**UNDERGRADUATE RESEARCH IN NATURAL AND CLINICAL SCIENCE AND TECHNOLOGY (URNCST) JOURNAL**
**Read more URNCST Journal articles and submit your own today at: https://www.urncst.com**

PRIMARY RESEARCH OPEN ACCESS

# Continuous Integration Using Gitlab

Mohammed Shamsul Arefeen, BCS Graduate [1]*, Michael Schiller, BCS Student [2]

[1] School of Computer science, Department of Science, University of Windsor, Windsor, Ontario, Canada N9B3P4
[2] School of Computer Science, Department of Science, University of Windsor, Windsor, Ontario, Canada N9B3P4

*Corresponding Author: shamsul@uwindsor.ca

**Abstract**
The method of Continuous Integration (CI) enables developers to use fast-paced development environments, such as Agile, without the quality of code being compromised. The use of stable repositories on which developers frequently make small changes, running tests with each modification, results in code that is highly tested and readily deployable. GitLab, like other tools of its kind, offers CI features along with code versioning, and several team organizational features that correctly map with Agile artifacts. Before discussing GitLab, this report contains a literature review explaining existing CI practices, and it identifies fundamental aspects that are necessary to achieving the industry standards of CI. This report also evaluates GitLab's effectiveness in maintaining CI standards using an available open-source project (OpenMCT), while also offering insight on the implications of the gathered findings. The paper concludes with possible directions of future inquiry, and also provides guidance on how one might expand on the work initiated by the team.

**Keywords:** continuous integration; agile; software testing; GitLab

## Introduction

Continuous integration (CI) is a development practice which aims to create high quality code with low risk or uncertainty in its behaviour. The process involves a team of developers with each individual merging their code into the currently stable and well-tested code branch, known as the mainline [1]. These merges should be performed daily, or more regularly if possible, by each developer in the team [1]. Due to the frequency of commits, CI is often paired with Agile development methodologies. When an individual has completed their task(s), they will run a set of automated tests to verify their work, and if this passes, it is then merged into the mainline which is again tested. If these tests were to fail, the attempted merge - commonly known as a pull request - would be rejected and the mainline would remain as it previously had been while the developer makes corrections, eventually making another merge request.

This paper will document the application of continuous integration using the tool GitLab and its efficiency at managing said application. It will explain the process used in all stages of the case study and will discuss the results of the study. In addition, it will discuss previous works which have been found to clearly describe the benefits of continuous integration using Git, and which have proven to be generally helpful in the process of its implementation. Lastly, it will seek to examine possible ways of building on that foundation which the study accomplished within itself, and areas of examination which the team wish to have followed if time had been permitting.

The concept first appeared in a publication for Extreme Programing, wherein it was mentioned that a team would need to "integrate and test several times a day", while also getting rid of code that was not used for integration [2]. Although there is no single correct technique for implementing Continuous Integration, the concept of CI was solidified later by Martin Fowler, who states that there are certain industry-relevant practices that are proven to uphold the standards of said methodology [1]. These practices include:

1. "Maintaining a single source repository" - the mainline [1]
2. Several commits made to the Mainline, daily [1]
3. Automating both Building and Testing [1]
4. Testing code locally, prior to testing the integration with mainline [1]

The automation aspect of CI is pertinent to Continuous Deployment (CD) as well - the latter ensures that all CI processes are then readily deployed to production [3]. Automating the build along with tests ensures that the number of steps required to deploy are reduced significantly by creating a stable code that is both functional and virtually fault-free [3].

### *Importance of CI in Agile*

The aforementioned points are presumed to have been carried out in a fast paced development environment, and as such it is very common to see CI and Agile being closely paired to each other [4]; Afterall, the roots of CI are deeply embedded into Extreme Programming [2]. Furthermore, the benefits of CI and Agile are mutual in nature - practice of one should benefit/enable the other. Acceptance tests, for example, play an integral role in Agile environments, and Continuous Integration can support these tests using its own methodology (build automation, frequent deployment, testing locally before merging mainline, etc.) [4].

### *Software Testing*

Testing, in particular unit testing, plays an integral role in CI, and as such it's important to discuss how a team might approach testing their code. There are several layers of testing that can be undertaken, but the bare minimum standard are unit tests [1]. The unit tests, or any tests for that matter, need to be automated, making way for code that is "self-testing" and one which can be used to test both the mainline, and the local repositories [1]. If combining CI with Agile, it is recommended to consider including the acceptance tests into the automated testing process [4]. Acceptance tests will not only build on the pre-existing unit tests, but also help to satisfy the user stories that come with each Agile sprint. A condition that should be met with high priority, regardless of the nature of the tests themselves, is that the testing process is fully automated [1].

### *Version Control*

The ability to integrate changes from several branches into one mainline is made possible through version control tools - the practice of which becomes a necessity in Agile teams. Version control tools such as GitHub allow pull requests to be performed - an action which merges the latest committed branch into the mainline [5]. Pull request performance needs to be maintained throughout the CI cycle, as any interruption or failure to merge could interrupt the workflow of the team. Moreover, pull request performance for a given repository can degrade with an increase in the number of contributors for a given project [6]. Therefore, a reliable version control tool needs to be chosen, one that offers consistent performance with scale and comes with an ability to restrict access to the repository to the development team at hand.

This information can be used to conduct studies, in an effort to show variations in approaches between different environments. The frequency and size of commits may vary depending on the size of the organization - developers from larger organizations tend to commit less frequently than their smaller counterparts [7]. There is a linear relationship between size of commits and the number of committers, variations arise, however, when developers have to choose between merging into mainline directly or merging first from local branches [7].

## Methods

### *Tools Used*

To evaluate Continuous Integration standards, the team has chosen GitLab as the primary tool - a platform which will enable the team to acquire experience using CI and also report on any findings that may be of significance to this study.

**GitLab:** GitLab is a web-based DevOps workflow application that boasts several options for managing Continuous Integration and Continuous Deployment (CI/CD), while also offering their customers with all the benefits of a standard version control tool [8]. GitLab and its services were extensively used throughout the duration of this study for various purposes that will be discussed in later sections. GitLab also provides team management and workflow tools like responsibility, milestones, issues, etc. which are designed to facilitate communication and collaboration within the team throughout the development lifecycle, enabling Agile development [8].

**GitLab CI/CD:** GitLab provides its customers with Continuous Integration tools such as merge requests to the mainline, automating build and tests, validating the changes in the mainline, etc. [9].

**GitLab Runner:** GitLab Runner is the app that handles the build automation and test automation and is responsible for running all integration pipelines; at least one instance of GitLab Runner needs to be running (preferably on a remote server) to completely utilize GitLab's CI/CD tools [10]. Multiple GitLab Runners can be installed on separate servers, and they can be configured to shared, specific or group settings, as per the team's requirements [10].

### *Open*-source *Project*

The subject on which this study was tested is OpenMCT - an open-source mission control framework developed by NASA, available on both mobile and desktop [11]. It was built using Node.Js, Angular.Js and Express.Js for front and back end respectively, and the tests are written with Jasmine.Js, and then run using Karma [11]. There are several reasons as to why this project was ideal for this study. Firstly, it is a large-scale and well-maintained project, meaning that it could be tested and modified in a realistic manner. Secondly, it included many pre-written unit tests (thanks to the past contributors), which reduced the need for unique more tests to be developed for the purpose of this study.

### *Experiment*

**Setup:** With the required tools and test subject being established, the next step for the team was to set up the GitLab environment, which included creating a group, and inviting the rest of the team members for access to the project resources. The team was then able to clone the repository of OpenMCT to a GitLab repository specific to this study, along with its pre-existing branches and commit history, and thereafter begin to experiment. Settings for

CI/CD had to be configured, followed by installation of GitLab Runner on a remote server. An important setting was to automatically merge a branch if pipeline succeeds - a decision which is left to the team members at hand. This does not affect the success/failure of the merge pipeline, as it will still show the results from the tests, and will only pass if all the jobs (comprised of several testing aspects) pass. Also required is a ".yml" file that sits in the code repository - this file carries with it instructions (as scripts) for the GitLab Runner - scripts such as initiating build, tests, caching, etc.

The team was then ready to clone the group project from GitLab to local repositories (respective to each team member) and begin working on their code. The code contribution was minimal, with the team sticking to writing an additional five unit tests distributed between different components (on-screen modules, APIs, etc.) throughout the project, but it was enough to test the functionality of GitLab CI.

**Implementation:** Each individual teammate is ought to test their code using the unit tests available before committing the changes and making a pull request to the remote repository. The pull request clones the local repository on GitLab servers and triggers the GitLab Runner on the remote server to start automating the build and testing. The pipeline for this request will indicate the results from the GitLab Runner, and the success of the test jobs. The team can either "manual merge" the latest branch into mainline or set GitLab to automatically merge if the pipeline succeeds.

Given the erroneous code commits, the goal here was to trigger a "pipeline fail" from pull requests (made from local to remote branch), due to erroneous code that was intentionally inserted into branch(es). This works because the code is self-testing, and if there are errors in the tests themselves, the merge should pipeline should fail. The team member who was assigned to the merge request will be alerted of the failure. Regardless of the results, GitLab provides the option to merge the remote repository into the mainline.

**Results**

Thanks to the CI Analytics tool provided by GitLab, it was possible for the team to acquire information from charts that were automatically generated from Git data [8]. The two most-significant findings include Pipeline Completion Times and Success to Failure rate of Pipelines. A pipeline represents a job that the GitLab Runner has undertaken and may consist of several jobs (test types), the results from which will be used to evaluate the success/failure of the pipeline. According to the guidelines for CI, a team should commit several times a day [1], and this means that numerous pipelines will be processed in a single day. Calculating the time required for each pipeline, and the success of each becomes an important study in understanding the effectiveness of this standard.

***Pipeline Completion Time***

The pipelines themselves correspond to pull requests initiated by teammates and are indicative of either changes in existing remote code branch, or a newly cloned (local to remote) code branch. Regardless, the results in Figure 1 appear to match the outcome that the team had predicted – with pipelines running on erroneous code failing, while those coming from well-tested branches passing the pipeline tests.

The maximum time a pipeline took to finish was 12 minutes, while the fastest pipeline lasted only 5 minutes. The results can be seen below in Figure 2 y-axis represents time in minutes, x-axis is the pipeline code.

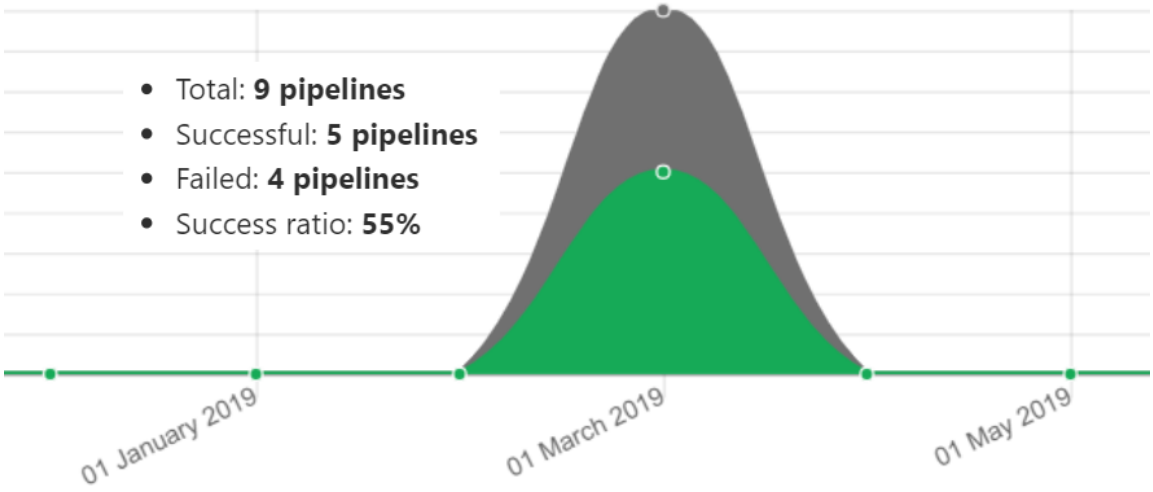


**Figure 1.** Success to failure ratio of pipelines


**Figure 2.** Time required to complete individual pipelines.

The experiment confirms that the time taken to complete a pipeline is relatively short and therefore should not discourage developers from making several commits a day. While the Success to Failure ratios indicate that it is indeed possible to catch faulty commits before they're integrated with the mainline. Therefore, stress should be placed on maintaining high standards for frequency of commits and the quality of tests used for the purpose of CI.

Whilst the results were positive in nature, there were a few noticeable weaknesses of which anyone hoping to implement it should be aware. It was found that erroneous code, if not caught by unit test(s), will be undetected by the CI tool and therefore be allowed to merge with the mainline. This demonstrates the dependence of the success of continuous integration on well-designed tests which will catch as many errors as possible, and how lacking certain tests might result in the mainline being (eventually) faulty due to erroneous code. Also, it is recognized that some developers may avoid running the automated tests before sending the request into the merge queue, to fasten the speed of code delivery. This is a major concern which undermines the very center of the process, and greatly improves the chance of bugs entering into the mainline. Through education and policy this must be discouraged within teams, and the importance of verification and validity must be stressed highly.

## Discussion

### *Evaluating the Results*

The gathered results clearly indicate the usefulness in the functionality of Continuous Integration using GitLab DevOps tools, against the aforementioned industry standards [1]. The successes/failures of the pipelines are what the team expected them to be - the failed pipelines exactly correspond to branches with erroneous code, while the successful pipelines represent those branches from well-tested code. Despite minor setbacks, wherein the integration (merge) pipeline would halt due to unavailability of GitLab Runner (pipeline "98c8e19d" in Figure 2), GitLab maintained a good standard of what to expect from a tool that made CI facilities available for development teams.

The time taken to complete a pipeline is an important metric in understanding how long a pull request would take before the next one can be processed. In a practical environment, several pipelines may be initiated simultaneous and unless multiple GitLab Runners are made available, the pipelines will have to be completed sequentially, in a chronological fashion.

What we derived from this experiment applies not only to a small team but can be extrapolated to understand what can be expected from practical scenarios in a real-world industry environment.

### *Assumptions*

Despite the team's best efforts to adhere to industry practices, the team lacked enough resources to put in quality contributions (code development and testing) to actually simulate a real-world environment. While the team assumes that this is not necessarily a major disadvantage for this study, it still puts the team back at least one factor from a one-to-one mapping of a real-world scenario. Therefore, we assume that any changes in code that are submitted for a pull request are small in magnitude and don't differ from the mainline a great degree, as per best practices for CI [1].

Another assumption that the team had made has to do with the usage of unit tests as opposed to any higher levels of testing. While still meeting the basic requirements for CI, it may not represent a realistic scenario - a company might choose to use acceptance tests or even regression testing. The testing and build processes need to be automated, and the team has maintained that criterion throughout each integration pipeline. However, as the aforementioned process packages came pre-packed with the OpenMCT project repository, the team didn't have the need to customize/re-write the automation scripts. This can change in the industry as per change in requirements, the stack

might change as well, and with it the packages for build automation and test automation.

## Conclusions

Continuous Integration (CI) is a necessary element for fast-paced development environments, and full effort should be made by teams to ensure the fulfillment of the bare minimum standards of CI. Through the literature review, the team identified four standards that are common to most CI environments - the fulfillment of which is necessary to ensure CI [1]. Through the use of GitLab CI/CD tools, the team was able to demonstrate the significance of CI in a simulated Agile environment, while also comparing the functionality of said tools against common standards for CI. Committing several times a day can help to catch errors in small batches, and writing tests for each component ensures that fewer errors are left undiscovered. The process of build and test automation are also a necessity for CI, as they enable the creation of a stable mainline, and facilitate production deployment.

### *Future Directions*

Due to time constraints of the study, it was deemed infeasible to fully develop a piece of software from scratch. Given more time, it would have been a useful and enlightening experience to see through the entire development lifecycle of an application and evaluate CI on that particular example. Also, it would be valuable to perform more realistic and numerous merging scenarios than have been done in this study, and to collect data regarding the relation of the rate successful merge requests and the time spent working on that branch. These are metrics that come into play, especially where there are significantly high number of contributors, and a similarly high number of pull requests being made to the mainline. Another variation is where a team chooses to use different levels of testing as opposed to unit tests – acceptance tests or regression tests are viable alternatives.

Another useful area of investigation would be to implement Continuous Integration using different tools, such as Codeship, Jenkins or CircleCI, and to compare these with GitLab. The team would like to evaluate said alternative tools and discover the degree to which they fulfill the standards of CI. Additionally, the team aims to make a more thorough comparison of DevOps features that are provided by the aforementioned tools. Also, there are multiple approaches to implementing continuous integration [12], with relation to the regularity of merges and various policies, and to experiment with these and compare their advantages and disadvantages would be valuable.

## List of Abbreviations

CI: Continuous Integration
CD: Continuous Development
API: Application Program Interface

## Conflicts of Interest

The author(s) declare that they have no conflicts of interest.

## Ethics Approval and/or Participant Consent

The only participants involved in the study were the authors and no additional consent was required for the study.

## Authors' Contributions

MSA and MPS: contributed to all aspects of the study including the drafting, evaluation of results and reviewing.
MPS: Researched available tools for CI, and features provided by GitLab that map to CI standards. Setup the experiment environment.
MSA: Researched on available open-source projects, conducted the case-study experiment and gathered results (analytics) from experiment.

## Funding

This study was not funded.

## Acknowledgements

Dr. Ziad Kobti: Provided direction and support in formulating the general outline of the study.
Dr. Pooya Morian Zadeh: Provided feedback on necessary improvements to quality of study.

## References

[1] Fowler M. Continuous Integration [Internet]. martinfowler.com. 2019 [cited 3 April 2019]. Available from: https://martinfowler.com/articles/continuousIntegration.html
[2] Beck K. Extreme programming: A humanistic discipline of software development. Fundamental Approaches to Software Engineering. 1998;:1-6. https://doi.org/10.1007/BFb0053579
[3] Meyer M. Continuous Integration and Its Tools. IEEE Software. 2014;31(3):14-16. https://doi.org/10.1109/MS.2014.58
[4] Stolberg S. Enabling Agile Testing through Continuous Integration. 2009 Agile Conference. 2009;. https://doi.org/10.1109/AGILE.2009.16
[5] About pull requests - GitHub Help [Internet]. Help.github.com. 2019 [cited 3 April 2019]. Available from: https://help.github.com/en/articles/about-pull-requests
[6] Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V. Quality and productivity outcomes relating to continuous integration in GitHub. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015. 2015;. https://doi.org/10.1145/2786805.2786850
[7] Ståhl D, Mårtensson T, Bosch J. The continuity of continuous integration: Correlations and consequences. Journal of Systems and Software. 2017;127:150-67. https://doi.org/10.1016/j.jss.2017.02.003

[8] The DevOps Lifecycle with GitLab [Internet]. GitLab. 2019 [cited 3 April 2019]. Available from: https://about.gitlab.com/stages-devops-lifecycle/
[9] GitLab Continuous Integration & Delivery [Internet]. GitLab. 2019 [cited 3 April 2019]. Available from: https://about.gitlab.com/product/continuous-integration/
[10] Configuring GitLab Runners [Internet]. GitLab. 2019 [cited 3 April 2019]. Available from: https://docs.gitlab.com/ee/ci/runners/
[11] nasa/openmct [Internet]. GitHub. 2019 [cited 3 April 2019]. Available from: https://github.com/nasa/openmct
[12] Ståhl D, Bosch J. Modeling continuous integration practice differences in industry software development. Journal of Systems and Software. 2014;87:48-59. https://doi.org/10.1016/j.jss.2013.08.032

**Article Information**
Managing Editor: Jeremy Y. Ng
Peer Reviewers: Ikjot Saini, Kalyani Selvarajah, Mahreen Nasir Butt
Article Dates: Received Aug 07 19; Published Sep 11 19

**Citation**
Please cite this article as follows:
Arefeen MS, Schiller M. Continuous integration using Gitlab. URNCST Journal. 2019 Sep 11: 3(8).
https://urncst.com/index.php/urncst/article/view/152
DOI Link: https://doi.org/10.26685/urncst.152