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 10, 2019

1 Introduction

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.

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.[6]

Companies want to make sure their code repositories are operational, bullet-proof, adaptable and of high quality. Google's strategy is to store these 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.[4]

However, the major issue is being able to maintain performance: If a code change "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].

To do this efficiently,i.e. with minimum costs and limited resources, recent studies advocate **Continuous Integration** as the preferred candidate to handle the problematic. [1, 6, 2, 5]

1.1 Continuous Integration

Quoting [5]: *"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"*. When developers upload a new version of their code, they are **committing** the changes to a branch. CI is a system that, when a commit is made, the code is automatically compiled and tested, in order to, if that is the case, rapidly catch a regression. To clarify this concept, lets take a look at Google's example of workflow in Figure , where changes are compiled and built in a serialized manner, each change is atomic and denoted as a Change List (CL).

Individually, tests are applied to a CL and if it passes, 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.

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

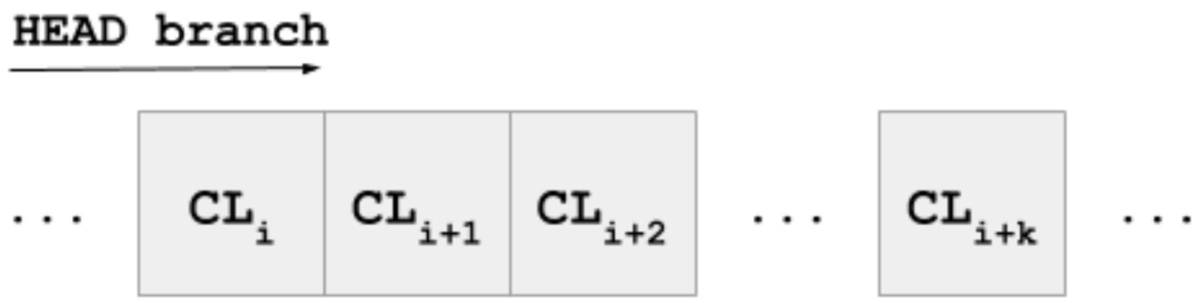

FIGURE 1: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [6]

- revert or roll back the repository, until the point where it was "green" [2] in this case CL_{i-1}. [6]

1.2 Continuous Testing

Applying tests is a measure of software quality, it tells us if a given change caused a regression and suites as a performance indicator. However tests do not run instantaneously. The time a developer has to wait, before her changes are incorporated into the master branch (or not), is called *turn-around time*. So there is a trade-off between correctness and speed.[1]

To better understand how to approach testing, let us consider the following hypothetical example:

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 [6], 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.

[2]a green branch only contains files that passed every test so far

1.3 Code Coverage

1.4 Test Selection

2 Objectives

3 Background

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] 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.

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

[6] 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.