Navigating the Rapids: Creating a Continuous Integration System for Github and Jenkins

A whitepaper for the 2020 Collegeville Workshop on Scientific Software, focusing on Developer Productivity.

Aaron L. Levine, Jim Willenbring, *Sandia National Laboratories*

There are many stories of projects benefitting greatly by implementing Continuous Integration (CI) solutions. Most describe dramatic reductions in test time, improved developer productivity, and cleaner code. What happens, however, when the publicly available CI solutions, such as Travis, Appveyor or Github Actions cannot meet the needs of the project? How does a project team move forward with implementing a CI solution? Many teams will decide that creating their own solution is the best course of action. Oftentimes, what these teams do not realize is that the software engineering methods used to develop these solutions can dramatically affect the development cost and maintainability of such a solution. This white paper will discuss the development of just such a solution called the "Pull Request Autotester", its implementation difficulties, multiple expansions, and lessons learned.

The Genesis of the Pull Request Autotester

The Structural Simulation Toolkit (SST) team at Sandia National Laboratories develops simulation technologies to research next-generation high-performance computer (HPCs) systems. The codebase consists of four main repositories and is designed to run on a variety of systems ranging from Linux-based desktop workstations to current generation HPCs. Until 2015 the SST team was using Subversion (SVN) for version control, with all development in a single branch, and struggled to keep a consistently stable and buildable codebase. The repository was open source, allowing researchers around the world to access the codebase. Frequently developers would unwittingly push broken code into the repository that caused nightly tests to fail. Recovery from this broken state usually took upwards of 3 days to correct. Meanwhile, developers and researchers who merged the broken code were blocked from further development until the issue was resolved. The probability of SST repository being "broken" on any given day was between 50-60%; therefore, most developers distrusted the repository and would not integrate changes unless absolutely necessary.

In 2015 SST migrated its codebase from Subversion to Git (hosted on Github). During this time, the SST team changed its workflow to a gitflow process and decided to implement an automated Continuous Integration (CI) system using Github's and Jenkins' API interfaces.

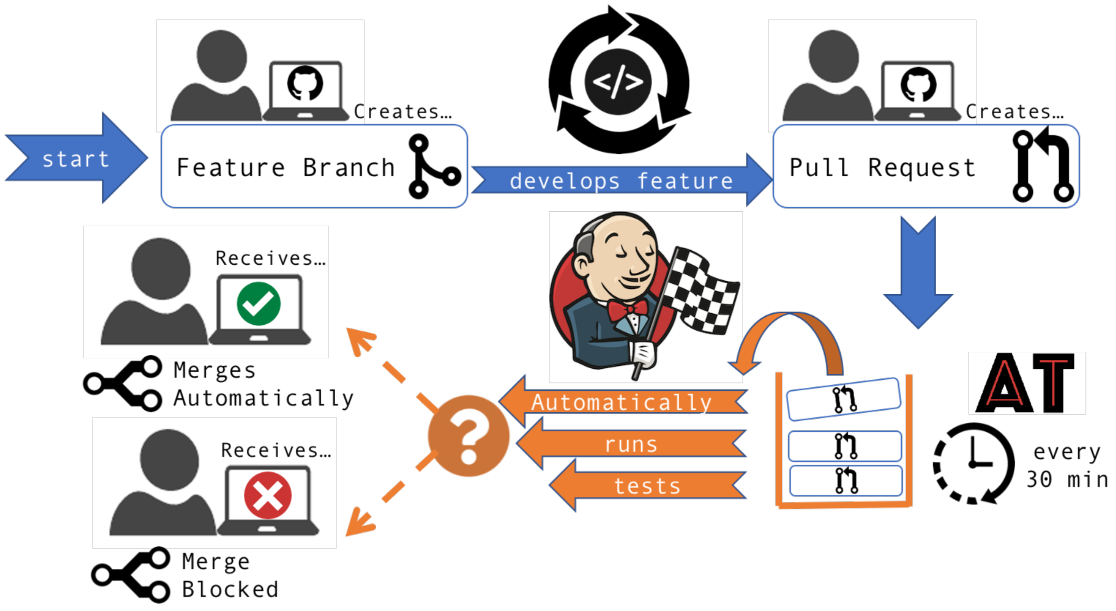
Initial Design Objectives

The development of the first generation Pull Request Autotester (Autotester) for SST started as a simple discussion of *"How can we regularly test our code **before** we commit it to our repository?".* That discussion initiated a tremendous number of changes for SST. First was to implement the gitflow workflow using two main branches (`master` and `devel`), followed by the implementation of the Autotester. As part of the workflow, we wanted to use Github's branch security features to protect our `master` and `devel` branches. Additionally, the SST team was heavily invested in using Jenkins for its testing platform and wanted to continue leveraging it. As a result, integration with Jenkins was also required. From this information, an initial set of "loose" requirements for the Autotester were generated: - Written in Python. - Identify Pull Requests targeting the `devel` branch. - Run one or more Jenkins jobs against the source branch of the Pull Request - Automatically merge the Pull Request if the tests pass.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2020-6631 C

The desired workflow for developers is described in the diagram below:



Implementation

The initial development efforts were to explore many of the 3rd party Python modules that communicated with both Jenkins and Github. The modules JenkinsAPI and PyGithub were chosen due to features, stability, and active development status. A number of small experimental test programs were created with these modules to get a sense of the operations required and how to properly use them.

Once we had a fair understanding of how to interact with Jenkins and Github, we made the following implementation decisions:

- Python Version - We targeted Python 2.7 due to the perceived (at the time) instability of Python 3.
- Configuration files - Configuration of the Autotester would be controlled by simple key/value structured configuration files.
- Logging - To facilitate tracking and debugging of Autotester processing, a generic logger would be created.
- Polling Github for Pull Requests - The Autotester would poll Github rather than using a push solution.
- Github configuration:
    - Repository branch protection - We configured our `master` and `devel` branches as read-only.
    - Pull Requests - All changes to the `devel` branch were to take place via Pull Requests.

After the initial set of “loose” requirements, no further effort was undergone to make a formal set of requirements. At the time we had a general idea of what we wanted to do, but we did not realize all of the nuances and small details required to implement. One example was that it sometimes took multiple attempts to query Github on the mergeable status of a Pull Request before getting a correct response. We opted for the “shoot from the hip” approach and created the Autotester “on the fly,” adding features and non-reproducible “one-shot” tests to prove functionality as needed. In retrospect, this caused repetitive rework of the Autotester code to implement new requirements that we likely could have anticipated had we practiced better requirements management.

Results from the Initial deployment

The development of the initial Autotester took about three months, and the changes after initial deployment took another three months. These 6 months were quite chaotic as many changes were being made to both the workflow and Autotester at the same time. Most of this time was due to the "loose" requirements for the Autotester and inexperience of the team.

About six months after the final changes to the Autotester, the SST project had made huge strides. The team had fully integrated the workflow and the Autotester was in production. The result of this was that the stability of the codebase had dramatically improved. At this point, both the `master` and `devel` branches were stable and buildable approximately 99% of the time. There were occasional failures that went unchecked by the Autotester, but those were due to coverage issues with the Jenkins CI tests and were quickly corrected. While many of the developers were initially resistant to the Autotester, they have since become strong supporters of its usage.

The Second Generation Autotester

Approximately one year after the deployment of the first generation Autotester, the Trilinos frameworks team expressed interest in the Autotester. Trilinos was experiencing similar difficulties as SST: developers submitting broken code to the codebase, many nightly tests failing consistently, and constant human interaction required to manage the workflow.

In order to support the Trilinos project, significant changes to the first generation Autotester were required. These changes were deemed the "second-generation" version of the Autotester and required efforts to maintain backwards functionality for SST. Again, non-reproducible "one-shot" tests were created to verify Autotester functionality. Some of these changes required were:

Generic Project Support: The Autotester had a large amount of code that was SST specific. There was a considerable effort to extract this code and make the Autotester "project agnostic".

Testing Throughput and Robustness: Trilinos was handling approximately ten times as much Pull Request traffic as SST, and its test time was about three times as long. Significant effort was required to handle the testing demands of Trilinos. Additionally, The PyGithub and JenkinsAPI modules had no low-level communication retry features. Complex solutions were required to "harden" the interfaces so that spurious network glitches would not break operations.

New Features: Many numerous new features were implemented to support the needs of the Trilinos project. While these were great improvements to its operation, they were sometimes quite difficult to integrate and required significant re-writes of code.

Once deployed, it took roughly as much time for the Trilinos team get used to the newer workflow imposed by the Autotester as it had for the SST team. About six months after the deployment of the second generation Autotester, the Trilinos project had made tremendous progress. Its codebase was significantly more stable and no longer has tests failing for long periods of time.

Difficulties Encountered in Development

The Autotester implementation had a number of difficulties. Inexperience and testing with Github were some of the primary issues. However, once the Autotester started to be used in production, many new requirements were discovered, and this caused significant re-writes of portions of the code.

Developer experience: The main developer of the Autotester was learning Python while developing the Autotester at the same time. This made it very difficult to implement a large complex application. Much of the existing Autotester code is not pythonic due to this inexperience.

Github knowledge: Since the SST team was quite new to Github, there was a large learning curve to understand Github's operations, features, and limitations. The team had to learn how to use Git and Github, a new workflow process (gitflow), and how to integrate the Autotester into that workflow. During this time, many of the new requirements for the Autotester were being generated as the SST team realized the power of the tool and how it could improve the workflow process.

The main developer was also new to Git and Github. This compounded many of the challenges in developing the Autotester as many Github features were being discovered during Autotester's development. Essential and useful features, such as Github statuses, were not recognized until well into the Autotester development.

Testing difficulties: Since the SST repositories were under active development, testing of the Autotester had to be completed using forked repositories and separate user accounts. Quick and dirty implementation tests were created on the fly to test the various features. These tests were one-shot configurations and were discarded as soon as the feature was working.

Code Inspections: The team realized that different developer roles required different types of code inspection. This required significant changes in the Autotester to support appropriate inspections of code submitted by Pull Request before testing. Github provided features for reviewing code, but at the time, PyGithub did not yet support those features. The developers had to create and submit changes to PyGithub, which caused a significant delay, in order to implement inspection of code before testing.

New requirements causing change to the existing design: After the initial version of the Autotester was deployed and while the second generation Autotester was created, a large number of new requirements were identified. Implementation of requirements usually required some refactoring of the code.

Additional applications: As the teams realized that more automation could be used, there were requests for other applications to automate the workflow. For example: Users began to question, "Could we do this automatically if all of our nightly tests pass?" This caused us to refactor a large portion of the code into a set of common python files so that we could create a new application called `masterautomerge`. Several other additional support applications were created using the modules of the Autotester: such as a nightly report of the overnight tests and weekly report of all issues in the Github repositories.

Jenkins CI tests were not providing enough coverage: While not specifically a feature of the Autotester, the SST team had to create a number of CI tests that the Autotester would be running. Initially the objective was to have tests provide a quick turn-around, simply returning a pass/fail result to the user. We discovered, however, that these short tests did not provide enough coverage and were allowing a number of failures through to the `devel` branch. Eventually a set of parallel tests which took a slightly longer runtime were developed to provide the necessary coverage.

Lessons Learned

The development of the Pull Request Autotester, while ultimately a success, had a large number of difficulties that could have been avoided with more upfront planning:

Minimal requirements up front and lack of understanding of the operational environment: The Autotester started in the same way that more projects do: as an experiment to see if it could be done. We did not have a full understanding of the available features of Github and our 3rd party modules. Had a more detailed set of requirements been created upfront along with plans for growth, the development would have been smoother and provided a significantly more maintainable application.

Limited project scope: The initial Autotester was coded specifically for the SST project. Expanding its scope to support much larger projects resulted in substantial effort that could have been avoided had it been designed agnostically from the beginning. In this same area, not recognizing that future projects would require significantly more throughput in processing Pull Requests caused unnecessary rework.

Too few and inexperienced developers: There were two people involved in the initial development of the Autotester. One was generating concepts for features, and the other was coding the actual application. The coder was learning Python, Git, and Github all at the same time. This introduced significant floundering and non-pythonic code. For an application of this complexity, it would have been better to have two to three people cooperatively developing the product. Other developers could have had the opportunity to perform code reviews and would have mitigated the effects of inexperience.

Project customization: Every project had special needs that were not met by the Autotester. Providing some sort of a customizable "plug-in" infrastructure would have reduced the amount of rework necessary to make the Autotester work for that project and would allow the project team to tweak operations to meet their needs.

Test Driven Development (TDD): Currently it is daunting to consider adding new features and correcting issues in the Autotester. There is no existing test system for the Autotester, and developers fear breaking existing functionality. The original developer is the only one who has confidence to make changes due to his intimate knowledge of the code. Had the Autotester been developed using TDD techniques, the quick, one-shot tests would have actually been built into the test system, and a consistent and repeatable set of tests could be run on any code changes to protect against failures. The best way to say this is, "CI testing tools need to use CI testing techniques themselves."

Impacts to the team: The introduction of automated CI systems can cause considerable upset to a team's workflow. Methods to mitigate and slowly introduce the system into the workflow, rather than conducting a massive overhaul of their entire process simultaneously, allows users more time to adapt to the change.

Conclusion

While there are numerous CI solutions for Github (Travis, Appveyor, Github Actions), these generally do not function inside of a corporate firewall, which many national laboratories and private companies require for cyber security. Projects that develop their own solutions can take multiple approaches but developing such a critical piece of infrastructure is not an insignificant task and should only be undertaken using mature software development processes.

Currently there are four projects using the Pull Request Autotester, with a fifth soon to be online. The development was ultimately a success, but due to poor requirements management and development methods upfront, it was a significantly more challenging project than originally anticipated. Though the second-generation code is stable, no major features are planned for future development, due to the risk of breaking existing features. Had the Autotester been developed as a full-fledged product with a team of developers, defined requirements, and a mature development process, there is no doubt that the code would have been written better, easily maintainable, and quickly developed, which provides better possibility for introduction of new features.

Due to the importance of the Pull Request Autotester for Sandia projects and the desire to implement additional features, a new Autotester (third generation) is currently in the early phases of development. Lessons learned from the previous generations are being applied. There will be a defined set of requirements, the product will be preemptively designed for expansion and customization by other projects, the development team will include multiple developers, and it will be developed using Test Driven Development techniques. Had this been done from the beginning, this effort and its associated costs could have been avoided.