INSTITUTO SUPERIOR TÉCNICO

PROJECTO MEFT

Keeping Master Green with Machine Learning

Author:
João Lousada

Supervisor:
Dr. Rui Dilão

Research work performed for the Master in Engineering Physics

at

Research Group Name
Department of Physics

December 18, 2019

1 Introduction

1.1 Version Control Systems

In software engineering, version control systems are a mean of keeping track of incremental versions of files and documents. Allowing the user to arbitrarily explore and recall the past changes that lead to that specific version. Usually in these kind of systems, changes are uniquely identified, either by a code number or letter, an associated timestamp and the author. One of the most used online platform for this effect is GitHub.

The central piece of these systems is the Repository. - which is the server where all the current and historical data are stored. The current work version of the repository is called working copy and to the line of development that culminated in this version is called a branch. A branch is organized in a time-wise fashion, where one can imagine a time axis, starting at the beginning of a project, with points scattered along it, representing when changes occurred and were uploaded from the working copy, to the repository - this procedure is called *committing* or *to commit*.

A repository may be composed of many branches, corresponding to different modification paths, caused by different approaches to solve problems. Branches can share common history and diverge at some point generating its own history and later they can be merged back into one another, or substitute one another. By default, when a repository is created, it only has one branch, which is called *master branch*. Later, at a given point, the set of files contained in version control might be split into two copies that can be developed at different speeds or with different approaches, independent of each other, giving rise to two branches. Like it is depicted in the following example [4]:

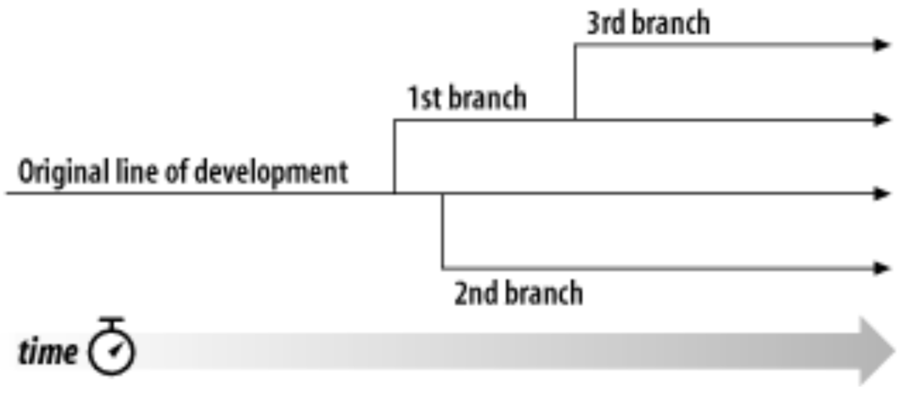

FIGURE 1: Branches of development [4]

Additionally, as time evolves and more commits are made, it is easy to keep track of the progress made so far, but most importantly it is mistake tolerant. For example, a developer starts writing code in a particular way and later on realises that there are mistakes, or her point of view of the problem was not the most appropriate. There are two solutions, scrap the progress and start with a different strategy, or try to reverse all the changes manually to end up at the start, which is boring, error-prone and counterproductive. With version control it is simple to "load" any version and go back to any past point.

From the developers and teams point of view, not having to worry about "messing up" and having the ability to make changes without a great deal of compromise, provides the right safety net to tackle problems from different angles and adapt to changing environments, without wasting resources.

This work intends to explore all the possible configurations of this system and the advantages and disadvantages of each different approach., applied to a real data set of a large company.

1.2 Mono-repositories vs. Multi-repositories

Google's strategy is to store billions of lines of code in one single giant repository, rather than having multiple repositories. This gives, the vast number of developers working at Google, the ability to access a centralized version of the code base (master branch [1]) - *"one single source of truth"*, foment code sharing and recycling, increase in development velocity, intertwine dependencies between projects and platforms, encourage cross-functionality in teams, broad boundaries regarding code ownership, enhance code visibility and many more. However, giant repositories may imply less autonomy and more compliance to the tools used and the dependencies between projects. [5]

1.3 Software Testing

Software Testing is a crucial process of modern Software Development, focused on conducting an empirical investigation, in order to estimate the quality of computer software, given the context it is applied to.[3] The technological age revolutionized society and, more than ever, millions of lines of code are produced at an unrivaled pace, specially in large companies with multiple products, that may have common dependencies. Code bases sizes are tremendously huge, moreover suffer constant change, therefore tests are a measure of software quality, their application results in a PASS/FAIL outcome. In version control, a branch is said to be "green", if it always compiles successfully and passes all the build steps. A "red" master corresponds to the opposite.

Tech companies want to make sure their code repositories are operational, bullet-proof, adaptable and of high quality, and ideally, always having a green master provides the right conditions to maximize productivity and avoid delays.

The problem relies on the fact that tests do not run instantaneously and changes can only be integrated after the tests give green light. The time a developer has to wait before her changes are incorporated into the master branch (or not), is called *turn-around time*, which can greatly affect performance: If a commit "breaks" the master branch (i.e. regression), for a wide number of reasons (code does not compile, failed tests, performance drop, etc), we might end up in a undesirable situation. If all developers use the same platform and it is faulty, future changes will aggravate and have a snowball effect, buildings mistakes on top of each other, resulting in a cascading hindering of productivity and causing delays in releasing new features and updates. Meaning it is crucial to detect, as soon as possible, when, where and who exactly broke the master branch, such that the developer can fix it or the regression can be pulled back, to mitigate "collateral damage".[1]

At the core of the problem lies a trade-off between correctness and speed, several different approaches can be taken to balance the two, which will be explored in section 2.

1.4 Continuous Integration

Continuous Integration is the practice of integrating new or changed code into an existing repository, with regularity, providing feedback on whether the code passed or failed a given test.

Quoting [6]: *"Continuous Integration (CI) testing is a popular software development technique that allows developers to easily check that their code can build successfully and pass tests across various system environments"*.

However, it is not straightforward to develop and configure this methodology, there are numerous approaches and several ways to solve the same problem, which are explored below. The goal here is to navigate through the options, comparing them and acknowledging what features can be leveraged at the expense of others, making it possible to find a scalable solution.

[1] a branch is a pointer to a timeline correspondent to the different versions of the project

2 Background

2.1 System Configuration

2.1.1 Naive-approach

To better understand how to approach testing, let us consider the most simpler of scenarios:

Louis and his 2 college friends decided to open a Start-Up to develop mobile applications. They decide to use GitHub to keep track of the progress made, in which everyone contributes. On a daily average, the number of commits is given by $N_C = 10$ and each commit is subjected to a battery of tests, composed by $N_T = 20$ tests, which gives a total of tests per day $N_T D = N_C \times N_T = 200$, assuming each tests takes 10 seconds to run, we end up with a total time of $T = 2000$ seconds. Later on, the Start-Up became a huge success and Louis had to hire more developers, to work on the multiple projects the company is involved in. Now, $N_C = 1000$ and $N_T = 10^4$, giving a total $N_T D = 10^7$ tests/day and a total time $T = 10^8$ seconds, which results in some astounding 1158 days, with the estimate of only 10 seconds tests, which in reality is certainly not true, according to [8], large tests take at most 15 minutes.

This strategy is called naive-approach, because it runs all of the tests for each commit. As we saw, as teams get bigger, both the number of commits and the numbers of tests grow linearly, so this solution does not scale. Also, as [2] stated, computer resources grow quadratically with two multiplicative linear factors.

For example, Google's source code is of the order of 2 billion lines of code and, on average, 40000 code changes are committed daily to Google's repository, adding to 15 million lines of code, affecting 250000 files every week, having the need to come up with new and innovative strategies to handle the problematic.[8]

2.1.2 Cumulative Approach

One possible solution to contour the large amount of time spent on testing and waiting for results is to accumulate all the commits made during working hours and test the last version, periodically, lets say during the night. This enables teams to develop at a constant speed without having the concern of breaking the master branch, but rather worrying about it in the future. In this case, the lag between making a commit and knowing if it passed or fail is very large, running the risk of stacking multiple mistakes on top of each other.

Using this approach, two possible outcomes can occur: either the build passes every test and the change is integrated, or it fails, causing a regression. Here, we encounter another crossroads, as the batch was tested as a cumulative result, we have no information whatsoever of what commit, exactly, caused the regression.

This problem can be solved by applying search algorithms:

- **Sequential Search** - splitting the batch into smaller parts and build all of them individually with time complexity is O(n).
- **Binary Search** - divide the batch in half and build the obtained subset, if pass, discard it, if fail, repeat until narrowing it down to a specific change. Time complexity O(log n)

After finding out which commit caused the regression, it can be fixed or rolled back to the version before the change was introduced, meaning that possibly the progress made during that working day has to be scrapped and re-done, postponing deadlines and, at a human level, lowering down motivational levels, with developers being "afraid" of making changes that will propagate and will only be detected later on, so allowing a large turn-around time may results in critical hampering of productivity.

In comparison with naive-approach, cumulative approach is more scalable in the way that it avoids testing every single commit made, only trying to test the last of the day, spending more time trying to trace back the origin of the failure. So in conclusion, we forfeit absolute correctness to gain speed and pragmatism, although this solution's computer resources grow linearly with the number and time of tests.

2.1.3 Test-Selection Approach

As seen above, a major problem still prevailing is the fact that tests occupy a relevant chunk of the time used in Continuous Integration. Associated with each repository, there is a battery of tests that, so far, is being applied to a certain commit, but what if we are in the case of applying the same tests twice or more ? What if a test is not adequate to the type of change a developer did ? If a test always passes, is it a good test ? Should one apply tests in an orderly way, prioritizing small tests or the ones that are more likely to fail ?

A good start would be to eliminate redundant tests, thus saving time and reducing the computational effort to execute them, this technique is called test suite reduction. [7] It consists on cherry-picking a subset from the battery of tests, or test suite, that, without loss of quality, guarantees coverage of the changes made. The importance of this task is shown by looking at Google's Continuous Testing data: in a history of 5.5 Million tests, only 63K of them ever failed, all the remaining has never failed; The pass:fail ration is about 99:1, so testing some of them less frequently may reduce turn-around time; Some files are more likely to fail than others, the ones with a lot of dependencies or that are modified by several developers. [2]

So it is straightforward to see that by examining such patterns, one can have a practical and useful line of investigation to significantly reduce turn-around time.

NOTE: Here i can develop the theory of test selection more mathematically. If there is spare space at the end.

2.1.4 Change-List Approach

In this case, changes are compiled and built in a serialized manner, each change is atomic and denoted as a Change List (CL). To clarify this concept, lets take a look at Google's example of workflow in Figure .

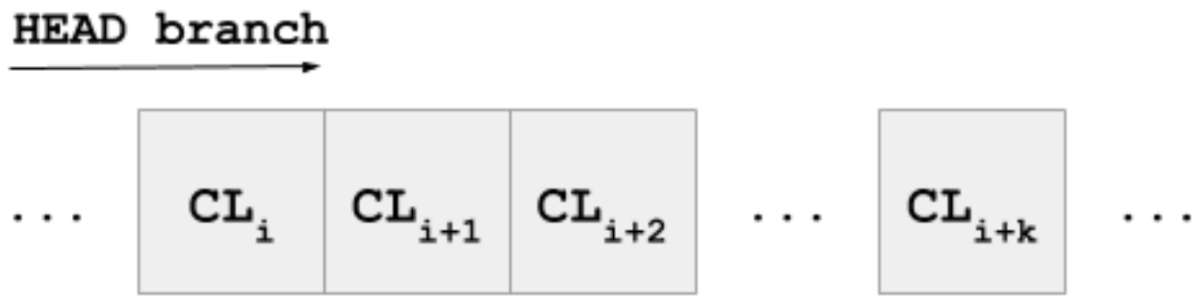

FIGURE 2: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [8]

Individually, tests validate each CL and if the outcome is pass, the change is incorporated into the master branch, resulting into a new version of the code.

In the case of a failed test, a regression is caused. For example, imagine that CL_i introduces a regression, causing failure on the consecutive system versions. Possible fixes and tips:

- investigate the modifications in CL_i, to obtain clues about the root cause of the regression.
- Debug the fail tests at CL_i, which is preferable to debug at later version CL_{i+k}, whereas after CL_i code volume might increase or mutate, misleading conclusions.
- assign the author of the change to fix the fault, rather than other developer.
- revert or roll back the repository, until the point where it was "green" in this case CL_{i-1}. [8]

The novelty is introduced in test validation:

- **Presubmit tests** - Prior to CL submission, preliminary small/medium tests are run, if all pass the submission proceeds, else it gets rejected. So instead of having to wait until all tests are complete, the developer has a quick estimate of the status of the project.
- **Postsubmit tests** - After CL submission, large/enormous tests are executed and if they indeed cause a regression, we are again in the situation where we have to find exactly what CL broke the build and, possibly, proceed in the same manner as Cumulative Approach to detect what version caused the regression. [8]

The major disadvantage, once more, emerges when a postsubmit test fails: a 45 minute test, can potentially take 7.5 hours to pinpoint which CL is faulty, given a 1000 CL search window. [8]. So automation techniques, quickly putting them to work and returning to a green branch is pivotal.

2.1.5 Always Green Master

3 Objectives

-
-
-

4 Sample-Data

References

[1] Sundaram Ananthanarayanan et al. “Keeping Master Green at Scale”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys ’19. Dresden, Germany: ACM, 2019, 29:1–29:15. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303970. URL: http://doi.acm.org/10.1145/3302424.3303970.

[2] Atif Memon et al. “Taming Google-scale Continuous Testing”. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 233–242. ISBN: 978-1-5386-2717-4. DOI: 10.1109/ICSE-SEIP.2017.16. URL: https://doi.org/10.1109/ICSE-SEIP.2017.16.

[3] B. Meyer. “Seven Principles of Software Testing”. In: *Computer* 41.8 (2008), pp. 99–101. ISSN: 1558-0814. DOI: 10.1109/MC.2008.306.

[4] Michael Pilato. *Version Control With Subversion*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2004. ISBN: 0596004486.

[5] Rachel Potvin and Josh Levenberg. "Why Google Stores Billions of Lines of Code in a Single Repository". In: *Commun. ACM* 59.7 (June 2016), pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/2854146. URL: http://doi.acm.org/10.1145/2854146.

[6] Mark Santolucito et al. *Statically Verifying Continuous Integration Configurations*. 2018. arXiv: 1805.04473 [cs.SE].

[7] Shin Yoo, Mark Harman, and Shmuel Ur. "Measuring and Improving Latency to Avoid Test Suite Wear Out". In: *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009* (Jan. 2009). DOI: 10.1109/ICSTW.2009.10.

[8] Celal Ziftci and Jim Reardon. "Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 113–122. ISBN: 978-1-5386-2717-4. DOI: 10.1109/ICSE-SEIP.2017.13. URL: https://doi.org/10.1109/ICSE-SEIP.2017.13.