Connection Between Rubric and Linux Kernel Best Practices

Sumedh Salvi[1], Chaitanya Patel[2], Evan Brown[3], Kunwar Vidhan[4], Sunil Dattatraya Upare[5],
ssalvi@ncsu.edu[1], *cpatel3@ncsu.ed*[2], *webrown2@ncsu.edu*[3], *kvidhan@ncsu.edu*[4], and *supare@ncsu.edu*[5]

[1,2,3,4,5]*Computer Science Department*
[1,2,3,4,5]*North Carolina State University, North Carolina, USA*

Abstract—In this report we will be comparing the parameters provided by the rubric with the Linux Kernel Best Practices. We will group the parameters and explain why a particular parameter is a part of or comparable to one of the five given Linux Kernel best practices.

1 Short Release Cycles

The part of the rubric that is in the group covered by Short Release Cycles is: *project members are committing often enough so that everyone can get your work*. In shorter release cycles, small pieces of code are deployed in shorter time duration. With this, the new code is immediately integrated into stable releases. The developers are not burdened with the idea of integrating huge features base with the risk of the projects losing their stability. The risk of a disruption is also greatly reduced which in turn improves efficiency. In our project, we divided the tasks into smaller parts and constantly kept committing files so that everyone is on par with each other's work.

2 Distributed Development Model

Distributed development model ensures that the process of code review becomes smooth and without any compromise to kernel stability. In a centralized system, where one person reviews and approves any code changes, this person becomes the bottleneck for the faster deployment and implementation of the given code base. In a distributed system, each task is handled by an individual who is an expert in that particular field. This also ensures that bugs do not get past through the review process and thus the kernel stability is ensured. The elements that are a part of the distributed development model are: *a) Decision made by unanimous vote:* Instead of single person holding authority over a code base and making all the decisions, all the parties related to the project are involved and decisions are taken based on a general consensus. *b) Group meeting has a round robin speaking order:* This ensures that everyone's opinions are accounted for rather than one person making all the decisions. *c) Group meetings had a moderator that managed round robin* A group moderator ensures that everyone gets the chance to speak and the process is organized in an unbiased manner. *d) group meeting moderator rotated among the group. e)code conforms to some packaging standard* By keeping a standard format of writing code, integration and understanding of the code improves. This helps in making the overall process easier. *e) code has can be downloaded from some standard package manager f) workload is spread over the whole team (one team member is often X times more productive than the others* In distributed development model, everyone gets an equal opportunity to contribute to the code base. This also ensures that updates take place across the entire system rather than a single section. All the points mentioned further are different ways and systems that could be used to keep a track of all the work that is taking place on the distributed system. This makes sure that the process of code development in a distributed system remains seamless, fast and accounted for. *g) Number of commits h) Number of commits: by different people i)Issues reports: there are many j) issues are being closed k) License: exists l) DOI badge: exists m) Docs: doco generated , format not ugly n) Docs: what: point descriptions of each class/function (in isolation) o) Docs: how: for common use cases X,Y,Z mini-tutorials showing worked examples on how to do X,Y,Z p) Docs: why: docs tell a story, motivate the whole thing, deliver a punchline that makes you want to rush out and use the thing q) Docs: 3 minute video, posted to YouTube. That convinces people why they want to work on your code. r) code conforms to some known patterns*

Comparing it with our project work, till 30th Sepetember, 2021 we have made 182 commits, with individual commits ranging from 25 to 35 per person, depending on the number of files they have edited or worked on slightly. We have a number of active issues along with many closed ones and we have done pretty much all of the tasks which we decided to do in this cycle. We have used the MIT license and have a number of different badges like DOI badge, code coverage badge, python

version badge, python application: passing badge. We have generated the different documents in a proper format, and with all the relevant information added so that anyone who uses our project can use it directly without any hassle. We have also added descriptions of all the commands in different files so that the users can have an idea on how to use these commands. These files also contains the video description of how one can use these commands.

3 Zero Internal Boundaries:

A developer in most cases makes changes to only certain parts of the code base. But this does not prevent his from making changes to other parts of the code base. It's valid as long as the changes are justifiable. This practise ensures that the problems are fixed right when they originate. It is not necessary to provide multiple workarounds. It also indirectly helps the developer to gain a global perspective on the code base rather than sticking to a specific repository. The rubric has three tuples which could be mapped to this category: *evidence that the whole team is using the same tools: everyone can get to all tools and files* This indirectly translates the fact that every developer within the group know each and every software or tool that is being used in the system. This ensures that everyone is aware of how the project actually works rather than just focusing on one aspect of the project. *evidence that the whole team is using the same tools (e.g. config files in the repo, updated by lots of different people)* This ensures consistency which makes the process of knowledge transfer, sharing and discussion easier. *evidence that the whole team is using the same tools (e.g. tutor can ask anyone to share screen, they demonstrate the system running on their computer) evidence that the members of the team are working across multiple places in the code base* This point provides a proof of the fact that the developers have indeed worked on different aspects of the code.
We have done most of our coding part in Python using and IDE like Pycharm, so all of us are using the same tools. We all are pushing to the main branch in or repository which gives us the idea that all the contributors are important and we trust them completely to directly push into the main branch. All of us can run the same code in our system and can perform all the tasks from their end. The main config files in our repository are being continuously updated by all the members along with the cog files.

4 Tools Matter:

Every type of project requires a certain set of tools and resources which need to used in unison for the system to work efficiently. These tools help in keeping a track of all the work that is taking place in the system It also helps in a smooth on-boarding and knowledge transfer of a new person. A user doesn't have to manually enter every file and check for syntax and version control. Following are the tuples which map with this ideology. *a) Use of version control tools, b) Repo has an up-to-date requirements.txt file c) Use of style checkers d) Use of code for-matters. d) Use of syntax checkers. e) Code coverage f) test cases exist g) test cases routinely executed.*
We all use Github as our version control tool and have an up to date requirements.txt file in our repository. Pycharm handles the part of style checkers, code for-matters and syntax checkers. For code coverage we have used codecov and we are getting 58% code coverage. Numerous test cases exista and they all get executed everytime any push is made to the main branch.

5 Consensus Oriented Model:

The Linux community strictly adheres to the rule that a code cannot be merged into the main unless an experienced and respected developer approves it. This rule makes sure that the chances of kernel going haywire is significantly reduced. One group cannot make unwarranted changes to the code base at the expense of other groups. While such restrictions are imposed the kernel code base remains flexible and scalable. Following tuples seems to match with the understanding of the practise. *a) the files CONTRIBUTING.md and CODEOF-CONDUCT.md has have multiple edits by multiple people* This makes sure than everyone has contributed the the project and that all the decisions have been made by considering every team member's opinion. *b) the files CONTRIBUTING.md lists coding standards and lots of tips on how to extend the system without screwing things up c) multiple people contribute to discussions* All the decisions are taken unanimously by the entire team and stakeholders. *d) issues are discussed before they are closed* A peer review process of the implemented feature makes sure that minimal amount of bugs slip through thus maintaining the stability of the product. *Channel chat exists and is active* and finally *test cases: a large proportion of issues related to file handling cases*

6 Low Regression Rule:

Developers are always striving to upgrade the code base. However, this is not done at the cost of quality. This is one of reasons to introduce the no-regressions rule. According to this rule, if a given kernel works in a specific setting, all the subsequent kernels must also be able to work in this setting too. However, in case the system becomes unstable because of the regression, the developers waste no time in addressing the issue and getting the system back in it's original state. The tuple which could be mapped to this metric is: *features released are not subsequently removed*

References

[1] $https://medium.com/@Packt_Pub/linux-kernel-development-best-practices-11c1474704d6$