Slackpoint: Linux Kernel Best Practices

Saksham Pandey
spandey5@gmail.com

Nihar Rao
nsrao@ncsu.edu

Palash Jhamb
pjhamb@ncsu.edu

Shruti Verma
sverma5@ncsu.edu

Manish Shinde
msshinde@ncsu.edu

ABSTRACT

Software Engineering is continuously evolving and has undergone a significant change in terms of timeless principles and aging practices.[3] Development of new software is an arduous task and these principles are an essential aspect of ensuring the end product is of the highest quality. One such highly acclaimed rubric that is widely followed are the Linux Kernel Best Practices which have also been followed in our project - Slackpoint.[2] This paper aims to provide a walkthrough of each Linux Kernel best practice and it's associated implementation in various phases of our project.

KEYWORDS

Open Source, Kernel, Linux, Release, Software Engineering

1 INTRODUCTION

Software is a program or a set of programs containing instructions that provide desired functionality. Software Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems. It is also working within well-defined and generally accepted human endeavor and Slackpoint is our attempt with respect to that. Slackpoint is a slack Bot that helps users perform tasks and then gamify them to make solving of these tasks fun and rewarding. It is written in Python, Slack API and is hosted on GitHub. The project has been developed as a solution which follows the Linux Kernel Best Practices discussed below.

2 SHORT RELEASE CYCLES

Short release cycles refers to the integration of newly developed functionality into an already existing stable release in shorter intervals of time, rather than integrating long pieces of code all at once after a very long period of time. This concept has numerous advantages such as making new functionality/improvements available to users faster, maintaining a stable system without any major deviations/disruptions, increases team flexibility etc. We have adapted this practise of short release cycles in our project by committing and integrating our respective functionalities/improvements as and when they were reviewed as stable. Our team generally followed the practice of committing whenever functionality of a new feature was completed or whenever a team member requested it. Waiting for a complete feature limited "broken" code that was in existence and kept the number of commits up since our defined features were relatively small and accomplished in a matter of days.

3 DISTRIBUTED DEVELOPMENT MODEL

Distributed development is a software development model where different portions of the project are assigned to different individuals, based on their familiarity with the area. This helps in seamless integration and code review of the project.

This practice requires that responsibility across a project to be distributed. A single talented developer can be easily outmatched as a code base grows [1]. The project 1 rubric ensures this by including points about how all users are contributing to the repository with some evidence existing in commits. The rubric also requires the use of issues which allows work to be easily defined and divided among group members. GitHub provides functionality to view contributions by each user, so we met the rubric requirements by ensuring members were contributing and there contributions are noted within the repository. We also utilized GitHub's issue tracking and progress boards distribute development.

4 CONSENSUS-ORIENTED MODEL

The Linux Kernel Community strictly adheres to the Consensus-Oriented Model which states that no change shall be implemented if even a single developer opposes it.[1] In projects where a large number of people are involved and directly hold a stake as a developer, exchange of different ideas is evident. It is important to have every stakeholder on board before implementing one of such ideas. Consensus oriented model suggests that any decision that is taken amongst the developers about the project must be a collective one. This ensures integrity among the developers and there will be a streamlined thought about the goal amongst the entire team throughout the project.

This thought-process was seriously followed during the implementation of our project. Every developer in our team contributed to the brainstorming process of the features even though they were not writing the code for it.

5 NO REGRESSION RULE

The main focus of the 'No Regression Rule' is that, with every new code or feature addition, the existing functionality of the application should not get affected. Since feature addition and code refactoring is a constant process that happens in any development environment, utmost care has to be taken to not eliminate any required functionality. Also, it is expected from the developers to address these issues quickly and bring the application back and running. We consider this as the most important principle in our development process as every individual who works on independent features should not hamper the other existing features in the application. This was ensured by having strong test cases and a code coverage mechanism which checks for minimum threshold values to be passed for each pull request and code commit that was made. As per our roadmap, there is scope for many updates. These updates will be done in accordance with the No Regression Rule, such that no update will change the basic functionality and interface of the tool. In our project, through running our tests and build on every push of code, we have maintained forward and backward compatibility on the whole application.

6 ZERO INTERNAL BOUNDARIES

The Zero Internal Boundaries principle says that there should be no boundaries between contributors to a project as to who can access different parts of the project. Anyone working on a project should be able to have access to the tools and code used in different parts 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]. We implement the Zero Internal Boundaries principle in our project as it can be seen that each of the team members contribute throughout various source files in the project, and we only use cross-platform open-source tools to ensure that team members can all work across the whole project i.e github. The project was developed in python3 and everyone had knowledge of the same and the resources used by everyone was same.

7 CONCLUSION

The Linux kernel has been around for over three decades in the open source sphere and continues to be extensively developed and used in several industries around the world.[1] With thousands of developers contributing to and updating the kernel, the importance of adhering to software development best practices is evident from it's unmatched success. We have endeavored to emulate these best practices during the development of Slackpoint [2]. The approaches chosen were inline with the goals of the team and therefore helped us reach the targeted milestone effectively.

REFERENCES

[1] https://medium.com/@Packt_Pub/linux-kernel-development-best-practices-11c1474704d6
[2] https://github.com/nihar4276/slackpoint-group18
[3] https://medium.com/designing-distributed-systems/paxos-a-distributed-consensus-algorithm-41946d5d7d9