WolfCare : Linux Kernel Best Practices

Divya Giridhar dgiridh@ncsu.edu
Shreyas Titus stitus@ncsu.edu
Dhruv Mukesh Patel dpatel49@ncsu.edu
Ravi Ghevariya rghevar@ncsu.edu
Manan Patel mrpatel8@ncsu.edu

***Abstract*—This report highlights the use of Linux Kernel best practices followed by Group 25 of the Software Engineering Course (CSC 510). Linux celebrated its 25th anniversary in 2016 and there have been many developmental projects over time. Our project WolfCare, is an Online Doctor Appointment and Symptoms Checker System and this paper will explain the applications of the linux kernel best practices in our project.**

I. Short Release Cycles

The first lesson that stands out is the importance of short release cycles. Short cycles overcome the disadvantages of long release cycles by making new code available quickly. They also add smaller amounts of code compared to long releases which lead to minimal disruption in code integration. This process is great for users as the software is continuously being updated and there is little incentive to try to merge code prematurely.

Our group follows this lesson where we open up issues and discussions for various topics and commit after a reviewing the changes to be stable. The first release, v1.0.0 gives the basic structure of the codebase and was created at the initial stage. The new release v1.0.1 is the final release where all functionalities of the projects are maintained. The usage of branching also highlights the evidence of short release cycles where smaller chunks of code are integrated into the codebase.

II. Distributed Development Model

In the distributed development model, the responsibility of code review is spread out among many maintainers of the project. This gives the project better quality as the work is distributed rather than being handled by a single developer. There is a broader sense of responsibility and better resource management.

In WolfCare, we follow this lesson in the form of issues and discussions. The commits are also an indication of distributing tasks where there is better code quality and test case. We developed a practice of creating a pull request and requesting a review from the other collaborators who is have better experience in a section of code. Through continuous feedback loop, we have ensured that the developer follows standard software engineering practices.

III. Zero Internal Boundaries

For the most part, at any given point of time only some parts of the code base are modified by developers. We do not need to have multiple solutions to the same problem and a single solution for a particular issue is sufficient. Most of the developers are aware of the different software and technologies being used for the project, which ensures that every person in the team has a broader perspective of the entire product instead of a myopic vision tailored to only specific parts of the project. This leads to easier discussions with other developers when information is being shared. Since the overall process is known by everyone in the team, it makes sure that the team has directly or indirectly worked on every aspect of the project.

IV. Tools Matter

Every single project has the underlying tools which needs to be used by everyone to provide results of a good standard. Due to having these various tools, we can monitor the work which is currently being done for the sake of the project. Due to this documentation and a set of standard tools which is required for each project, if any new developers join the group, they can easily be brought up to speed. For the sake of our project we are using github as our version control tool and have a document titled ”requirements.txt”, which tells readers what are the different tools which they require to contribute or modify the project. We have also used codecov for checking the code coverage of our project and this currently stands at 82% at the time of writing.

V. Consensus Oriented Model

A rule which forms the pinnacle of the Linux Community is that a code cannot be merged to the main repository unless it is approved by an experienced developer. When various groups are working on a project, one of the groups cannot make changes without the approval of another team, since that could potentially cause havoc to what the other team is working on. To ensure that the entire team working on a particular project are given an equal share when they are discussing things, we have two files in our repository which aids in this process. We have the Contributing.MD file as well as the Code_Of_Conduct.MD file, to make sure that certain guidelines are maintained when anyone is trying to contribute to the project. The team can feel free to edit these files if they feel that something needs to be changed and it would be discussed and approved if it is considered valid. We have chat channels as well to make sure that everyone’s opinion is heard and an overall consensus is reached before moving forward with the project.

VI. No Regression Rule

As a developer, we know that the code can always be better optimised and improved, but not at the cost of the quality of the product. This is the reason we have the rule pertaining to

no regressions. If a certain regression proves to be unstable, the developers have the option to get back to the previous state instead of trying to fix the issue with regard to the new regression which could cause a significant amount of down time with respect to the product. If a given kernel works in a specific setting, all subsequent kernels must work there, too. Thus this rule gives assurance to users that an upgrade would not break their system.