Restoring productivity through the advanced usage of Git

Roscoe A. Bartlett
bartlettroscoe.github.io
rabartl@sandia.gov
Department of Software Engineering & Research, Sandia National Laboratories

I. Introduction

The best software development processes are usually the simplest processes. However, there are situations where sticking with simple processes can lead to large reductions in productivity and delays due to difficult constraints and realities that exist in many projects. In many cases, adopting more sophisticated development and integration processes using advanced Git [1] workflows can restore much of the productivity that would otherwise be lost. The skilled usage of the Git distributed version control system and the exploitation of special properties of the problem at hand can often be used to work around the most difficult situations.

Below we provide one example where the more advanced usage of Git has been used to avoid significant reductions in development productivity. The goal is to provide some motivation and inspiration for more developers in the computational science and engineering (CSE) community to acquire a deeper understanding of Git and the building blocks for advanced Git workflows. Given these tools, they can develop similar customized workflows to help boost their productivity as well.

II. Background

At the foundation of modern agile software development processes are the practices of test-driven development, rapid-response peer reviews, and continuous integration (CI) of different streams of development [4], [5], [8], [10]. Modern Git-based hosting platforms like GitHub and GitLab provide solid foundations for modern agile methods. Development is done in small short-lived topic branches that are created off of the main development branch, called "trunk" (i.e. `master`, `main` or `develop`) [2]. The next set of changes in the next topic branch (by the same or a different developer) are then based on the most up-to-date version of `trunk`. This simple continuous integration workflow is depicted in Figure 1. This quick integration approach minimizes the occurrence of *textual*[1] and semantic[2] merge conflicts and allows the development team to move very quickly in an agile way. Merge conflicts can be a large cause of deterioration in development productivity.

With this simple CI workflow, pull requests (PRs[3]) are created for these small topic branches and a rapid code review by one or more peer software developers is done. Automated testing of the changes in the PR is also done on each PR using integrated testing services such as Travis CI, Circle CI, GitHub Actions, GitLab CI, or using custom continuous testing systems. Once a PR is reviewed and approved, and the automated testing comes back passing, then the PR is merged into `trunk`. This PR rapid-review and pre-merge testing process is a step up from the classic CI processes described in [5] and [8] where reviews really had to be done using pair programming and testing was only done post-merge (i.e. after the merge of the changes into `trunk`).

The definition of CI from XP [5] is:

> Integrate and test changes after no more than a couple of hours.

Also from [5], "The integration step is unpredictable, but can easily take more time than the original programming. The longer you wait to integrate, the more it costs and the more unpredictable the cost becomes." Therefore, the key goal for frequent and rapid integrations into `trunk` is to avoid merge conflicts. The role of testing the changes is to ensure that developers stay productive because pulling the updated `trunk` and experiencing broken code is highly disruptive and kills developer productivity. In order to facilitate frequent integration and maintain working software, one of the primary practices in XP [5] is the "Ten Minute Build":

> Automatically build the whole system and run all of the tests in ten minutes.

The argument is that 10 minutes is fast enough to remove any excuse for developers not to merge their work at least once a day.

SAND2020-6704 C: This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

[1] *Textual conflicts* are when lines in the files themselves conflict when doing a `git merge` or `git rebase` and requires manual intervention to resolve the conflicts in order to have have an updated Git branch

[2] *Semantic conflicts* occur when the branches merge just fine with Git but the software is otherwise broken due to inconsistent changes.

[3] Pull requests (PRs) in GitHub are instead called merge requests (MRs) in GitLab and have almost identical usages, features, and workflows and can therefore the term PR and MR used interchangeably. Therefore, for the reminder of this paper, the term PR will be used to mean PRs with GitHub or MRs with GitLab.

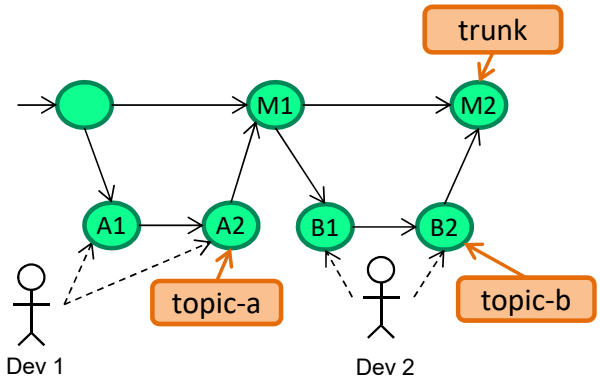

Fig. 1. **Simple continuous integration workflow with short-lived topic branches**: Developers make changes on short-lived topic branches that have a quick review by another developer and are quickly tested and merged to `trunk`. Another developer (or the same developer) then creates a new topic branch based off of the most up-to-date version of the code, thereby eliminating the chance of a merge conflict.

III. Real-world impediments to simple fast continuous integration

For many projects, there is a tug of war between the frequency of integrations with `trunk` and how much testing one does to ensure the stability of the software before merging to `trunk`. As the level of testing is increased (i.e. adding more build configurations and more platforms) the wall clock time to run the tests increases and that delays the integrations with `trunk`. As the integrations with `trunk` become less frequent, the greater the chance for expensive merge conflicts when integrating to `trunk` or the more delay that occurs for follow-on work. Alternatively, with less testing, integrations can occur more frequently which reduces the probability of experiencing merge conflicts or delaying follow-on work. But with less testing, the chance that broken code will be merged to `trunk` increases which then delays the detection of defects and can damage the productivity of developers who may experience broken software. As time to detect defects goes up, the cost to fix defects grows exponentially in many cases [9]. Detecting, triaging and debugging defects is a very expensive activity which kills the productivity of the development team and slows their progress in implementing new features. So ultimately, the trade off between merging more frequently to `trunk` versus doing more testing before merging to `trunk` comes down to balancing the costs of dealing with merge conflicts and delays in follow-on work with the increased costs of removing defects after the merge to `trunk`. This is the fundamental struggle in the implementation of continuous integration in real-life projects.

In some projects, the `trunk` branch needs to be maintained in a near deployable state at all times. For larger more complex projects, therefore, the level of testing before merging to `trunk` needs to be very high which can significantly delay integrations with all of its productivity-robbing impacts. In addition, issues with both the implementation of the infrastructure driving automated testing and issues in the software being tested itself can cause additional delays in integrations. An interesting case study of this is the current continuous integration process and pre-merge automated testing implementation being used for the Trilinos project [11].

Trilinos [11] is a large collection of software that contains advanced numerical algorithms for constructing computational science simulation codes. It is implemented mostly in C++ and uses a lot of deeply nested templating which is known to lead to very long build times. In addition, Trilinos must maintain portability to a number of advanced platforms that drive important internal projects at Sandia National Laboratories (SNL). One important customer is the Advanced Technology Development and Mitigation (ATDM) project at SNL which currently drives much of the Trilinos and close ATDM application (APP) development. The ATDM APPs SPARC [7] and EMPIRE [6] require relatively frequent integrations with the main Trilinos development branch `develop`. Critical platforms include machines with GPUs, various node threading and acceleration architectures which involve builds with every major compiler including Clang, GCC, Intel, IBM, and CUDA. Therefore, the Trilinos `develop` branch must maintain a high degree of stability on a wide range of these challenging platforms that drive the ATDM project.

To maintain the stability of the `develop` branch, the Trilinos team has implemented a custom pre-merge PR testing process using Jenkins to test PRs in GitHub. Currently, there are seven primary build configurations of Trilinos that test with Clang, Intel, CUDA, and three different versions of GCC. Within these seven configurations, several critical options are varied (e.g. debug vs. optimized, runtime checking vs. no runtime checking, shared vs. static libraries, different data-types for template instantiations, etc.) to provide some reasonable coverage and protection for a range of important SNL customers. The implementation of this PR testing system has helped to dramatically improve the stability of the `develop` branch.

While the Trilinos PR testing process has been very successful at improving the stability of the main `develop` branch, it has not come without some cost and impacts to the basic CI workflow. Due to some of the issues described below, there have been periods of time where there have been significant delays in integrating topic branches into the main `develop` branch. The implementation of CI testing in real-world CSE projects is non-trivial and the below discussion is meant to highlight some of the non-trivial challenges that can occur and to explain the causes of CI delays that motivate using more advanced Git workflows to work-around these challenges. First, there have been periods of time where some of the PR builds took many hours to run, depending on what was changed in the topic branch compared to `develop` (e.g. the Intel build took 5 hours and the CUDA build took nearly 10 hours in wall clock time to run at different points in time). Since PR testing is performed in each PR independently and since there are finite computational cycles to run these builds, some PRs have seen

significant delays before the PR testing processes even started to test the topic branch in the PR. Delays starting testing (due to waiting for computational resources to come available currently being used to test other PRs) of several hours are fairly common and can be over 24 hours in some cases. Also, random failures in the Trilinos tests are sometimes injected into the `develop` branch that can fail a PR testing iteration. In addition, the Trilinos PR testing process has experienced some random failures in the underlying testing infrastructure (e.g. git fetch errors due to network connectivity problems, Jenkins communication problems, disks running out of disk space, overloading the RAM on the compile node crashing the build with out-of-memory errors, overloading of the cores on the compute node resulting in test timeouts that would not occur if the CPU was not overloaded, etc.). Random failures in Trilinos itself which sneak onto the `develop` branch along with the random infrastructure failures can result in PR build/test iteration failures that have nothing to do with the PR topic branch being tested. And a single test failure in a single one of the seven PR builds requires rerunning all seven builds again from the beginning which increases the chances of having a failed PR testing iteration. For example, if there is only a 20% chance of a failure in any one of the seven PR builds, then the chance of having at least one of the PR builds having a failure jumps to $1-(1-0.2)^7 = 0.79$ or 80%. In this example, with only a 20% probability for any one PR testing iteration to pass, it can take many PR testing iterations for all seven PR builds to pass and allow the merge of the PR to `develop`. Such extreme cases are not the norm but there are periods of time where this has the case (but there are no systematic statistics being collected to show the frequency of these occurrences).

The combination of delays in starting PR testing, expensive PR builds once started, random failures in the Trilinos tests themselves, random failures in the underlying PR testing infrastructure, and requiring a single PR test iteration with 100% passing tests in all seven PR builds has resulted in periods of time where a given PR topic branch took several days to merge. In some rare extreme cases, the delay in merging a PR topic branch to `develop` has stretched into two weeks.

When it can take up to several days or over a week to merge a PR topic branch to `trunk` before follow-on work can begin based on that, simple fast continuous integration (i.e. where small topic branches are merged frequently) is not realized. In situations like this, one can adapt by using other approaches to maintain productivity. Some examples of more advanced Git workflows to work around long delays to merge to `develop` in the Trilinos PR testing process are described in the next section.

IV. Advanced Git workflows to recover productivity

To work around situations where simple continuous integration breaks down, one can often exploit special properties of the code base, the development team, and other considerations to devise a specialized Git workflow to avoid productivity losses. One example where this has been done is the SNL ATDM project involving specialized build configurations of Trilinos supporting the ATDM program [3].

The ATDM Trilinos build configuration is version controlled along with the rest of Trilinos. This ensures the configuration is consistent with the Trilinos source code and it simplifies the deployment of this configuration system to the ATDM APP customer codes. The ATDM Trilinos build configuration is contained in the Trilinos Git repository and currently contains over 150 files and 6k lines of code. Changes isolated to just these files results in the Trilinos PR tester to only run a small number of unit tests for the ATDM Trilinos configuration code. But sometimes, changes to other files in Trilinos are needed as well including base-level files that trigger the enable and testing of all Trilinos packages and the full Trilinos test suite. (The PRs that trigger the full Trilinos test suite to run are the ones that often result in the longest delays in merging due to needing many PR testing interactions until they all pass.) The ATDM Trilinos configuration currently supports 46 different builds of Trilinos on eight different platforms including several advanced platforms consisting of multiple GPUs per node and several different threaded CPU and accelerator architectures that drive the ATDM program. These different ATDM Trilinos builds are run every 24 hours against a Trilinos branch called `atdm-nightly` and post to the Trilinos CDash site, where they are monitored. The `atdm-nightly` branch is updated from the Trilinos `develop` every day at 9 PM local time. Often times, changes in the underlying systems, third-party libraries, and changes in Trilinos itself require updates to the ATDM Trilinos configuration files. Each update of these ATDM Trilinos configuration files requires creating a PR and subjecting it to PR testing involving the seven PR build configurations before it is merged to the main Trilinos `develop` branch.

In the early days of the development of the ATDM Trilinos configuration and stabilization effort, there were periods of time where significant delays were experienced in the merging of PRs to the `develop` branch due to some of the issues described above. This caused unnecessary delays in deploying changes to the ATDM Trilinos configuration to nightly testing and delayed follow-up work. Therefore, to address these delays, a more sophisticated set of Git workflows was developed. The key elements of that workflow are shown in Figure 2.

The workflow shown in Figure 2 addresses two problems. The first problem addressed is in avoiding delays deploying updates to the ATDM Trilinos configuration to the nightly ATDM Trilinos nightly builds to restore the running of the Trilinos test suite to protect ATDM customers. To accomplish this, one cannot directly merge the topic branches to the `atdm-nightly` branch in the middle of the day as that would result in inconsistent versions being tested for the current testing day. Instead, the topic branch in the unmerged PR (e.g. `topic-b` and `topic-c`) is merged to an intermediate branch `atdm-nightly-manual-updates`. The `atdm-nightly` branch is then updated

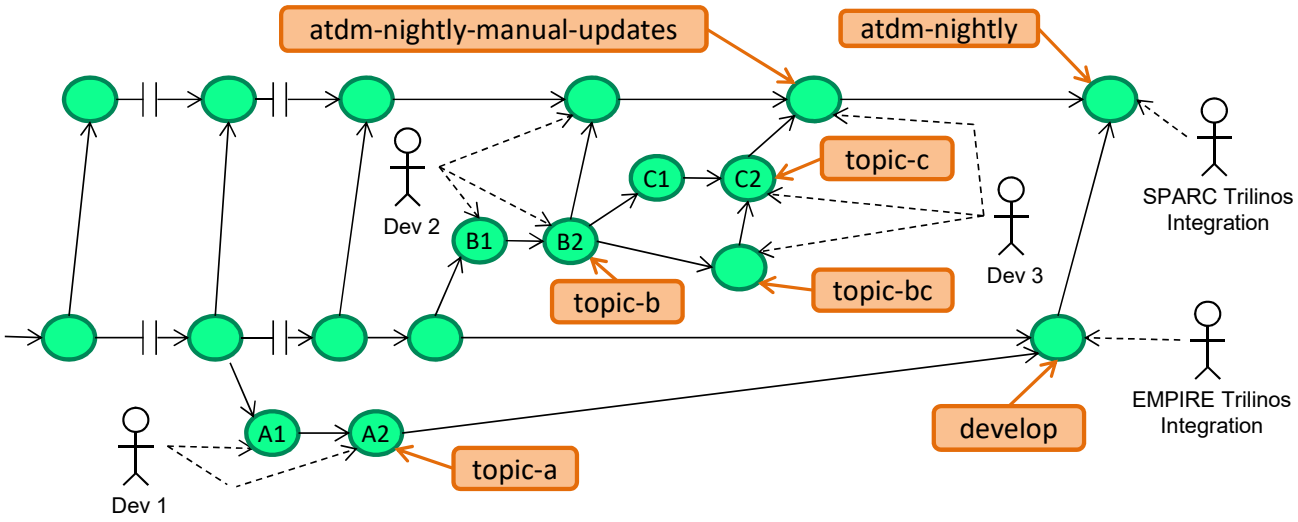

Fig. 2. **ATDM Trilinos Configuration Git Workflow**: This is the workflow used to develop and deploy changes to the ATDM Trilinos configuration to drive testing of Trilinos on the ATDM platforms and integration testing with the SPARC APP code.

each night by first fast-forward merging the `atdm-nightly-manual-updates` branch and then merging in the `develop` branch. That way, the nightly builds for the next testing day include all of the changes in the `develop` branch and all of the manual merges from the prior day through `atdm-nightly-manual-updates`.

The second problem this workflow addresses is avoiding merge conflicts between different change sets or avoiding delays in starting follow-up work due to delays caused by PR testing issues described above. In the scenario shown in Figure 2, the branches `topic-b` and `topic-c` are changes/additions to the ATDM Trilinos configuration that involve the same files. If these changes were to be made on two independent branches from the same ancestor, then their merge would cause massive conflicts. To avoid these conflicts, the changes in the `topic-c` branch must be made following the changes made in the `topic-b` branch. But since the merge of `topic-b` to `develop` is being delayed due to issues with the PR testing process described above, one will need to either wait to make the changes in `topic-c` until `topic-b` is merged to `develop` (which could be several days or longer) or one will have to use a more sophisticated workflow which is shown in Figure 2. In this case, the `topic-c` branch is created off of the un-merged `topic-b` branch to avoid any conflicts. To facilitate the code review of the `topic-c` branch, a new branch called `topic-bc` is created off of `topic-b` and then a PR for `topic-c` is created against `topic-bc`. This allows for a quick code review of the changes unique to `topic-c`. A PR of `topic-bc` is then created against the `develop` branch, which, when merged, will merge the `topic-b` and `topic-c` branches to `develop`.

Also shown in Figure 2 is the `topic-a` branch which was created by a regular Trilinos developer and which takes more than one day for the PR tester to allow the merge to `develop` in this scenario.

The adoption of this more sophisticated Git workflow has allowed the ATDM Trilinos stabilization and integration effort to remain productive and to be less sensitive to delays caused by issues with the Trilinos PR testing system. Without using this more sophisticated workflow over the last 1.5 years, the this effort would be a least a month behind where it is now.

V. Conclusions

In an ideal world, simple continuous integration with trunk-based development (depicted in Figure 1) would yield the greatest productivity. But in the real world, software is expensive to build and test; automated testing processes are non-robust and are subject to many types of random failures; developers don't always chase down random failures as quickly as they should; no one is watching over the entire process looking for troubling behavior and then acting quickly to resolve it; code bases are not always partitioned cleanly separating concerns and avoiding unnecessary synchronizations. It is in situations like this where more knowledge and skill with the advanced usage of the Git distributed version control system can have the largest impact by working around these problems and recovering most of the development teams productivity that would otherwise be lost. Armed with advanced knowledge and skill with Git, a small development team can quickly devise workflows that work around difficult situations by exploiting special properties of the problem at hand.

An example was given where the advanced usage of Git was able to eliminate almost all of the productivity losses and schedule delays that would have occurred with following the simple CI process due to delays caused by issues in the Trilinos PR testing process. Without using this more sophisticated workflow over the last 1.5 years, the ATDM Trilinos stabilization effort would be a least a month behind where it is currently.

However, these increases in productivity through the advanced usage of Git do not come without a cost. In addition to the increase in complexity in the processes themselves that result from the usage of advanced Git workflows, there is also a large up-front cost in training developers to learn Git well enough to understand and perform the workflows correctly. But amazing things can happen when all of the developers in a team reach this level of Git understanding and skill!

REFERENCES

[1] Git distributed version control system. https://git-scm.com.

[2] Turnk based development. https://trunkbaseddevelopment.com.

[3] R. Bartlett and J. R. Frye. Creating stable productive cse software development and integration processes in unstable environments on the path to exascale. pages 1–8, 2019.

[4] K. Beck. *Test Driven Development*. Addison Wesley, 2003.

[5] K. Beck. *Extreme Programming (Second Edition)*. Addison Wesley, 2005.

[6] Matthew Tyler Bettencourt, Eric C Cyr, Richard Michael Jack Kramer, Sean Miller, Roger P. Pawlowski, Edward Geoffrey Phillips, Allen C. Robinson, and John N. Shadid. Empire - em/pic/fluid simulation code. 8 2017.

[7] Paul Crozier, Micah Howard, William J. Rider, Brian Andrew Freno, Steven W. Bova, and Brian Carnes. Advanced technology and mitigation (ATDM) SPARC re-entry code fiscal year 2017 progress and accomplishments for ECP. 9 2017.

[8] P. Duvall and et. al. *Continuous Integration*. Addison Wesley, 2007.

[9] S. McConnell. *Code Complete: Second Edition*. Microsoft Press, 2004.

[10] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development*. Addison Wesley, 2007.

[11] The Trilinos Project Team. *The Trilinos Project Website*.