HCAL automated software building and deployment with GitLab

David Southwick

Abstract

This report details the creation and implementation of continuous integration practices within the Hadronic Calorimeter (HCAL) subdetector of the Compact Muon Solenoid (CMS) at the European Organization for Nuclear Research (CERN). Steps detail migration of detector operations code base from SVN to GIT, followed by creation and development of a new Continuous Integration (CI) workflow for developers.
1 List of abbreviations

HCAL Hadronic Calorimeter
CERN European Organization for Nuclear Research
CMS Compact Muon Solenoid
XDAQ CMS Cross-DAQ
SCM Source Code Manager
SaaS Software-as-a-Service
RPM RedHat Package Manager
ROOT CERN data analysis framework
amc13 13 slot Acromag telecom crate
CI Continuous Integration
2 Introduction

At the Hadronic Calorimeter (HCAL) subdetector group of the Compact Muon Solenoid experiment (CMS) at the European center for Nuclear Research (CERN), the first Continuous Integration practices have been introduced and adopted by HCAL Online Computing group. These changes include a necessary change to source code storage, distribution, and daily development routines. Here are reported the details of this workflow change as well as the demonstration of the resulting products of the new Continuous Integration process to produce physics data in a stand alone (separate from CERN centralized services) environment. This work was realized over a six month period with the HCAL Operations group.

3 Workflow Changes

CMS HCAL’s detector code base consists of application-specific implementations of proprietary data acquisition framework CMS Cross-DAQ (XDAQ) \[3\]. The C++ code developed using the XDAQ framework is split thematically amongst eighteen packages that address different aspects of the data acquisition pipeline specific to HCAL. 375,000 lines of code are compiled and built into RedHat RPMs for installation across various server hardware responsible for data acquisition. The same software packages are additionally made available for independent hardware development installations via centralized repository. Maintaining the HCAL code base and repository has been a non-trivial custodial task for physicists and engineers since its creation. A delicate software build workflow coupled with the high frequency of new student and staff collaborators made it of interest to examine methods to simplify and automate this task.

To explore this goal, current industry standard practices in collaborative development environments were examined; specifically where a Source Code Manager (SCM) is used. It is clear that there are ongoing trends for distributed, independent development, and shorter development cycles that promote rapid development feedback.\[1\]\[7\] Similarly, the introduction and uptake of containerized services, or Software-as-a-Service (SaaS) is a trend that will likely continue. Accordingly, European Organization for Nuclear Research (CERN) central IT has followed these trends by moving historically dedicated CERN services to simpler, steady-state containerized versions.\[2\] Conducting a feasibility study of available Continuous Integration (CI) and development solutions (GitHub vs GitLab) was the first step toward simplification of the HCAL software development operations.

3.1 Choice of Continuous Integration platform

Requirements for a possible CI framework were collected from members of the HCAL online computing and operations group, the body responsible for the DAQ operation within the subdetector group over several meetings. The primary requirements identified for creating a CI system were that it be easy to use with minimal training, that it integrates with either SVN or GIT source code managers, that it support automated and traceable build pipelines for HCAL software, and that it provide access control that would comply with CERN computing account permissions. Surveying tools available identified two solutions that could provide a complete CI solution: Jenkins and GitLab.
Initially, Jenkins build system was proposed as a minimal complement to the already existing HCAL SVN repo. Jenkins, the open source successor to the Hudson build system from Oracle, is a mature build framework with an extensive external plugin ecosystem with many additional features available [5]. Jenkins, however, does not provide a hosted solution - it is left to the user to setup their own instance host. To determine the feasibility of automated building, a prototype Jenkins installation was configured to use the HCAL SVN code repository. The base installation was straightforward, however the configuration and customization was very complex. Every feature required by HCAL was only provided via external plugin that required its own configuration. The amount of setup work Jenkins requires for a transparent build-on-commit CI experience is rigorous, and the maintenance of the resulting landscape of independent plugins only compounds this. Setup and maintenance aside, the granularity provided by such a modular system does afford much greater flexibility and customization, such as support for additional source code managers.

GitHub, originally a simple Git repository hosting service, has grown to become an industry leader in repository hosting. The addition of issue trackers and external plugin triggers, has matured the site into a popular project management service. Many HCAL affiliated institutions continue to choose GitHub public hosting as their primary code hosting over CERN provided resources, indicating increased familiarity with GIT version control, and a desire for additional platform utility beyond simple SVN access. The project management advantages of GitHub can be combined with Jenkins to provide a full service build automation, publishing, and development platform, giving HCAL contributors a choice of tools to use [6]. The only unresolved requirement that remains was access control, which GitHub unfortunately only licenses to private enterprise instances.

During this prototype development of Jenkins CI, CERN central IT services began offering a GitLab pilot service for evaluation. GitLab, a much younger and less mature product than GitHub, is an open-source SaaS solution which includes all the features of a GitHub and Jenkins duo combined in one service, along with many of the paid subscription functionality of GitHub (such as integration with CERN computing accounts) for free. Considering the number of HCAL collaboration projects already using GitHub for its GIT service, the choice to setup a GitLab instance for comparison was obvious.

Setup complexity and workflow flexibility are the highest interest, as this workflow would need to be both powerful enough for experienced developers, yet accessible to newcomers. To compare Jenkins build automation against GitLab, the steps required to recreate the manual build process HCAL had been using to compile and publish their software were recorded and compared on each platform. Examining the build requirements, both platforms required relatively the same effort to setup the requisite Docker build environment for compilation, and the description of the individual build steps are both performed in comparable markup languages. Both services are straightforward in description and ease of use to publish build artifacts (RPMs) to production servers, and traceability (including log access) of the build process is equally accessible. The major differences between the two are in the housekeeping. GitLab, being a unified service, describes the build process in a single text file that is stored as part of the repository alongside the source code. Jenkins stores these build instructions externally, within the Jenkins application, split amongst its various plugins. One major advantage of the GitLab method of storing the build instructions in the repository is that novices and experts alike can modify, contribute, and improve the workflow to their liking, as
its access permissions are controlled the same manner as any file within the repository. Since the build description is version controlled, a historical documentation of the build process is automatically recorded. Additionally, describing the build stages for parallelization and across different architectures can be generalized with GitLab, as the context of the job can be extracted from the git branch being built. Replicating such flexibility requires duplicating job descriptions in Jenkins - creating a complex environment to maintain for future changes. Most significantly, the additional workload introduced for Jenkins configuration maintenance reduced the advantages gained by automating the workflow, due to its administrator-user model. Such maintenance can be safely made available to all developers, and collaborated upon using simple git commits, to better leverage the expertise of all. The decision was made to move ahead with GitLab.

However, selecting GitLab created a requisite task of translating the HCAL SVN repository into GIT.

### 3.2 Source control management translation

Like GitHub, GitLab is designed to work with GIT repositories. This required a faithful translation of the HCAL SVN repository, respecting the full historical information of the organization. Both SVN and GIT provide the same fundamental version control functionality, but they approach the task in different manners, and offer very different atomic permission models. One of the core differences between them is that GIT offers an advanced tagging system based on the commit hash. This is maintained across branches and forks. SVN tagging is simpler; a complete duplication of the code to be tagged is copied into a branch simply named "tags". Translating this tagging branch into GIT hash-based history was accomplished using the git add-on tools built for import from subversion, and lots of patience. The mapping of the original SVN committer’s computing identity to relative commit hash identity was mapped and verification of the migration was performed using depth traversing scripts to compare commit time and author with diff contents. Overall, the SVN translation imported 7,273 commits to GIT while maintaining attribution to the original authors work.
3.3 Continuous Integration workflow

During SVN era of HCAL detector code, each package within the repository was built manually by command-line scripts. While package building may be a non-complex task itself for experienced users, the expected learning curve for a visiting physics collaborator is demanding. Given the diversity and number of contributing members from different universities, a simpler, more accessible and verifiable way to build and test one’s code contributions was required.

Automated building and testing of code has been commercially available for several years, resulting in numerous platforms for automation of development tasks. HCAL chose to adopt GitLab Flow as a model for code development. From this model, HCAL would maintain two branches (Integration and Release) per supported operating system. All development work would be done on a developer’s private fork of the repository, and improvements could be requested for merging into the master Integration branch from a developer’s private branch. From here, at regular intervals or significant project deadlines, the master Integration branch may be merged into Release branch, tagged with a release number, and the resulting RPMs published using a GitLab pipeline deploy action described in the pipeline file.

3.3.1 HCAL build pipeline

At its most basic, the HCAL build automation is comprised of four stages: Pre-check, Build, Test, and Publish. These stages and their sub steps are described in the special pipeline description file (gitlab-ci.yml) that is version controlled as part of the repository. Upon commit to the repository (or to a forked copy), the build pipeline file is read and executed until completion, or failure. A report email detailing the result is issued to the user responsible for the commit (and thus the pipeline). The detail of the stages are as follows:

Pre-check performs a comparison of the commit against the most recent release tag in the Release branch. Due to the package dependencies in HCAL code, this checking stage explicitly examines any changes present for resulting dependencies that would require updating to result in a build that will pickup all the changes introduced in the dependent packages. If any of these checks fails, the build is marked as a failure, and the pipeline stops.

The Build stage starts after a successful check of internal dependencies, and proceeds to build each package using inside a custom Docker build environment which contains the necessary XDAQ, CERN data analysis framework (ROOT), and 13 slot Acromag telecom crate (amc13) build dependencies. The steps for the build stage are to descend the dependency tree of the project breadth-first and compile each package individually. Upon success, each resulting set of libraries is collected and made available to downstream jobs, and additionally compiled into a package RPM. On success, the stage is marked a success, and the resulting build artifacts (the RPMS) are collected, and the pipeline moves to the testing stage.

Testing is currently an optional stage which produces a static C++ code analysis report. This is done by calling a custom Docker image which has static code analysis tool installed into it, which analyzes the entire repository and publishes the report to an HCAL documentation server. This stage has much improvement possible for the future - for example unit tests and code coverage are completely unused at current. This stage passes even if error
flags are encountered. Finally, if all previous stages are successful, the final stage is a manually initiated stage that publishes the RPMs from the Build stage to the HCAL repository, and produces the automated installation scripts that describe the current release. This stage can only be initiated by members of a specific HCAL LDAP group, and uses shared SSH keys that will only authenticate from the upstream (main) HCAL Release branch, and not on any forked copy.

The description of this pipeline and its implementation is distributed as part of the repository. Thus, any member that forks a copy of the HCAL project receives in their fork a working automated build system that they are free to play with, modify, and contribute improvements through Merge Requests. This new access to immediate feedback to code contributions is a first for HCAL, and will hopefully lead to many improvements in the future.

References


[4] Introduction to GitLab Flow — GitLab. URL: https://docs.gitlab.com/ee/workflow/gitlab_flow.html
