EmTa - Employee Meal Tracking Assistant : Software Engineering Project 1

Aditya Ravikant Jadhav
North Carolina State University
Raleigh, USA
ajadhav3@ncsu.edu

Abhishek Arvindkumar Upadhyay
North Carolina State University
Raleigh, USA
aaupadhy@ncsu.edu

Ashritha Bommagani
North Carolina State University
Raleigh, USA
abommag@ncsu.edu

Akash Mukesh Sanghani
North Carolina State University
Raleigh, USA
amsangha@ncsu.edu

Harsh Bathija
North Carolina State University
Raleigh, USA
hbathij@ncsu.edu

ABSTRACT

Linux Kernel Best practices are widely used to ensure better software development cycles. This paper explains the five the Linux Kernel Best Practices for Software Development, their implementation in the development process and their benefits.

KEYWORDS

software development, linux kernel best practices

ACM Reference Format:
Aditya Ravikant Jadhav, Abhishek Arvindkumar Upadhyay, Ashritha Bommagani, Akash Mukesh Sanghani, and Harsh Bathija. 2021. EmTa - Employee Meal Tracking Assistant : Software Engineering Project 1. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Linux Kernel is a vast open source project which has been in use for decades. Part of what makes it such an effective tool is the practices which have made it a prime example of collaborative development. The Linux kernel is a very huge project and they have had their own challenges. A part the reason behind the success of Linux kernel is their development process. Linux Kernel defines a set of rules for the people working on it, which are considered to be very efficient. These five practices are as follows:

(1) Zero Internal Boundaries
(2) No Regressions Rule
(3) Consensus Oriented Model
(4) Distributed Development Model
(5) Short Release Cycles

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 LINUX KERNEL DEVELOPMENT BEST PRACTICES

2.1 Zero Internal Boundaries

The idea of zero internal boundaries imply that there should no restrictions amongst the contributors as to who can access what specifically. Anyone working on the project should have equal access to all the tools and libraries used for the implementation of the project. This principle corresponds to the "evidence that the whole team is using the same tools" and "evidence that the members of the team are working across multiple places in the code base" criteria in the project 1 rubric[1].

In order to ensure that we followed this rule in our project, we kept our requirements.txt completely up to date to avoid any miscommunication within the team members regarding the same. Each of our team member had same administrative permissions across all the files of the projects. There were no tools or libraries which were only available to some of the team members. We had divided the workload by assigning specific tasks to be completed individually, however, every team member communicated via virtual meetings, emails, etc to execute and test each part of the project.

2.2 No Regression Rule

The No Regression Rule states that when the new upgrades are released, existing code or functionality should not be lost. There should be proper documentation regarding the previous and current versions of the code because if the code quality reduces the developers can roll back to the previous versions for investigation. In this way even if the no regression rule is broken we can acknowledge the problem and take necessary steps to rectify it.

To make sure we followed this rule, all features, upgrades, and bugs/bug fixes were well documented. We constantly monitored to see if the new upgrades lead to the quality reduction. We used branches for features to be implemented and made sure that when they were merged into the master branch, it did not affect the functionality of other features.

2.3 Consensus Oriented Model

The consensus-oriented model of development implies that there should be no changes made to the code base, if any (respected)developer is opposed to it. In other words, no single group can make changes to the code base at the expense of the other groups. In this way, the integrity of the code base is maintained.

We made sure that this rubric was maintained in our project by assigning at least 2 reviewers to check the code before getting merged to the master branch. We kept the issue short so that it was easily comprehensible and concise to ensure easy merging.

2.4 Distributed Development Model

The distributed development model break the project into various components and assigns it to different developers. In this way, no one is responsible for the review and development of the whole project. The rubric includes this best practice by making sure that the "workload is spread over the whole team" [1] and keeping track of each team member's contribution. Though developers are assigned to the components they are most comfortable with, it is important that all the developers have good understanding of the other of the project. An important part of the distributed development model is communication which implies that the team members should be connected via voice or chat channel in order to avoid conflicts that may arise due to miscommunication.

We primarily used discord server to communicate and had two in-person meetings a week. The workload was distributed evenly where we assigned ourselves individually and in pairs for frontend development, backend development, database management and documentation. All the changes and design was decided unanimously and we ensured that they get up branches were short-lived add merged after thorough discussion with the team members.

2.5 Short Release Cycles

Short Release Cycles are important as they guide us towards our goal clearly and the overall progress can be reviewed more comprehensively. This provides continuous evaluation of what is important and what is not which would have not been possible at the start of the project. Short release cycles are advantageous because they make it easy to estimate how long something will take by defining a unit of work and a time range rather than trying to prepare large features over a long period of time.

This is a difficult practice to follow for a project of this scope and time period, as we have only been working on it for one month. Therefore, our short release cycles were done as frequent merged pull requests rather than full releases.

3 CONCLUSION AND FUTURE WORK

We followed the Linux-Kernel standard development practices which were both efficient and easy to implement without any major learning curve. We communicated thoroughly with all the team members and maintained the Github repository in such a way that each commit was intuitive and self-explanatory to other fellow developers. The project roadmap was clearly discussed and the workload was distributed evenly amongst all the team members. We took each decisions by incorporating opinions and consent of all the team members.

Shorter release cycles could have been implemented more efficiently if we would have more than one month of time period. For a project of this scope and time limit, however, we feel that this did not limit our efficiency in working on the same.