Media Analyzer: Best Practices

Jianlin Chen
jchen42@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Connor Dolan
cjdolan@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Joshua Mason
jlmason6@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Rohan Rao
rsrao@ncsu.edu
North Carolina State University
Raleigh, NC, USA

David White
dawhite4@ncsu.edu
North Carolina State University
Raleigh, NC, USA

1 INTRODUCTION

In this document, we will analyze the relationship between CSC510 Project 1's rubric and six of the Linux Kernel best practices. The best practices come from the lessons learned section of the 2017 Linux Kernel Development Report. The lessons come from over 26 years of highly successful experience.

2 SHORT RELEASE CYCLES

The goal of short release cycles is to ensure new code is quickly integrated into the system to reduce issues related to integration, as well as to ensure code that is integrated is actually working properly. Short release cycles are promoted by and related to multiple items on the rubric. The most obvious of these is the "Short Release Cycles" entry sharing the same name as the best practice. This item is a positive indicator of a team's work as it suggests code is frequently being produced and merged into the product. Therefore, short release cycles are also strongly related to the "Number of Commits" item on the rubric, as commits are a tangible way to quantify how many iterations a given code base went through. The quick releases are assisted by continuous integration, so the routine execution of test cases is another rubric item that encourages short release cycles.

One aspect of the short release cycle emphasized by Linux is that if developers miss a cycle, they know they will be able to include their work in the next one that isn't too far away. This way, there is "little incentive to try to merge code prematurely"[1]. This aspect is not fully captured by the rubric. Even though the rubric mentions having short release cycles, the scope of the project is too short to fully demonstrate this. An item such as "branches/features are testing locally and with automated testing before merges" may help with reducing premature merges rather than a broad short-release cycle criterion.

3 ZERO INTERNAL BOUNDARIES

The members of a team that operates with zero internal boundaries all have access to the entire code base and can use this access to fix any problems they encounter at the root. It assists developers in gaining a holistic understanding of the code and helps limit the amount of time team members spend blocked, waiting on others to complete their work. This practice is measured in the rubric through categories that include "workload is spread over the whole team" and "Evidence that members of team are working across multiple places in the code base", and more. The last category in particular is a strong indication of zero internal boundaries, since it suggests that developers both have an understanding of the code and are able to access and work on any part of it. The workload being spread over the whole team also supports this practice, since it means that team members didn't simply stop working if they finished a relatively easy task and migrated to other necessary work. Since this practice necessitates a deep understanding of the project, all documentation criteria on the rubric are relevant as well. The existence of how, what, and why documentation serves to aid developers that start to work on unfamiliar sections of the code base. Similarly, evidence of common tools and configuration helps demonstrate that the developers all had the capability to work on any part of the code that they encountered problems with.

4 THE NO-REGRESSIONS RULE

The no-regression rule ensures that if a project works in a specific setting, all future versions must work there too. This is important for building trust with users and developers, giving assurance that upgrades will not break their systems. This way, people will follow along as new features and capabilities are added. The rule is measured in the rubric in a couple of different ways. The first way is the "use of version control". Version control is an essential tool for tracking changes, assisting with debugging, and ensuring that a working product is always available. It also helps ensure that team development does not result in conflicting changes. This is related to no-regressions by helping to prevent breaking systems. Another way no-regressions is measured by the rubric is the use of testing and "test cases are routinely executed". GitHub Actions is a good example of this. These types of automated tests automatically show whether or not new versions of code break previously working tests. This is one of the most influential ways to enforce the no-regressions rule. Lastly, "other automated analysis tools" is a way to enforce the no-regressions rule similar to the previous one. These tools can analyze things like security vulnerabilities that general tests may miss, helping ensure that future versions of a project do not have an increase in issues or unresolved issues.

5 CONSENSUS-ORIENTED MODEL

Taking a consensus-oriented approach means that proposed changes to the system are accepted by all members of the team. More practically, this means that code written by developers must be approved by someone on the team and that no one person can push code that is detrimental to somebody else's work. This is outlined in

the rubric in the "issues are discussed before they are closed". This ensures that developers don't simply push tons of code to fix a bug or implement a feature without consulting the rest of the team. Evidence that the whole team is using the same tools and everyone can run every aspect of the project are important guidelines from the rubric here, since developers should be testing the work of others to gain consensus. This consensus is also supported by the existence of a strong CONTRIBUTING.md file, since all developers have to follow these standards when updating the code base.

An item that could strengthen the relationship between the rubric and this Linux rule would be "there is a review process for pull requests". While discussion on issues helps establish a consensus-oriented model, the most important aspect of establishing this model is how PR's are reviewed and accepted. If PR's are consistently reviewed by varying members of the team before being accepted, this shows a strong consensus-oriented model.

6 DISTRIBUTED DEVELOPMENT MODEL

The distributed development model is a practice in which code review and integration of new code into the larger system is divided evenly among the entire development team. This benchmark is evaluated on the rubric in categories like "number of commits by different people" since this provides evidence that the development work done is distributed relatively equally among team members. Since it relates to code review as well, the discussion of issues before they are closed is another aspect of the rubric that is related to the distributed development model. Since anyone is free to discuss issues, provide comments, or ask questions, the overall examination of new features or existing bugs can be spread throughout the team. Since this approach relies on everyone making updates to and reviewing the code base, it's important that everyone has access to all aspects of the code base and the capabilities to run that code. Additionally, code should be of the same general format, so the use of style checkers and code formatters are very useful. Proper use of version control is also very important here since the team should be able to compare new functionality to the way the system worked before, likely on different branches.

7 REFERENCES

[1] Jonathan Corbet and Greg Kroah-Hartman. 2017. 2017 Linux Kernel Development Report. Retrieved October 9, 2022.