Group 16: CSC 510 Software Engineering Git Simplified Aligning with Linux Kernel Best Practices

Swetha Gavini
lgavini@ncsu.edu

Ramya Sai Mullapudi
rmullap@ncsu.edu

Arjun Madhusudan
amadhu2@ncsu.edu

Rohan Prabhune
rjprabhu@ncsu.edu

Krishna Saurabh
kvankad@ncsu.edu

ABSTRACT

Git Simplified is a tool designed to help make git commands easier and more intuitive to use for beginners and advanced users alike. This first section should help you familiarize yourself with this project, so you can spend more time improving/adding features instead of trying to learn how our code works.[2] These takeaways from the most revolutionised practices in the field of software development broadens our knowledge in development lifecycle, and also team work.

This document explains how our project and development aligns with the Linux kernel best practices and led to a successful phase of the project.

KEYWORDS

Linux Kernel, best practices, git simplified, software engineering, development cycle

1 INTRODUCTION

Dealing with one cycle of a software engineering project has made us realize that the project is all about resolving conflicts in commits, build failures, overwritten code, redundant tasks, decision-making, figuring out important issues opened, aligning the work with different phases within itself. All along the way the secret to resolving all this is the team's capacity for introspection and change.[3]

As the evolution is following, we aligned our principles of developing with the LINUX kernel best practices. We could not resist to see that a resilience created by one evolution of kernel developers is a fetched norm to work together and improve the process.

The key practices that our project aligns with Linux kernel best practices are:

1.1 Short release cycles

Our project has done short release cycles. With identified that doing short releases allowed to find the conflicts in code easily while integrating with the last stable build. The new code has fewer changes and fundamental feature introductions which helped us avoid major disruptions and debugging. When we tried out a longer release after taking up the project on one branch, it did not align with other team member who was doing a shorter release. A disruption in the principles also bring the affect back. After we created a principle that each feature added would be immediately, we could see less pressure among teams and no major disruptions. We also found that doing a short release cycle made it possible for someone who missed adding in a feature could quickly do it in the next cycle. Hence, short release cycles are important. Shorter new codes are easier to review and make any changes immediately to make a good stable release. We have done smaller and frequent PRs, commits, releases to maintain a stable build all along.

1.2 Distributed Development model

A distributed system has always been known to parallelize work and give a freedom of thought for a developer for a work assigned. Each team member took different tasks to contribute towards phase 1 of our project. Each member's work was independent with the area they are comfortable in. From the project dashboard, we could take a head start on any issue, feature, or an enhancement we want aligning with our interest and familiarity.

We brainstormed together on what features and enhancements to build on and created a bunch of tasks in issues section. Spreading out responsibility for code review and integration gives the project resources to take over other's tasks when needed.

With this interactive and distributed model figured to be the best way to maintain pace in the development cycle. With code reviews and integration across all the tasks by other team members, everyone knew what was being developed, modified, and enhanced in the project. This seamless code review paired with short release cycles would take less time to review, discuss, integrate, and maintain stability without sacrificing review or quality.

1.3 Consensus-Oriented Model

A consensus-oriented model means, [4]"a proposed change will not be merged if a respected developer opposed it". Our team confides in this rule. Most of our code reviews have all the team members as an assignee. We have a to-and-fro conversation about a developer's approach to ensure that we are on the same page.

This might look time taking, but we were positive that this approach holds all the members united, and we all could discuss some disruptions that might raise with or without this code change. This led us to detect some minor issues in setup, version control, and usability of the features we decided for our project.

1.4 The No-regression rule

A No-regression rule means that if a code base works on a specific setting, all the subsequent code merges must work there too. Doing short releases and maintaining a consensus-oriented model is one step closer to a no regression rule.[1]

A quick review for each release and each commit we do, let's us discuss our code base setting, and versions which we want our code to be compatible with. We run an automated pytest and code coverage to check for software regressions and bugs each time a member makes a commit. If there is an issue that is affected by regression, we immediately address the issue and get the application up. This rule assures our team that releases and phases in the project cycle will not break any existing features. A confidence on a strong "no regression" rule gives the team to work on new capabilities.

1.5 Zero Internal Boundaries

Maintaining a distributed developmental model where each team member is responsible for specific parts of the code yet reviewing one another is a strength for being on the same page. This not only helped us be aware of the changes we are making to the code from time to time, but also an easy shift from one part of the project to another.

When a regression or new issue shows up in the project, we did not have to wait for the code owner to fix it. Anyone available to do it as soon as possible could change it. This confidence comes from the consensus agreement we have with other team members, where we go over the change to fix the issue among all the members to justify the change before confirming to merge to the code base.

By seeing that different issues raised by one team member has been justifiable fixed by others prove that our project has zero internal boundaries.

REFERENCES

[1] Greg Kroah-Hartman. 2016. 9 lessons from 25 years of Linux kernel development. (2016).

[2] Grpup 16 members. 2021. Our project code. https://github.com/arjunptm/GITS.

[3] Timothy Menzies. 2021. 5 Linux kernel best practices. https://github.com/txt/se21/tree/master/docs.

[4] Packtpub.2018.*LinuxKernelDevelopmentBestPractices*.(2018).