate this process. This includes use of a central code repository, a continuous integration server performing automatic builds and running automatic test, as well as automated software packaging. For each revision of the software in the code repository the continuous delivery pipeline automatically provides a software package that is ready to be released. The decision to deploy this new version of the software into our production environment is the only manual step remaining. The high degree of reproducibility as well as extensive automated tests allow us to release more frequently without jeopardizing the quality of our production systems.

INTRODUCTION

FRIB [1] is a project under cooperative agreement between US Department of Energy and Michigan State University (MSU). It is under construction on the campus of MSU and will be a new national user facility for nuclear physics. Its driver accelerator is designed to accelerate all stable ions to energies >200 MeV/u with beam power on the target up to 400 kW [2]. Commissioning of the front-end is currently underway and the remaining parts of the accelerator are planned to be commissioned over the next two years.

FRIB's controls group strives to support commissioning and operation by rolling out bug fixes and new features as fast as possible. To make this happen we are following principles of agile software development which include iterative, incremental and evolutionary development and a short feedback and adaption cycle. Unfortunately this approach can be slowed down considerably by the fact that building and deploying control-system software can be a complex and error prone process that often requires considerable manual work by experts. In the following we will describe how we speed up the build and deployment process for FRIB's controls software by following continuous integration (CI) and continuous delivery (CD) principles.

* Work supported by the U. S. Department of Energy Office of Science under Cooperative Agreement DE-SC0000661
† konrad@frib.msu.edu

CONTINUOUS DELIVERY

The process of building and deploying software generally consists of a series of tasks that can be thought of as a pipeline. Figure 1 shows the steps of a typical software build and deployment process as it has been implemented at FRIB. Each task in this pipeline is carried out after the preceding step has been completed successfully. In the following we will describe each of these steps in detail.

Revision Control

All source code required to build software for FRIB's accelerator control system is stored on a central Git [3] repository server. The repository server is running Atlassian Bitbucket Server [4] which, in addition to basic Git server functionality, provides a web interface, pull requests and branch permissions.

Our Git work flow largely follows the Gitflow [5] approach which requires developers to implement new features or bug fixes on feature branches allowing them to work on their feature without the risk of breaking other developer's build. If however the number of feature branches becomes too high and feature branches live for too long merge conflicts are becoming more likely. To reduce time-consuming conflict resolution we are following the practice of CI which requires feature branches to be merged into a shared mainline frequently. CI principles generally recommend branches to be merged at least once a day. In our experience many controls projects have a rather low rate of change or a very low number of developers making a life time of a few days feasible with an acceptable risk of running into merge conflicts. For critical code we use pull requests as a tool to facilitate code reviews. The goal still remains the same: Code reviews and possibly required rework should be performed timely so that feature branches can be merged into mainline as fast as possible.

A significant amount of FRIB's control system software is developed in collaboration with other laboratories with its source code being tracked in a an upstream repository on the Internet. In this case our CI server mirrors the mainline branch of the upstream repository into a branch in our local repository on a regular basis. Following CI principles, we are merging upstream changes into our own mainline branch as soon as possible to keep merge conflicts with our feature branches to a minimum. In general we are aiming to keep the number of FRIB-specific modifications to a minimum. Instead we prefer to contribute our improvements back to the upstream project. This reduces the risk of merge conflict in our repository and thus reduces the maintenance effort in the long run. At the same time this approach allows us to fix critical bugs in our local repository until they are fixed upstream.

ISBN 978-3-95450-189-2

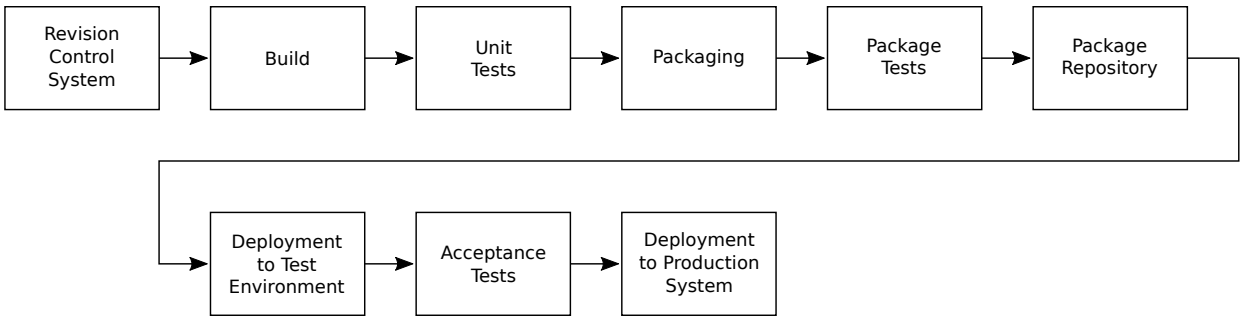

Figure 1: Overview of the FRIB continuous delivery pipeline.

Build

Building after each commit/merge is a best practice that ensures build issues are caught as early as possible. Frequent integration of feature branches into the mainline branch means the mainline code needs to be built frequently. At FRIB software is built automatically by a CI cluster running Jenkins [6]. This cluster consists of two Linux build nodes, a Windows build node, and a build node for FPGA projects. The two Linux build nodes can each run four jobs in parallel and are thus capable of completing builds within an acceptable time even at times of high load. A separate Jenkins master machine orchestrates the software builds by distributing them to the appropriate build nodes based on build requirements and load. All machines are virtual machines which facilitates adding resources or more build nodes.

Build logs are being archived for each build along with the artifacts that resulted from a successful build (e. g. binary files/package files). This data is available to developers via a web interface. For each successful build the resulting artifacts are passed on to the next step in the pipeline.

Unit Tests

Developers need to ensure their unit tests can be executed in a standard way (e. g. by running `make test`). Tests that have been set up in this way are automatically executed after the software has been built. Test results printed to standard output are archived by Jenkins as part of the build log. Test logs that are compliant to well known standards can also be visualized by Jenkins. For packages that do not come with unit tests this step is skipped automatically.

Packaging

FRIB is using Debian GNU/Linux as the standard operating system for control-system computers. Software is packaged into Debian packages to facilitate installation, upgrade and complete removal. Packages also allow developers to define dependencies between software components. Possible incompatibilities with specific versions of these dependencies can also be reflected in these dependencies.

Debian packages are built using Jenkins Debian Glue [7] which adds Debian packaging capabilities to Jenkins. Packages are generated in a two-step process. First a Debian source package is generated from the source code stored in the Git repository. In a second step this source package is built in a "clean-room" environment consisting of a `chroot` sandbox. In addition to a Debian base system only packages that are defined in the list of build dependencies for the package are installed in the sandbox guaranteeing a reproducible environment. Each build starts with a fresh sandbox. By using copy-on-write techniques Jenkins Debian Glue is able to speed up the process of generating build environments significantly.

Binary packages are built for each entry in a predefined build matrix. This matrix can be configured for each project individually and can contain multiple target architectures as well as multiple Debian operating system versions. Being able to build packages for multiple operating system versions is important during a phase of operating system upgrades where some machines might already run the new version while others are still using an older version.

Package Tests

A set of automated tests are run on each Debian package to catch common packaging issues. These tests ensure the package can be installed, upgraded and removed cleanly.

Package Repository

Once a Debian package has passed all tests it is pushed into a Debian package repository. This package repository is being managed by aptly [8]. It lives on the Jenkins master machine and makes packages available for download via HTTP. The package repository uses signatures based on state-of-the-art cryptography to make it more difficult for attackers to inject malicious packages into the controls network. Packages from this repository can also be installed by Jenkins to satisfy build dependencies of other packages.

Management of Jenkins Jobs

For most job types the build pipeline is broken into multiple jobs on the CI server to make it more obvious to devel-

ISBN 978-3-95450-189-2

opers in which step of the pipeline an error occurred. For Debian packages the pipeline consists of the following jobs:

1. Build source package
2. Build binary package(s)
3. Run package tests
4. Push package to the package repository

Note that the source package is only built once whereas the remaining steps are being performed for all entries of the build matrix.

So far more than 700 jobs are defined on the FRIB Jenkins server; the majority are related to building Debian packages. Instead of setting them up manually they are managed using Jenkins Job Builder [9]. This tool configures jobs based on a set of templates and a short description of each project. This ensures all jobs of a certain type are configured the same way.

In addition to this basic job management we automatically generate triggers between pipelines belonging to different projects based on the dependencies defined in the Debian packages. This ensures that after a library has been built all projects that use this library are being rebuilt automatically as well. This approach helps to catch issues as early as possible. The dependencies are visualized by Jenkins in form of a dependency graph.

AUTOMATIC DEPLOYMENT

FRIB uses Puppet [10] for IT configuration management. This tool takes a description of the configuration of the target computers as an input and ensures these machines are configured accordingly. For most machines deployment is completely automated allowing to re-install machines from scratch within a few minutes.

Deployment to Test Environment

At FRIB controls applications are tested in a development/test environment before they are deployed to the production network. This test environment mimics the FRIB production network as close as feasible (similar network layout, running the same services etc.). Configuration of machines on the test network is also managed by Puppet. Deployment to the test environment follows the principles of *continuous deployment* which means that the latest version of each application is deployed to this network automatically.

Acceptance Test

For most applications a manual acceptance test is performed in the test environment. If this tests passes the corresponding packages can be deployed to the production environment.

Deployment to Production Environment

Deployment to the production network follows the principles of *continuous delivery*. Each code version is being built into a package which potentially can be released to the production environment. However, the release process itself requires a manual decision. This decision is based on the results of the acceptance test as well as on the operational needs of the accelerator. For example deployment can be postponed until the next maintenance shutdown.

SUMMARY

FRIB's build and deployment process has been automated successfully. This allows us to support commissioning and operation of FRIB by applying an agile development work flow with short adaption cycles. The automated processes have improved reproducibility significantly. Full traceability makes it easy to find out which version of the source code has been used to build a certain package. It also allows developers to inspect the corresponding build logs and intermediary build results making troubleshooting much easier. CD makes it easier for developers to contribute to many different projects since no expert knowledge is required to build and deploy the software. This facilitates team work.

The biggest challenge we encountered while introducing CD is a lack of test automation which can lead to a lack of developer confidence. In a similar way users can feel uncomfortable with software being upgraded frequently, especially during critical parts of accelerator operation. Until test coverage has been improved this can be overcome by careful scheduling of upgrades.

REFERENCES

[1] FRIB, http://www.frib.msu.edu

[2] J. Wei *et al.*, "FRIB Accelerator: Design and Construction Status," in *Proc. 13th Int. Conf. on Heavy Ion Accelerator Technology. (HIAT'15)*, Yokohama, Japan, September 2015, paper MOM1I02, pp. 6–10.

[3] Git Distributed Version Control System, https://git-scm.com/

[4] Atlassian Bitbucket Server, https://bitbucket.org/product/server

[5] V. Driessen: A successful Git branching model, http://nvie.com/posts/a-successful-git-branching-model/

[6] Jenkins Automation Server, https://jenkins.io/

[7] Jenkins Debian Glue, http://jenkins-debian-glue.org/

[8] aptly Debian Repository Management Tool, https://www.aptly.info

[9] Jenkins Job Builder, http://docs.openstack.org/infra/jenkins-job-builder/

[10] Puppet Configuration Management Tool, https://puppet.com/

ISBN 978-3-95450-189-2