An Empirical Investigation on the Prevalence, Impact, and Manifestation of Rework Commits in the Merge Requests Mechanism

by

Julien LEGAULT

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLEMENT FOR A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, DECEMBER 20, 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Francis Bordeleau, Thesis Supervisor
Department of Software and Information Technology Engineering, École de technologie supérieure

Mr. Mohammed Sayagh, Thesis Co-supervisor
Department of Software and Information Technology Engineering, École de technologie supérieure

Mr. Julien Gascon-Samson, President of the Board of Examiners
Department of Software and Information Technology Engineering at École de technologie supérieure

Mrs. Ghizlane El Boussaidi, Member of the jury
Department of Software and Information Technology Engineering at École de technologie supérieure

THIS THESIS WAS PRENSENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND PUBLIC

NOVEMBER 29, 2022

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGMENT

I would like to express my deepest appreciation to my director Prof. Francis Bordeleau who guided me on the path through the metaphorical forest of completing my Master's degree and thesis. Francis enabled me to take part in the life of department beyond just the thesis and provided me with many opportunities to learn under his mentorship. I also am grateful for my co-director Dr. Mohammed Sayagh who is at the tip of the spear about the research being done in our field. His knowledge shaped the direction I was able to take with this work.

This research would not have been possible without the close contribution of Dr. Bassem R. Guendy (CICD DevOps Specialist at Kaloom) who went above in his professional duties to help me understand his industrial world. Our industrial partner was very generous with their time and financial support to help me achieve this thesis.

Finally, I would like to give my biggest acknowledgement to my parents, who's unconditional love and unwavering support throughout my life and education has been invaluable and essential to my happiness and to the accomplishment of this thesis.

Une étude empirique sur la prévalence, les impacts et les manifestations des Rework Commits dans le mécanisme de demandes de fusion

Julien LEGAULT

RÉSUMÉ

La motivation principale du DevOps est d'offrir de la valeur en continu aux clients. Un des bénéfices de cette approche pour Kaloom, notre partenaire industriel, est d'accélérer le flux de développement en réduisant le temps d'attente des diverses phases du processus de livraison du logiciel. Pour permettre l'amélioration continue, nous nous sommes concentrés sur la détection de bogues le plus tôt possible dans la chaine CI/CD pour améliorer le flux DevOps.

Le but de cette recherche était d'améliorer le processus de revue de code. Pour accomplir ce but, nous avons observé le processus de notre partenaire industriel en étudiant les *rework commits* des demandes de fusion (MR). Les questions de recherche abordées sont les suivantes : RQ1.: Quel est la distribution des *rework commits* chez Kaloom ? RQ2.: Comment les métriques reliées à l'effort changent t'elles avec le nombre de *rework commits* ? RQ3.: Comment la variation du nombre de fichiers associés à un MR influence t'elle le nombre de *rework commits* ?

Nos résultats montrent que le nombre de MR avec plusieurs *reworks* continue d'augmenter et nous suggérons de développer des approches pour réduire les *reworks* et faciliter le processus d'intégration de code. Les *reworks* peuvent avoir un impact significatif sur le temps d'attente et peuvent parfois être attribués à des fichiers manquants ou à la façon d'utiliser GitLab. Pour les MR avec plus de huit *reworks*, une médiane de cinq fichiers manquants a été trouvée, suggérant le besoin de mieux identifier les dépendances interfichiers avant d'ouvrir un MR.

Mots-clés: *rework commit*, *code rework*, DevOps, demande de fusion

An Empirical Investigation on the Prevalence, Impact, and Manifestation of Rework Commits in the Merge Requests Mechanism

Julien LEGAULT

ABSTRACT

The overall driving motivation behind DevOps is to continuously provide value to customers. One benefit of DevOps that is of particular interest to our industrial partner, is accelerating the flow of development through the reduction of the lead time of various phases of the software delivery process. To keep the continuous improvement process going, we needed to focus on means of improving the DevOps flow, such as improving detection of bugs as soon as possible in the review process.

The goal of this research is to help improve the review process. To accomplish this goal, we provided a first overview of the code review process of our industrial partner by looking at rework commits in the submitted merge requests (MR) to detect possible issues earlier in their CI/CD pipeline. The emerging research questions are studied for this purpose: RQ1.: What is the distribution of rework commits across MR at Kaloom? RQ2.: How does effort related metrics change with the number of rework commits? RQ3.: How does the variation in the number of files associated with an MR affect the number of rework commits?

Our findings shows that the number of MR with multiple reworks is continuously increasing, and we suggest the need for approaches to reduce rework for a smooth integration process. Rework can have a significant impact on the MR lead time and can sometimes be attributed to missing file dependencies or way of working with GitLab. For MR with more than eight rework commits, a median of 5 missing files was found, suggesting the need for better identifying missing dependencies before opening merge requests.

Keywords: rework commit, code rework, DevOps, Merge Request

TABLE OF CONTENTS

Page

CHAPTER 1 RELATED WORK AND TOOLS 7
1.1 Code Review 7
1.2 Commit Classification 9
1.3 Definition of Rework Commits 10
1.4 Thesis of Seyedbehnam Mashari (2022) 10
1.5 GitLab Analytics 11

CHAPTER 2 INDUSTRIAL CONTEXT OF THE RESEARCH PROJECT 15
2.1 Kaloom Software Delivery Process 15
2.2 Kaloom Code Review Process 17
2.3 Kaloom Product Teams 17

CHAPTER 3 METHODOLOGY AND TOOLS 19
3.1 Research Methodology 19
3.2 MR Data Extraction Tool 22
3.2.1 Tool Implementation 22
3.2.2 Technical Challenge 27
3.2.3 Validation 27
3.2.4 Contextual and technical limitations 29
3.2.5 Other tools 31

CHAPTER 4 EMPIRICAL RESULTS 33
4.1 What is the distribution of rework commits across MR at Kaloom? 34
4.1.1 Motivation 34
4.1.2 Approach 34
4.1.3 Results 35
4.2 RQ2. What is the relation between rework commits and the lead time? 38
4.2.1 Motivation 38
4.2.2 Approach 38
4.2.3 Results 39
4.3 RQ3. How does the variation in the number of files associated with a MR affect the number of rework commits? 43
4.3.1 Motivation 43
4.3.2 Approach 43
4.3.3 Results 43

CHAPTER 5 DISCUSSIONS AND CONCLUSION 47
5.1 Summary 47
5.2 Research Limitations 48

5.3 Threat to Validity 49
5.4 Future Work 50
5.5 Final word 50

APPENDIX I DETAILS ON TECHNICAL CHALLENGES OF THE MR DATA EXTRACTION TOOL 53

LIST OF REFERENCES 59

LIST OF TABLES

Page

Table 2.1 Product Team Groups at Kaloom 18

Table 3.1 Rework Size Groups 21

Table 3.2 MR Data Extraction Tool Data and Metrics directly used in this thesis ...23

Table 3.3 MR Data Extraction Tool Data and Metrics not used in this thesis.....24

LIST OF FIGURES

Page

Figure 2.1	Kaloom's software delivery process	15
Figure 3.1	Research Methodology diagram	19
Figure 4.1	Distribution of MR Count per Rework Size across the Kaloom Groups	36
Figure 4.2	Evolution of the merge requests with a medium and large rework	37
Figure 4.3	Evolution of the merge requests with a medium and large rework	37
Figure 4.4	Boxplot of Lead Time per Rework Size across the Kaloom workgroups	39
Figure 4.5	Boxplot of the average time between reviews	40
Figure 4.6	Size of the merge requests according to their rework size	41
Figure 4.7	Spearman correlation between the MR size and the rework commits	42
Figure 4.8	The percentage of MRs that exhibits each of the rework sizes with a non-null delta	44
Figure 4.9	The percentage of MRs that exhibits each of the rework sizes with a non-null delta	45
Figure 4.10	Positive (files that are missed in a merge request) vs negative (files that did not have to change in a merge request) delta files for the three rework groups	46

LIST OF ABREVIATIONS

API	Application programming interface
CI/CD	Continuous integration and continuous delivery pipeline
CP	Control Plane
DP	Data Plane
LOC	Lines of code
MP	Management Plane
MR	Merge request
OS	Operating System
PF	Platform Group
QL	Quality level
TR	Trouble report

INTRODUCTION

0.1 Context and Motivations

Kaloom™ is a distributed data center networking software company located in Montreal. Leveraging 5G and Cloud Edge Fabric technologies, they developed a Data Center and Edge Networking software solution for their Telco, Data Center and Cloud Services Provider clients (https://www.kaloom.com/). The company benefits from using a DevOps approach to development which enables faster and economical software development and delivery.

Kaloom uses a DevOps approach as their way of working in four different groups within the company. The overall driving motivation behind DevOps is to continuously provide value to customers. One such benefit that is of particular interest to Kaloom is accelerating the flow of development through the reduction of the lead time of various phases of the software delivery process. In the last few years, a significant effort has been made on their part to reduce the lead time of the build and packaging phase from an hour and a half to 11 minutes.

This success demonstrates their commitment to improve the Continuous Integration and Continuous Delivery pipeline (CI/CD pipeline), which falls under the First Way of DevOps (Kim, Humble, Debois & Willis, 2021). However, the concept of DevOps flow can't be reduced to an optimal CI/CD pipeline. Optimisation of the DevOps flow also involves the importance of breaking local, siloed process, into a global flow and improvement of the whole value chain. To accomplish this, we need to do more than improve each task individually. The means to accomplish this high-level objective includes making work visible and avoiding propagating defects further down the value chain. Kaloom already implemented good practices to achieve this, like their efficient CI/CD pipeline, but they reached a limit to which this local process cannot easily be improved any further. To keep the continuous improvement process going, we need to take a step back and focus on other means of improving the DevOps flow, such as making work easier to visualise and improve detection of bugs as soon as possible in the review process.

The goal of this research is to help improve the review process to maximize its impact on quality. To accomplish this task, we measured and provided a first overview of the code review process by looking at rework commits and changes in the submitted merge requests (MR)

0.2 Problem and research questions

The problem addressed by the research questions is to provide a first step into improving the Review phase while using the MR data available to us. To accomplish this goal, this research offers an overview of the work being done at the MR level and investigates how the MR process can be improved to detect possible issues earlier that would otherwise be found later in the CI/CD pipeline. From this problem emerges our research questions.

RQ1.: What is the distribution of rework commits across MR at Kaloom?

The goal of this research question is to provide a global picture of reworks commits across all MR processed between 2020 and 2021 by the four Kaloom working groups. This research question also positions the main rework metric on which the other research questions can use statistical analysis to determine correlation between the number of reworks and other metrics. Prior to understanding relationships between metrics, we must first understand how MR with reworks are distributed across Kaloom. Before this study, Kaloom had no means to visualise the amount of rework commits done after MR were open. To provide methods and tools to enable people managing the review process at Kaloom to analyse the metrics and data to be interpreted, we must first align our understanding of reworks in partnership with Kaloom to understand the context in which they occur.

RQ2.: How does effort related metrics change with the number of rework commits?

The second research question provides a clearer picture of the MR situation in terms of metrics which are related to developer effort. This overview is a good first step in making the work visible, a key principle in improving the DevOps flow. This research question focuses on two

core attributes of a MR which best represent work being done. The first core attribute of the MR used to represent work is the *Lead Time*. The *Lead Time* can be viewed as the total time taken to work on a MR and offers temporal view of that work. The second attribute is the MR Size, which is a common measure of the amount of work done in a MR. The concept of size could be expanded to include the amount of files, commits and lines of code. The number of lines of code (LOC) was used to represent the *MR Size*. This choice benefits from using the simplest and smallest unit of size to represent the *MR Size*. The goal of this research question is to understand the relation between the Lead Time, MR size and rework commits.

RQ3.: How does the variation in the number of files associated with a MR affect the number of rework commits?

Kaloom managers have an interest in identifying potential dependencies issues in MR. For this reason, we explore the impact of the number of files changed between the moment a MR is created and the moment it is closed or merged. We specifically look at how rework commits are related to new or missing files in the resolution of the MR. This also gives Kaloom an idea of how much of the MR review effort is related to issues with file dependencies.

0.3 Contributions

This thesis makes both scientific and technical contributions. From a scientific perspective, the research methodology (see chapter 3) developed to investigate the three research questions (described in the previous section) has allowed better understanding of the code review process at Kaloom and resulted in findings that will be used to improve the process. Also, the research methodology developed in this thesis will be used in other research project of the Kaloom-TELUS ÉTS IRC to conduct a similar study with other companies, in particular TELUS.

At the technical level, a tool, called GitLab MR Data Extraction (see section 3.2), was developed to extract MR data from GitLab and compute a set of metrics that constituted the basis of the investigation of the research questions. Besides being used at the core of the

research project of this thesis, the GitLab MR Data Extraction tool is an important contribution of this research project to the Kaloom-TELUS ÉTS IRC. The tool already contributed to the research project of Mashari (2022) and will be used to analyze the code review process of other companies. Also, to increase its industrial impact, the tool has been made available in open source (to allow other companies to conduct similar analysis on their process).

0.4 Thesis Outline

Chapter 1: Related Work provides a review of the literature and existing tools relevant to this thesis. It is broken down into five relevant topics: code review, commit classification, definition of rework commits, the Master thesis of Seyedbehnam Mashari, and GitLab Analytics tools.

Chapter 2: Industrial Context describes our industrial partner Kaloom. It contains two sections. The first section, **Kaloom Software Delivery Process** explains the Kaloom software delivery process to allow the reader to understand the evolution of product quality through the evolution of MR quality within the software delivery process. This section also allows the reader to position the Review Phase within the software delivery process with a visual representation of the Kaloom software delivery process. The second section of this chapter, **Kaloom Code Review Process** describes the four groups who uses this software delivery process and the important differences between them that are taken into consideration for this research.

Chapter 3: Methodology and Tools describes the methodology used in this research as well as the tools we developed and used. The first section of the chapter presents the methodology and the research questions associated to this study. The second section details the MR data collection process through the development of software tools for this research. The GitLab MR Data Extraction Tool is explained in detail with its metrics.

Chapter 4: Results Analysis presents the goal, approach and results for each research question. The research findings are also described in that chapter.

Chapter 5: Discussions and Conclusion provides discussions around our findings as well as the research's limitations and threats to validity. The chapter ends with suggested future work related to our findings.

CHAPTER 1

RELATED WORK AND TOOLS

The goal of this section is to discuss related work to the code review mechanism and classification of commits, which are the closest work to our study as we study a specific type of commits (rework) in the code review mechanism. We also explain the importance of another master's thesis that we contributed with during our research. Finally, we look at why the existing tools in GitLab Analytics aren't sufficient to perform our data collection.

1.1 Code Review

Code review constitutes a core activity of most industrial and open-source software development processes. Many papers have studied the role and impact of code review on different aspects of the software development process and of the resulting software quality. For example, McIntosh, Kame, Adams & Hassan (2015) studied the relationship between modern code review and long-term software quality in four open sources projects using explanatory machine learning models. They conclude that poorly reviewed source code ends up with a negative impact on software quality. Ram, Achyudh, Sawant, Castelluccio and Bacchelli (2018) found that reviewability of a code change was linked to factors like change descriptions, size, and coherent commit history. They conducted an experiment where they employed 10 professional developers to review 98 code changes through their reviewability evaluation tool. Another study (Ebert,Castor, Novielli and Serebrenik, 2019) also tried to understand the confusion that slows down the review process, mostly due to reviewers not understanding the intention behind a code change. They built a confusion framework around the review process, artifacts, and developers to study the relationship between them. Their findings support the idea that the missing rationale, familiarity with the code and discussions around non-functional requirements are the most common source of confusion. Thongtanunam, McIntosh, Hassan & Iida (2016) studied the relation between code modification ownership, code review and post-release defects. They found that reviewers can review a code change of a module on which they did not make any code change. They also

found that having practitioners that do not have enough expertise on a module reviewing that module can increase the chances of post-release defects. Davila & Nunes (Davila & Nunes, 2021) also remarked via a systematic literature review that open-source projects are used in 85% of studies on code review. They report that the goal of the existing studies is to understand the impact in terms of benefits and challenges of the code review mechanism. They also report other studies that tried to improve the code review mechanism itself. They also found studies that shows human expectations linked to the code review activity. The participants expect knowledge sharing and learning when performing code review as it is a good opportunity to synchronize the understanding of the code within the team.

As this last line focuses on studying the impact of the code review mechanism, another line of research focuses on how to improve the code review mechanism itself. For example, Thongtanunam, Tantithamthavorn, Kula, Yoshida, Iida, Matsumoto (2015) found that code reviewer assignment related problems can slowdown the lead time of code reviews. They also developed a recommendation system called RevFinder to suggest reviewers. The intuition behind their approach is that reviewers who previously reviewed a file in a similar path to the new code change are the better to review the new code change. Their approach is able to recommend the appropriate reviewer at a rank of 4. Previously, Thongtanunam, Kula, Cruz, Yoshida, and Iida (2014) proposed an algorithm to recommend reviewers based on file path similarity. The evaluation of their approach on open-source projects shows an accuracy up to 77.9%. The idea of using automated tools to help perform code review is also explored by Chatley, R., & Jones, L. (2018) where they proposed a tool called Diggit. A different approach using a non-dominated sorting genetic algorithm to solve the multi-objective problem of maximising reviewer expertise while minimizing reviewer's workload was also developed by Chouchen, Ouni, Mkaouer, Kula and Inoue (2020) with efficient results. Recent studies are exploring more advanced review recommendations. Hong, Tantithamthavorn, Thongtanunam (2022) use machine learning models to predict which lines of the code might be more problematic and require further attention from the reviewers. Their approach, called REVSPOT, has a high accuracy of 81% and 93%. A recent empirical study by Hirao, McIntosh, Ihara, Matsumoto (2022) shows that 15% to 37% of the patches can have a divergent

score. They also found that abandoned patches for external issues suffer from divergent review scores.

Our work is complementary to this line of research as we focus on how to improve the code review by reducing the amont of changes that happen during the code review, which we refer to as rework commits. We wish to first understand these commits before suggesting any solutions to optimize the number of (not completely remove) rework commits and their associated costs. Our empirical study shed light on the prevalence and impact of rework commits suggesting future work to develop approaches to minimize the rework commits.

1.2 Commit Classification

Studies shows the impact that commit messages can have on code change classification and its usefulness in rework identification. Analysing these commit messages can be used to identify the kind of work the commit represents. Corrective commits are literally a MR rework commit if they are performed as the result of a code review during a MR.

Yan, Fu, Zhang, Yang, and Kymer (2016) studied commit classification using a semantic analysis model and labeled samples they developed on five open-source projects. They used words such as “fix”, “create” and “correct” to determine the type of commit and established probabilistic relationship. In their second research question, they tackled multi-category commits classification. Their last research question quantified the distribution of single and multi-category changes. Ghadhab, Jenhani, Mkaouer and Messoud (2021) looked at the categorization of code change commits to help with maintenance tasks and managing technical debt. They developed a model named BERT which uses language processing to categorize commits between three type of maintenance tasks: corrective, perfective and adaptive. Their tool achieved 79.66% accuracy representing an 8% improvement over the state-of-the-art model. Amit & Feitelson (2021) suggested the creation of a Corrective Commit Probability metric to estimate the effort invested in bug fixing. They computed this metric for 7557 large projects on GitHub and used it to form an investment scale. Their findings shows that the

bottom 10% of projects spends around 6% of effort on bug fixing while the top 10% of projects spent at least 39% of effort on bug fixing. They also found that the lower investment profiles were linked to smaller files, lower coupling, use of language like C# and JS, younger projects, better perceived code quality and lower code churn and number of developers (Amit & Feitelson, 2021).

While our work is not to classify commits messages under different categories, we leverage simple heuristics to identify rework commits. In future work, we will leverage and evaluate more advanced techniques to identify and classify rework commits.

1.3 Definition of Rework Commits

Reducing unnecessary rework and their associated costs is a prevalent theme in the lean methodology and therefore in DevOps. Rework itself can take many forms, and its definition can vary depending on the context. Forsgren. Humble & Kim (2017) defines unnecessary rework as a value lost on non-value-added work. Rework is qualified as avoidable work through improved processes (Humble & Kim, 2017). This broad definition of rework can be understood as rework originating from various parts of the software delivery process. In our thesis, we specifically use the term rework to mean corrections that were performed inside a MR. Our scope of rework usually implies a rework commit inside a MR, which are the Git commits that are being added to a MR as the result of a code review.

1.4 Thesis of Seyedbehnam Mashari (2022)

The research of Mashari (2022) and the development of the MR Extraction tool is part of the research activities of the Kaloom-TELUS ÉTS Industrial Research Chair (IRC) in DevOps. In this context, we performed this research in close collaboration with Seyedbehnam Mashari, who wrote his thesis based on the same industrial case. For this reason, our MR Extraction Tool was designed and developed with both thesis's research questions in mind. His work focused on different research questions that are closely related to this thesis. In his thesis, Mashari investigated the potential correlation between Lead Time and Size of an MR. His

conclusion, based on the analysis of the four product groups at Kaloom, was that there is no correlation between Lead Time and Size of MR. This result allows us to treat Lead Time and MR Size as independent metrics in this work. His thesis also showed that Kaloom groups who are using hardware testing tend to have higher average Lead Time. This observation is also useful for our analysis, as we are in position to further elaborate on those results with our studied metrics. In his research, he studied the behaviour practices of the Kaloom developers during a sprint. We do not elaborate on that topic in our research, but we do look at the number of commenters per MR as a measure of participation. As with his thesis, we also performed manual analysis of some MR for validation purposes with Kaloom, although his thesis was more specifically focused on understanding outliers and exceptions. In our case, we are trying to explain the observed results with those example MR.

1.5 GitLab Analytics

Gitlab Analytics (www.gitlab.com) provides a set of relevant DevOps and analytics features, which includes a set of basic metrics related to the Review Phase. In particular, the Code Review (CR) and Merge Request (MR) sections of GitLab Analytics are directly relevant to our research. However, they do not provide some of the key metrics we need to answer our research questions.

For example, the Code Review section of GitLab Analytics offers a metric called *Review Time* that provides a measure of the time elapsed since the creation of a currently open MR. This metric is relevant, but insufficient for our needs, as we are interested in all MRs for a given period, including the ones that are *closed* or *merged* that are not collected by the *Review Time* metric.

GitLab Merge Request Analytics also includes a throughput chart that provides the number of merged MR for each month as well a table with MR Details. Because the throughput chart is month-based, it does not provide the level of granularity we need and therefore cannot be used for our study. Regarding the metrics provided by the GitLab MR Table, some of them can be

used by our MR Extraction Tool. However, they are not sufficient. The MR table only contains merged MR with the *Date Merged*, *Time to Merge*, *Commit count*, *Pipelines Count*, *Line changes* and *Assignees*. Most of these metrics are relevant, but only available for *merged* MR and we are again interested in all MR, *open*, *closed,* and *merged*.

The name of the metrics used by GitLab Analytics is also problematic as it may vary depending on the MR state. For example, *open* MR have a *Review Time* in the code review section but *merged* MR have a *Time to Merge* in the merge request section. Both terms represent the same concept for our purposes but have different names. In our MR Extraction Tool, we use the generic single terminology of *Lead Time* to represent the time elapse between the creation of a MR and when it is merged or closed. If the MR is *open*, the *Lead Time* is still ongoing but returns the current value at the time of extraction.

The *Date Merged* from the *GitLab Analytics Merge Request* table is also relevant, but we do not directly include the Creation Date, which our tool also needs to some critical functions like calculating lead time and data extraction configuration. The number of commits in a MR is also a key metric used in our research that we cannot obtain for *closed* MR with GitLab Analytics. The *Line change* metric is a relevant metric but is calculated in an inconvenient way. We also need a "*Merge Request Size*" metric, but the existing *Line change* metric provides the lines of code (LOC) added and the LOC removed. Our MR Extraction Tool will therefore need to implement a metric to represent the MR size to answer our RQ1 and RQ2. Finally, the *Assignee* is the person assigned to review the MR. This is a relevant metric, but we need to take a deeper look into the interactions that are involved in a MR review. This *Assignee* metric, which represents the MR reviewers assigned to the MR, is not enough to get an understanding of the review process. Our tool renames this *Assignee* as *Reviewer* but will also try to differentiate the *Assigned Reviewer* from the actual active reviewers, called *True Reviewer* in our tool.

The exploration of the *Gitlab Analytics* section of GitLab shows that the MR information available on the platform is incomplete and insufficient for our study. It contains useful metrics

that can be combined and others that should be explored in more details. For this reason, we justify the need to develop our own tool, The *MR Extraction Tool* is used for the data collection of this study, but also constitutes a technical contribution useful to other researchers in the DevOps and Code Review field.

CHAPTER 2

INDUSTRIAL CONTEXT OF THE RESEARCH PROJECT

In this chapter, we describe the industrial context of our industrial partner involved in this research project. Section 2.1 explains the software delivery process used by Kaloom and take a deeper look at its Review phase. Section 2.2 details the relevant context around Kaloom's code review process. In section 2.3, we discuss the specifics of the different Kaloom product teams.

2.1 Kaloom Software Delivery Process

The following figure illustrates Kaloom's software delivery process from the earliest moment code is written until it's ready for deployment in production.

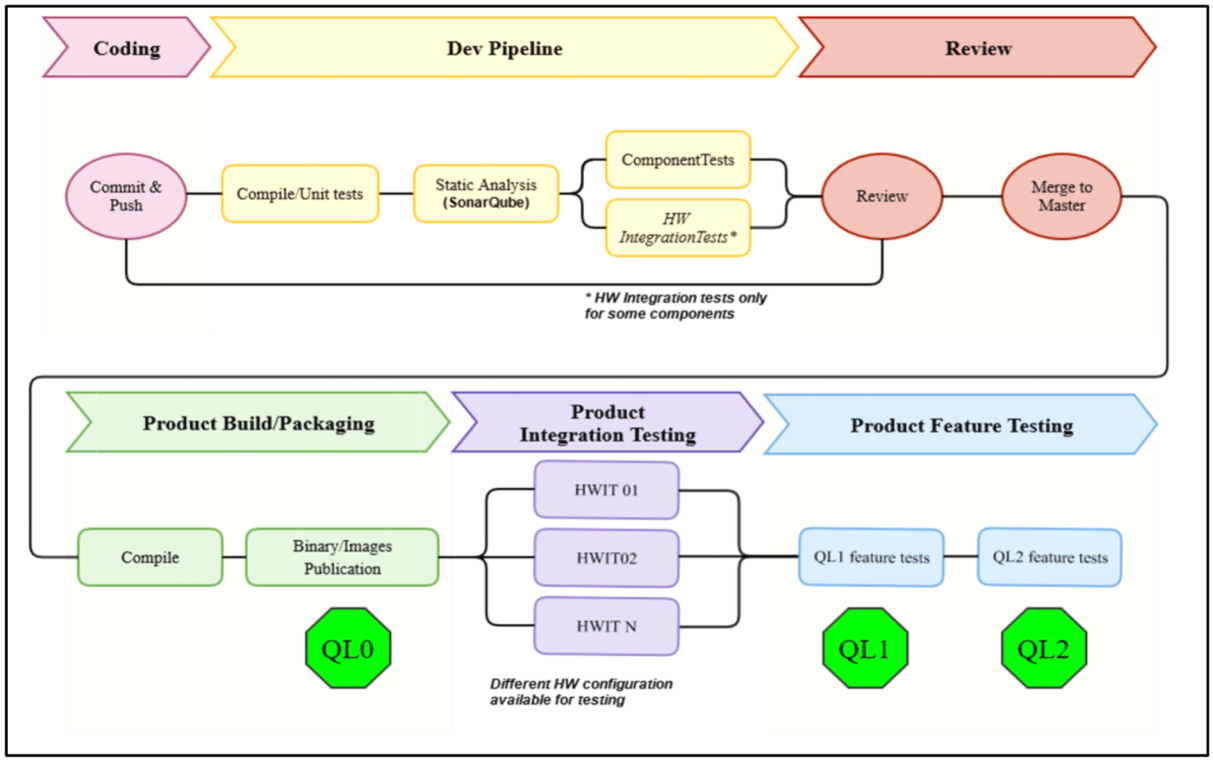

Figure 2.1 Kaloom's software delivery process

The Kaloom software delivery process shown in figure 2.1 is mainly divided in six phases represented by arrows: *Coding*, *Dev Pipeline*, *Review*, *Product Build/Packaging*, *Product Integration Testing*, and *Product Feature Testing*. Figure 2.1 also shows the steps of each of the phases in rectangles for automated tasks and circles for manual tasks. The figure also represents the quality levels with green octagons. The *Coding* phase contains the *Commit & Push* activity, where code is developed, and version controlled under GitLab.

The *Dev Pipeline* phase consist of a series of automated tests activities, where we find *Compile/Unit tests*, *Static Analysis*, *ComponentTests* and *HW IntegrationTests*. The static code analysis is performed by SonarQube, an automated tool integrated in the GitLab CI/CD pipeline. The compilation and unit tests are also automated by the GitLab CI/CD pipeline. For some projects with longer or riskier hardware components, automated hardware tests are also run in this phase before the Product Integration Testing phase.

The *Review* phase contains the two activities that are central to our research. Taking a closer look at the Review phase, we find the *Review* activity. This activity uses the GitLab Merge Request mechanism to facilitate code review by other developers. The *Review* activity is the focus of this research and further detailed in section 2.2. Depending on the code review, the process might fall back to the beginning of the *Coding* phase if code changes are required or move on to the *Merge to Master* activity. The code *Merge to Master* activity is also done via the same MR mechanism from the code review but isn't necessarily done the same way across the different groups at Kaloom. The Kaloom product is architectured as microservices/components. The component/microservice represents a git submodule of the main product git repository. Once there is a merge to master branch of any component/microservice git submodule, this triggers product packaging and first QL0 is added upon successful packaging of the product.

Figure 2.1 shows the first Quality Level (QL0) is achieved in the *Product Build/Packaging* phase. It also informs us that the *Product Integration Testing* phase allows different

configurations for the integration tests. The *Product Feature Testing* phase is centered around attaining QL1 and QL2.

2.2 Kaloom Code Review Process

During the software delivery process there is an opportunity to enhance product quality through code quality during the code review activity. For Kaloom, the code review is part of the Review Phase of their software delivery process. This phase uses GitLab, a version control tool and platform for CI/CD automation as well as JIRA, an issue and project tracking software. JIRA is used to manage the trouble reports (TR) which contains a higher-level description of issues, while GitLab is responsible for tracking work done on the code base through merge requests (MR). Therefore, a tacit link exists between a TR and the MR which solves it. The same MR can also be related to more than one TR.

Over the last two years, Kaloom started showing an interest in traceability between the TR and their related MR to improve the Review phase. They desire to see the impact of a MR on product quality, notably by looking at new TR resulting from previous MR. Currently, there is not a complete coverage of traceability between the two tools, making code change impact analysis difficult. Moreover, this research team has access to the MR data and a limited access to the Kaloom GitLab platform through the help of Kaloom's senior CI/CD DevOps specialist.

2.3 Kaloom Product Teams

This section details the different groups at Kaloom. These four groups differ by their responsibilities, sizes and programming languages. We describe each group in Table 2.1 and investigate how these attributes impact our findings in Chapter 4.

Table 2.1 Product Team Groups at Kaloom

Group	Description	Programming Language	GitLab Projects	Group Size
Control Plane	Pathing for packets and frames	Golang	21	11
Data Plane	Packet and frame forwarding between interfaces	P4 (testing in Golang)	5	15
Management Plane	Management and Communication with the Fabric using YANG models	Golang	9	7
Platform	Low-level platform management	Red Hat Linux and OpenShift	22	8

In Table 2.1, we describe each group's main responsibility, programming language and sizes. We note that Golang is the main programming language for the Control Plane (CP) and Management Plane (MP) and they also have the most projects. The Platform (PF) group is unique by its low-level orientation around an Operating System (OS) than the other groups. The Data Plane (DP) group is also differentiated by it's programming language which uses mostly P4 for networking devices. SonarQube isn't used with this language, which means there is no automated static code analysis available for the DP group. Both DP and PF groups relies on hardware integration testing where CP and MP groups uses software component testing. We are curious to see how these particularities and similarities between the Kaloom groups may impact our results.

CHAPTER 3

METHODOLOGY AND TOOLS

In this chapter, we present the research methodology used in this project (section 3.1) and the tool, called GitLab MR Data Extraction, that was developed to support the analysis of the MR data (section 3.2). The first section of the chapter presents the methodology for the study of the impact of different metrics on the rework commits. This section explains how MR rework was determined to be used as a measure of MR quality for this research. The tool section (section 3.2) describes the MR data collection process through the development of software tools in order to conduct this study.

3.1 Research Methodology

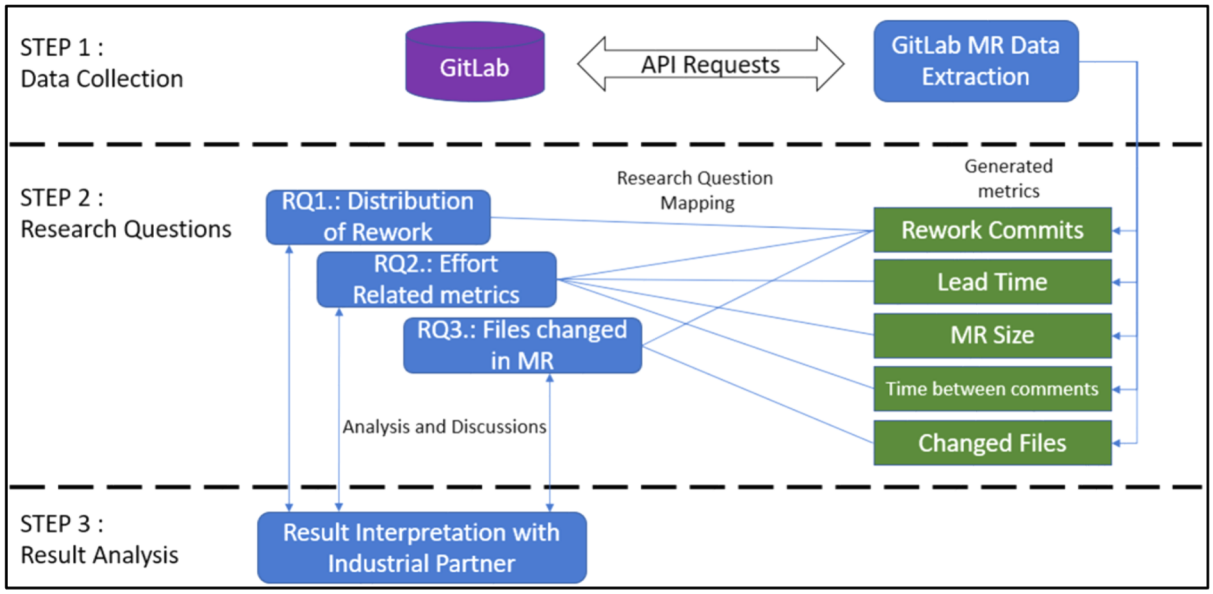

Figure 3.1 Research Methodology diagram

Figure 3.1 shows the methodology used in this research study. The process is composed of three steps : Data Collection, Research Questions, and Result Analysis. GitLab, which provides the required external data for this study, is represented by a purple cylinder while the research

activities are represented by blue boxes. The set of metrics generated by the GitLab MR Data Extraction tool during the data collection step that are used as a basis to answer the research questions are represented in green boxes.

Step 1: Data Collection

The first step, Data Collection, consists in extracting Merge Request (MR) data from GitLab, using its public API (https://docs.gitlab.com/ee/api/api_resources.html). For this purpose, we developed a tool, called GitLab MR Data Extraction (represented by the top blue rectangle in figure 3.1), described in section 3.2. The GitLab MR Data Extraction tool connects to the GitLab repository and makes successive API requests to fetch the MR data. The results from these API requests to GitLab are then processed and compiled into a CSV file, containing all of the tool's metrics for each extracted MR.

Step 2: Research Questions

This step aims at using the metrics produced by the GitLab MR Data Extraction to investigate our research questions. Figure 3.1 illustrates the list of research questions (represented by blue rectangles) and their associations with the metrics (represented by green boxes) used in their investigation. For example, the metric Rework Commit is used for all research questions (RQ1, RQ2, and RQ3). For RQ2, the efforts metrics that were selected are the MR lead time and MR size. We used changed files metric to answer RQ3.

For this research project, the investigation of the three research questions is based on all of the MR of the four Kaloom groups for the period of January 1st, 2020 to August 12th, 2021. This accounts for a total of 8,878 MR.

Step 3: Results Analysis

The purpose of step 3 is to analyze the results and discuss them with people responsible for supporting the DevOps process at Kaloom. We presented our results to Kaloom managers and discussed our interpretation with them. Following manual inspections of some MR with these experts, we can further our understanding of the impact of rework on the code review process.

An iterative discussion was required to be able to work efficiently with Kaloom, as our comprehension of the MR data evolved.

Rework Size

We noticed that discretely analyzing each rework count was too fine grain to allow making meaningful observations (leading to interesting findings). For this reason, to make the analysis more meaningful, we clustered the MR in four rework commits size groups: No Rework, for MR that have no rework commit; Small, for MR that have one or two rework commits; Medium, , for MR that have between three and seven rework commits; and Large, for MR that have more than seven rework commits. Table 3.1 shows each rework group and the amount of rework commits they contain.

Table 3.1 Rework Size Groups

Rework Size	Rework commit range
No Rework	0
Small	1 - 2
Medium	3 - 7
Large	8 +

3.2 MR Data Extraction Tool

A main part of this research project was focused on the development of the MR Data Extraction tool. The tool, which constitutes a main technical contribution of this thesis, was developed in collaboration with Kaloom as a general purpose and extensible research tool to extract MR data from GitLab using its public API (https://docs.gitlab.com/ee/api/api_resources.html). It was developed to provide Kaloom with the information they needed to better understand the MR review process based on different research questions (including the ones addressed by this thesis and the thesis of S. Mashari (2022)).

As discussed in the related work section, the development of the tool was motivated by the fact that the functionality provided by GitLab Analytics was insufficient to conduct our research. GitLab Analytics provides a basic suite of features that allows obtaining data and metrics related to MR, but it is missing several of the metrics we need. As an example, it does not provide the lead time and rework commits metrics we need.

Besides playing a central role in this research project, the tool is also used by researchers involved in the research activities of the Kaloom-TELUS ÉTS IRC in DevOps to study different aspects of MR and will be made available in open source to the GitLab community (www.gitlab.com).

3.2.1 Tool Implementation

The *MR Data Extraction* tool is developed in Python and produces a CSV file that contains the set of data and metrics described in Table 3.2 and Table 3.3. The tool is configured using a json text file that contains the required information about the Gitlab repository on which the tool is to be executed, the date range for MR extraction, the user credentials that allows accessing the GitLab repository, and other optional information, as, for example, sprint dates to get the accurate Sprint Day data in the results.

The tool was developed iteratively to fit our research requirements as we found and explored results with Kaloom managers and refined our understanding of MR quality.

Each row of the resulting csv file corresponds to a MR and the columns correspond to the set of metrics. There are 27 values per MR which falls under 3 types of elements: *Data*, *Calculated*, and *Derived*. *Data* variables are simple data, either directly returned by the API or directly computed by the API. *Calculated* variables are data points resulting from a simple calculation of other data points. *Derived* variables are custom variables that we defined that needs to undergo some level of analysis before being calculated. These derived variables can be straightforward data that we know how to calculate but require some preprocessing or they can be technically difficult data to obtain for which we try our best approach at getting a value.

Table 3.2 and Table 3.3 lists all the metrics returned by the MR Data Extraction tool for each MR, provides a description of those metrics and establishes the type of variable. The metrics directly used to answer the research questions of this thesis are presented in Table 3.2. The remaining metrics are presented in Table 3.3 and have been used to investigate different aspects of MR not reported in this thesis. Several of them have been used at the core of the M.Sc. research work of Mashari (2022), including the ones related to sprints (Sprint Day, End Sprint Day, and Sprint), and the creation (Creation Date) and closing/merging (End Date) dates of MR. Metrics with remarks in their type definition have an asterix and are explained below the tables. These remarks offer clarifications about how their type was assigned or how they are calculated.

Table 3.2 MR Data Extraction Tool Data and Metrics directly used in this thesis

Name	Description	Type
Project ID	Project identifier	Data
MR ID	MR identifier	Data
State	State of the MR: *opened, closed* or *merged*	Data

Table 3.3 MR Data Extraction Tool Data and Metrics directly used in this thesis (continued)

Name	Description	Type
MR Size	MR Size calculated in terms of the total number of LOC added or removed. MR Size = LOC added + LOC removed	Derived *
Creation Date	Date when a MR was created	Data
Lead Time	Total time between the creation of a MR and its merger or closure.	Calculated *
# Discussions	Number of human discussions in the discussion section	Derived *
Commits	Total number of commits included in a MR	Data
Rework Commits	NFumber of commits related to a MR made after its creation	Calculated *
# Initial Files	Number of files included in a MR at the time of its creation	Derived *
Initial Files	List of the names of the files included in a MR at the time of its creation	Derived *
# Final Files	Number of files included in a MR at the time of its merger (or closure)	Derived *
Final Files	List of the names of the files included in a MR at the time of its merger (or closure)	Derived *

Table 3.4 MR Data Extraction Tool Data and Metrics not used in this thesis

Name	Description	Type
End Date	Date when a MR was *merged* or *closed*	Data
Sprint Day	Sprint day number[1] of the creation of the MR	Calculated
End Sprint Day	Sprint day number of the merger/closure of the MR	Calculated
Sprint	Sprint identifier	Calculated
Comments after merge	Number of discussions that were made after the merger of a MR. This metric can be used as an indicator that rework was necessary after review of the MR.	Calculated *

[1] Kaloom software delivery process is based on three weeks sprints (i.e. 21 days). The *sprint day number* is an integer between 1 and 21 that corresponds to the day of a sprint when a MR is created, merged, or closed.

Table 3.5 MR Data Extraction Tool Data and Metrics not used in this thesis (continued)

Name	Description	Type
Time to First Comment	Time between the creation of a MR and the first discussion by human other than the MR Author	Derived
Mean Time Between Commits	Average time between commits related to a MR. It is calculated as *Lead Time*/*Commits*	Calculated
Mean Time Between Review	Mean time between two discussions	Derived
# Commiters	Number of commiters involved in a MR	Data
Commiters	List of the names of the commiters involved in a MR	Data
Assigned Reviewer	Name of the reviewer who was assigned to the MR in GitLab	Data
True Reviewer	Name of the commenter who had the most interactions with MR Author in the discussions	Derived
# Commenters	Number of commenters involved in a MR	Data
Commenters	List of the names of the commenters involved in a MR	Data

The *Rework Commits* is the sum of all commits made after the MR is created. These commits are assumed to originate from changes required following a code review. This metric represents rework in our study. It is also use to calculate *Rework Size* groups.

The *Lead Time* of an MR is a critical metric to our research. It is defined as the total time taken by a MR. The *Lead Time* is usually fixed for *closed* and *merged* MR unless they are reopened in the future. MR that are in the *opened* state have an ongoing *Lead Time* until they are closed or merged.

Initial and *Final files* were expected to be simple data points, but the GitLab API is limited in its capacity to track filename changes. Getting the list of filenames that changed across MR and comparing them can not be done using a single API call with GitLab. The data is available but required processing to get the filenames. We also needed to take some decisions about how

we interpret the concept of a file in a merge request. The GitLab API doesn't let you access which files changed directly from the Merge Request API. To solve this issue and compute theses metrics, we had to detect when filenames were renamed and track the filename changes. We accomplished this through a separate Commit API call to verify each commit in the MR which allowed us to maintain an accurate representation of the quantity of files in the MR.

The # *Discussions* is a derived metric that is a best effort to count the number of human-made comments on a MR. This is a difficult variable to obtain because different automated tools also produce comments in the MR discussion feed in GitLab. Some of these automated comments can be detected via the use of a boolean flag called *system* in the Gitlab API, but not all tools use this method to flag an automated message. For example, Kaloom uses some proprietary security vulnerabilities detection automation that is run by a functional account which is not considered as Gitlab native automation during the static code analysis phase of the MR process. The comments resulting from this tool are currently not labelled as *system* automated comments, thus escaping the elimination step. Furthermore, not every reviewer writes a text comment in the discussion feed when they want to express a concern. For example, GitLab offers an alternative way to highlight source code similar to how text editors can highlight syntax errors. These code highlights are an integral part of the review process but can't be counted as a discussion by our tool. Our tool can only count textual comments left in the discussion field. At Kaloom, these comments are usually representative of an issue that can't be trivialised to a code highlighting without further feedback. Again, how comments are used can vary between Kaloom groups and individuals, and procedures are not the same between groups. Despite the inaccuracies of # *Discussions*, it still provides information that allows identifying code review concerns that were important enough to warrant the use of a comment and are useful to our study.

MR Size was expected to be a simple data point but turned out to be a derived variable that requires processing. Surprisingly, GitLab doesn't offer a clear MR Size data point in the API, even though their website application displays a similar variable called "changes". To obtain an accurate measure of the size of an MR we use the LOC that were modified by the MR. This

includes added and removed LOC. We define our version of *MR Size* by adding those two values together. The process to obtain this information represents a technical challenge that is also further explained in the following 3.2.2 Technical Challenge section.

3.2.2 Technical Challenge

The main technical challenges faced during the development of the GitLab MR Extraction tool were related to the limitations of the GitLab API. Our tool is limited by what the GitLab API can provide. Some metrics were impossible to obtain directly and had to be calculated with a degree of approximation. The *MR Size* discussed in the previous section is an example of such metric which was particularly difficult to obtain. Although GitLab provides a "number of changes" metric on their platform, this data is oddly absent from their API. For this reason, we had to build our own version of the *MR Size* using the available data. We used the sum of added and removed LOC to represent the *MR size* as this information was accessible via the commit chunk header information in the Commit API of GitLab.

Another technical challenge is the execution time of the tool. An API call takes approximatively 1-3 seconds to complete, making the tool slow for large amounts of MR. To mitigate this issue, we cache the API results with common data that can be reused for multiple MRs in memory, reducing the number of API calls needed.

A more detailed description of the technical challenges can be found in Appendix 1 (Details on Technical Challenges of the MR Data Extraction Tool) which covers the MR Size calculation as well as the parallelisation of the tool.

3.2.3 Validation

We validated the MR Extraction Tool at each iteration of its development with Kaloom to make sure the tool could answer their needs in a real-life business scenario of quality

improvement. This validation was synchronized with the incremental evolution of the tool over weekly meetings where Kaloom and our researchers were able to report on the usefulness of the tool and request changes and improvements to the tool to fit changing requirements. This evolutive process allowed us to validate and improve the tool iteratively as we discovered more metrics that we required for our respective usage. This iterative validation had three major impacts on the development: the removal of the Google Spreadsheet feature, the addition of a configuration file and a conversion to local time zone.

Google spreadsheet removal

The initial usability requirements were based around human readability. The tool generated results inside a Google spreadsheet. This spreadsheet was bound to a Google account and access to the sheet was configured inside the tool. Although this feature was a helpful first step, we realised that forcing a particular technology for viewing data was a superfluous limitation and that the large quantity of MR data in the final product would be better suited for a file type which could be used in other automated tools. We replaced the google spreadsheet output with a csv file which was easier to work with for subsequent analysis in R and python scripts.

Configuration file

The introduction of a configuration file was also a significant step in the evolution of the tool because it enabled Kaloom to easily use the tool on their code base and generate results without giving the researchers global access to sensitive information. The evolution of the configuration file is closely linked to evolution of the different features that were added to the tool to make this security requirement possible. This configuration file is also a critical feature that enables the tool to continue to be developed in an open-source environment and facilitate its use on any repository.

Local time zone conversion

Additionally, in the context of a related project, we discovered that all the timestamps were returned in the UTC time zone. This caused ambiguity issues when the time and day of the week was important for analysis. We therefore decided to globally convert all dates and times returned by the tool to the local time zone of Eastern Standard Time used in Montreal where Kaloom is located to facilitate time-based analysis.

3.2.4 Contextual and technical limitations

The contextual limitation has to do with the way of working of the different groups at Kaloom. The four groups at Kaloom don't use the same merging process and this merging process isn't always the same within a group. Some merging options have an impact on the availability of data in the GitLab API. For our tool to work, we need to know the commit SHA of the MR. Commits are uniquely identified by their SHA number, which is a common and simple way to refer to any commit. This unique SHA identifier is also required in API calls to obtain data. The location of this commit SHA varies depending on how the MR was merged and its status. Depending on if a squash commit was used, or if a branch was deleted after a merge, the MR may not have an accessible commit SHA by the API. The MR Extraction Tool uses the *merged_sha* field in the GitLab API to find the MR, with 2 fallback fields if the SHA can't be found there. If we can't trace the MR with these fields, then the tool can't extract all the data from it and that MR is ignored.

Although a MR can be represented as a single entity, it cannot always be associated to a single commit. MR often contains multiple commits, each with their own SHA identifiers. The approval of a MR also leads to more commits onto the branch where it's being merged. This plurality of commits requires an understanding of the merging process to select which commit SHA is relevant to our analysis.

To answer this, the Gitlab API offers three interesting commit SHA as part of the Merge Request API. The first available information in the API is the Merge Request SHA. This represents the last commit on the branch that is about to be merged. Although it is the commit that identifies when the Merge Request was made, it is not the best choice for our analysis. This is because it only contains the files that were changed right before the MR was made. If multiple commits exist on the branch that is being merged, then using this SHA would prevent us from seeing all the filenames from earlier commits in the branch being merged. Since we can't guarantee that branches will only contain one commit being merged, we need a better commit choice to represent the MR.

A relevant option with the merging process is the option to "squash" commits. Some platforms like GitHub and GitLab allows for the creation of a *Squash Commit* upon approving a MR. The purpose of a *Squash Commit* is to condense the commit history into a single commit that represents all the changes being merged. This is an excellent candidate to represent the MR, as it contains exactly what we are looking for. The issue is that we are not guaranteed that our industrial partners uses this *Squash Commit* option for every MR in every group. This option can be disabled by default and have different degrees of enforcement policy. Therefore, it is a valid candidate if it exists, but is not a guaranteed source of information.

Finally, the GitLab API offers us the Merge Commit SHA. This is the SHA of the commit on the resulting merge branch. To our knowledge, this commit seems to be equivalent to a squash commit but on the target branch. A threat to validity would be to forget to verify that MR with no squash commit only produce one commit on the resulting branch. For example, if we found that there were multiple commits on the target branch, then an iterative approach would be required to get all the filenames of each commit pertaining to a MR. This complexity becomes a problem, because commits from within a same MR are subject to code review and revisions, and that's the relationship we're already trying to measure. We would then have to decide on a cut-off point where we represent the MR as a single set of files and measure the evolution of these files in future commits not included in the MR itself. To avoid this complexity, the

current implementation uses the Merge Commit SHA as it is the closest representation of a single squash commit that we are guaranteed to have that represents the whole MR.

3.2.5 Other tools

As part of the research project, auxiliary tools were also developed to investigate different aspects of the review phase. These other tools include the *MR File Recurrency* tool and the *Merging* tool. The *MR File Recurrency* tool gathered data pertaining only to file names and tracked their evolution across a series of MR while the *Merging* tool formatted all the results from the MR Data Extraction Tool and the MR file recurrency tool together to enable combined analysis . Overall, the data collection covered all MR made by the four Kaloom groups in the years 2020 and 2021.

CHAPTER 4

EMPIRICAL RESULTS

In this chapter, we will present and discuss the results of our empirical study on understanding the rework for Kaloom's MR during the Review Phase. The goal of our empirical study is to first explore and understand the phenomena of rework at the Kaloom's context. We define rework in the context of this thesis as the commits that are made between the submission of a merge request and its acceptation. To answer our research questions, we leverage the collection of metrics obtained by the MR Extraction tool described in the previous chapter.

Our empirical study can be summarised on the following research questions:

- **RQ1: What is the distribution of rework commits across MR at Kaloom?**

Although the percentage (14.3%) of merge requests with a medium or large rework is small, such a percentage still accounts for a large number (1,279) of merge requests. These merge requests with rework are continuously increasing over time and exists on all of our studied sprints. Our results prove the prevalent of rework in our studied merge requests. **Our results suggest that rework exists in the evolution of the Kaloom's projects.**

- **RQ2: What is the relation between rework commits and the lead time?**

The median lead time of merge requests with medium or large rework are 17 times higher than the lead time of the merge requests with no or small amount of rework. The median lead time of merge requests with large rework accounts for a median of 4.65 days, which can be in an industrial setting a whole working week. While merge requests with medium or large amount of rework tend to have a large MR size, the size of the MRs is not the main factor that leads to rework as there is a moderate spearman correlation between the MR size and the amount of rework it receives. **Our results suggests that merge requests with rework are time-consuming and needs to be optimized.**

- **RQ3. How does the variation in the number of files associated with a MR affect the number of rework commits?**

Most of the merge requests with a medium or large amount of rework have a delta file that is not null. For instance, the median delta for the large amount of rework is five files, while that median is zero for merge requests with small rework. Most of the delta files are about missed files rather than files that did not have to change. **Our results suggest the need for tools that assist developers in identifying their dependent files to reduce the amount of rework.**

Take-home message: Rework is predominant and time-consuming in the context of Kaloom, and most of the rework is due to missing dependencies. Therefore, we urge future work to develop solutions that reduces the number of rework with a first suggestion on build an automation tool that recommends files to change.

In the remaining of this section, we present the results of our research questions with a motivation and the approach of each RQ.

4.1 What is the distribution of rework commits across MR at Kaloom?

4.1.1 Motivation

To understand the rework in the Kaloom's context, we first explore how prevalent are rework commits in the four different groups of Kaloom. Our research question will shed light whether there is a need to further investigate how to optimize the rework and what data limitations future work needs to take into consideration when reducing the rework at the merge requests reviews.

4.1.2 Approach

To understand rework commits at Kaloom, we calculated the statistical distribution of rework commits across the four Kaloom groups. The distribution was then grouped into rework size

groups: no rework for merge requests with no commits between the creation and the approval of the merge requests, small for merge requests with 1 to 2 rework commits, medium for 3 to 7 rework commits, large for merge requests with more than 7 rework commits. To better interpret our findings, we randomly selected merge requests that we manually studied with Kaloom's managers to better understand what occurred in the review process of these merge requests.

4.1.3 Results

Finding 1: We observe that the no rework and small rework categories are the majority across all groups at Kaloom, representing a cumulative total of 85.5% of all the MR, as shown in Figure 5.1. The figure shows the number of merge requests per each of our four categories of rework and the four groups of Kaloom. The no rework category forms 47.5% of the merge requests, meaning that almost half of all Kaloom MRs are merged without requiring any further rework. That said, **we also observe that the medium and large rework category represents 10.1% and 4.2% of all merge request, which accounts for 903 and 376 merge requests, respectively.** We consider such number of merge requests as large. In fact, since our study considers merge requests that are created and merged between January 2020 and October 2021 with an average of 58 merge requests with medium or large rework per month.

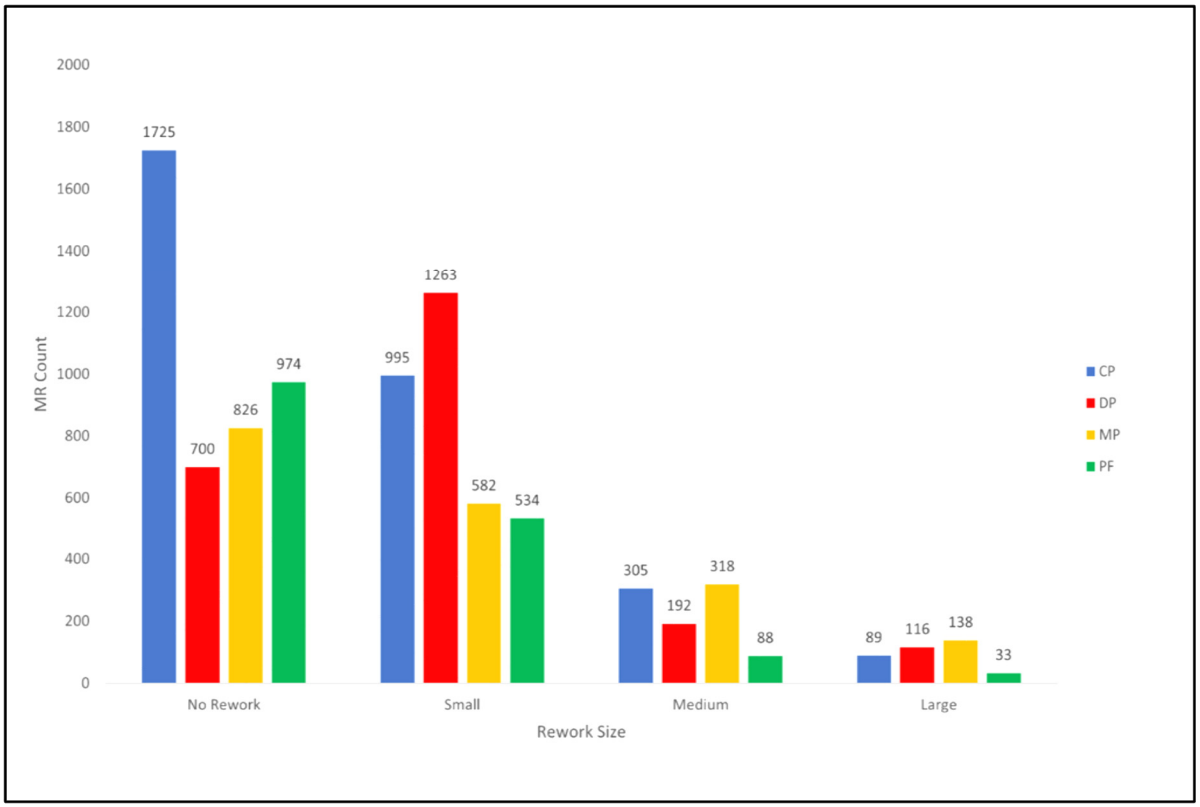

Figure 4.1 Distribution of MR Count per Rework Size across the Kaloom Groups

Furthermore, all groups seem to have more merge requests that are merged with no further rework except for the DP group for which the majority of its merge requests requires a one or tow extra commits to be merged. Our observation suggests that the review process is conducted differently in that Kaloom group. After discussion with Kaloom managers, we learned that the DP group has a longer build time in the CI/CD pipeline and that they perform extensive code review of the code in the P4 language, which has a lack of supporting quality assurance tools.

Interestingly, we observe that technical details about the GitLab version can explain some of the large reworks. For instance, on older versions of GitLab, code reviews could not always be grouped under a single revision. Each block of code might have its own small code review, ending up with multiple comments, each of which is addressed with a different commit. Early 2021, GitLab developed a feature where code review comments could be grouped into a single code revision, lowering the number of required commits to reply to each revision.

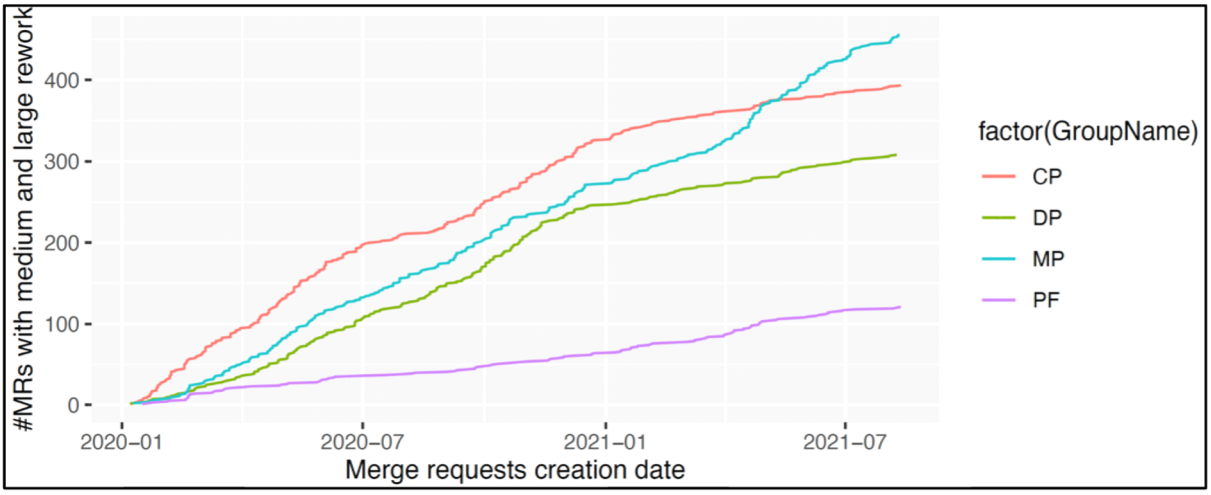

Figure 4.2 Evolution of the merge requests with a medium and large rework

That said, even after 2021 the merge requests from the medium and large reworks groups keep increasing, as shown in Figure 5.2. The same figure shows that for the four groups there is a continuous evolution of the number of merge requests with more than 3 rework commits, especially for the MP group.

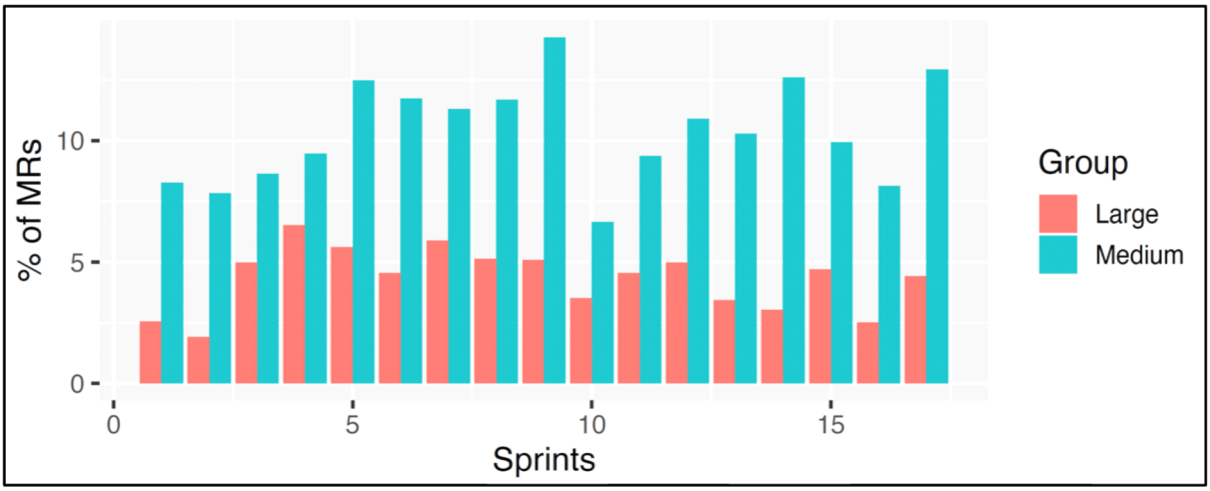

Figure 4.3 Evolution of the merge requests with a medium and large rework

Finding 2: We also observe that medium and large rework exist in all the sprints, as shown in Figure 4.3. Although the percentage of MRs with medium and large rework fluctuate over

time, we do not observe that such a problem is decreasing. For instance, the percentage of MRs with large rework went from 1.9% of the MRs in the 2nd sprint to 4.4% of the MRs in the last sprint. It reaches 6.5% of the MRs in the 4th sprint. As stated previously, although these percentages are small, they account for a large number of MRs. In the following research question, we will investigate whether MRs with rework requires more lead time than MRs with no rework to better quantify the impact of the rework.

Take-home message: The number of merge requests with multiple rework commits exists and it is still continuously increasing. While the version of Gitlab might explain some of the large reworks, it is not the only reason for the rework. Our results suggest the need for approaches to reduce such amount of rework with the goal of having a smooth integration process.

4.2 RQ2. What is the relation between rework commits and the lead time?

4.2.1 Motivation

The goal of this research question is to quantify the impact of rework on the lead time of merge requests. While it is expected that large merge requests with more rework commits will have a larger lead time, we wish to quantify such a difference. That will motivate whether there is an urgent need to reduce the rework or whether rework does not have a significant negative impact.

4.2.2 Approach

To identify the impact of the amount of rework on the lead time, we statistically compare the distribution of the lead time (time between the acceptance of a merge request and its creation) between the four groups of merges requests (i.e., these with no rework, low number of rework, medium number of rework, and large number of rework) that are used in RQ1. To explain our results, we investigate whether there are any differences in terms of the time required to give

a review as well as whether the size of the merge requests is different between the four categories of merge requests.

4.2.3 Results

Finding 1: The median lead time of merge requests with medium and large number of rework is 17 times higher than the median lead time for merge requests with no rework or a low number of rework, as shown in Figure 5.4. Such a median is 25.6, 4.8, 22.2, and 46.6 for the CP, DP, MP, and PF groups, respectively. For instance, the median lead time for merge requests with no, low, medium, and large merge requests is 1.03, 19.66, 42.02, and 111.8 hours, respectively. Note that 111.8 hours is a very long review process as it is equivalent to 4.65 days which can be a whole working week. Finally, the lead time of the no rework vs low, low vs medium, medium vs large reworks groups are all statistically significantly different (Wilcoxon test; p-value < 2.2e-16).

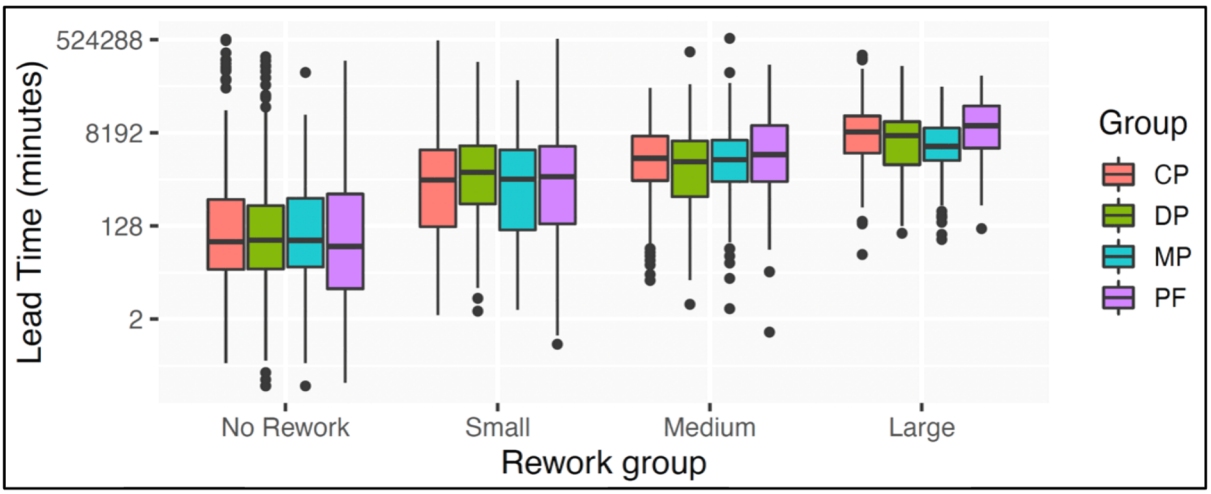

Figure 4.4 Boxplot of Lead Time per Rework Size across the Kaloom workgroups

Finding 2: MR with no reworks have a smaller lead time on average, but they still reach the same high extreme values as other rework size groups. This can be attributed to a concern for global optimisation of the Lead Time across the whole DevOps software delivery process, delaying the merge approval to save time in later testing phases. We also observe, via

a manual analysis with Kaloom's managers, cases with a no rework with a high lead time due to dependency between related changes. Developers have to first merge other dependent merge requests before the one with no rework and high lead time. One such example is the case of a big migration that occurred at Kaloom across the CP and DP group when they switched over to a replacing technology for one of their products. The MR was opened early and was only closed after the migration was completed. This MR had no rework because it only served to track changes across the migration branch and wasn't really part of the normal review phase. MR which requires longer automated testing like for the DP group are also usually delayed until the automated tests have passed to avoid running long hardware tests again after the merge if we know that they will fail. The nature of the products at Kaloom means that not all MR can simply be automatically merged after the CI/CD pipeline automated tests pass. Some tests in the Product Integration Testing and Product Feature Testing phase requires the use of specialised hardware and particular care is brought to some MR to ensure that they do not move on to the next phase if they don't pass the automated tests. This saves time on a global scale, but results in longer lead times in the review phase. This is especially the case in the DP group, which often displays different trends from other Kaloom workgroups in the results. Thus, **our results suggest that focusing on the relation between the rework and lead time might need to take into consideration the human efforts on reviewing a code change as some merge even with no rework can take a long review time**.

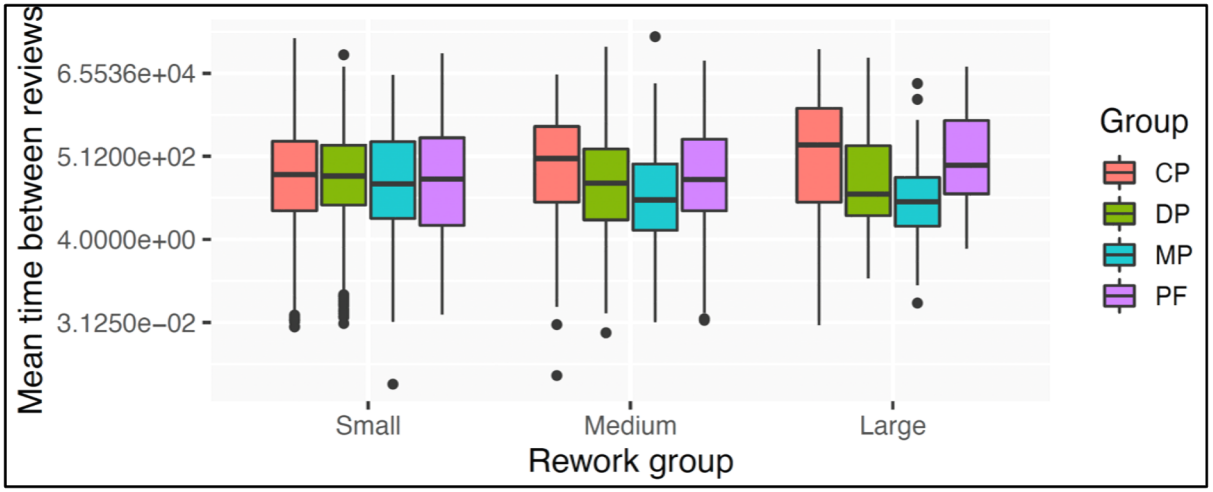

Figure 4.5 Boxplot of the average time between reviews

Finding 3: Developers receive reviews at a similar pace between merge requests with small, medium, and large rework, as shown in Figure 4.5. For instance, we do not observe any statistically significant differences (Wilcoxon test; Alpha = 0.01) between the time between reviews of the three rework categories for all the projects except two cases. These two cases are the diffrences between the small vs medium and small vs large for the CP group. One of the two differences is negligeable according to the effect size (Cohens'd = -0.03), whereas the second difference is medium (Cohens'd = -0.54). Our results suggest that medium and large rework does not mean that reviewers infrequently review changes or condensate the reviewers within a short period of time. Each review might require the same amounts of efforts to be reviewed, while we suggest the optimization of the rework so reviewers spend time on more merge requests rather than commenting the same merge request again and again.

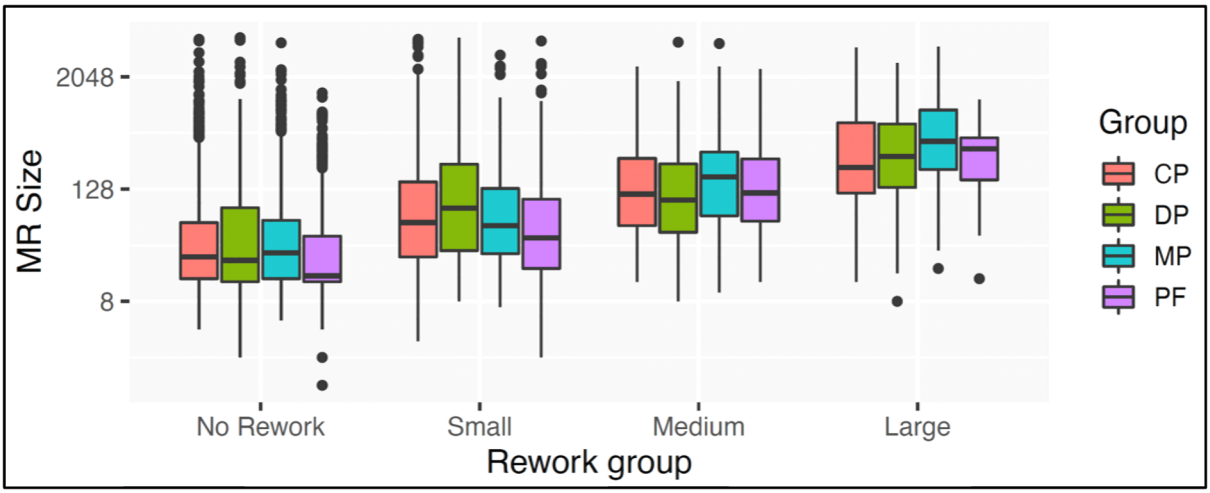

Figure 4.6 Size of the merge requests according to their rework size

Finding 4: The median size of merge requests with a large or medium rework is 5.4 times larger than merge requests with no or small rework, as shown in Figure 4.6. For instance, the median MR Size of the MRs from the no rework, small, medium, and large groups of rework is 20, 60, 133, and 328 respectively. All the differences between the MR size of the four categories of rework are statistically significantly different (Wilcoxon test; alpha = 0.01) except for the MR size of the small vs medium rework group for the DP group. Our results shed light on the importance of the merge request size on the amount of rework. However, all

we conclude is that merge requests with large rework tend to have more code change than the other merge requests. That said, there is a moderate spearman correlation (Spearman correlation = 0.47 as shown in Figure 4.7) between the size of a merge request and the amount of its rework suggesting the impact of other factors on the amount of rework.

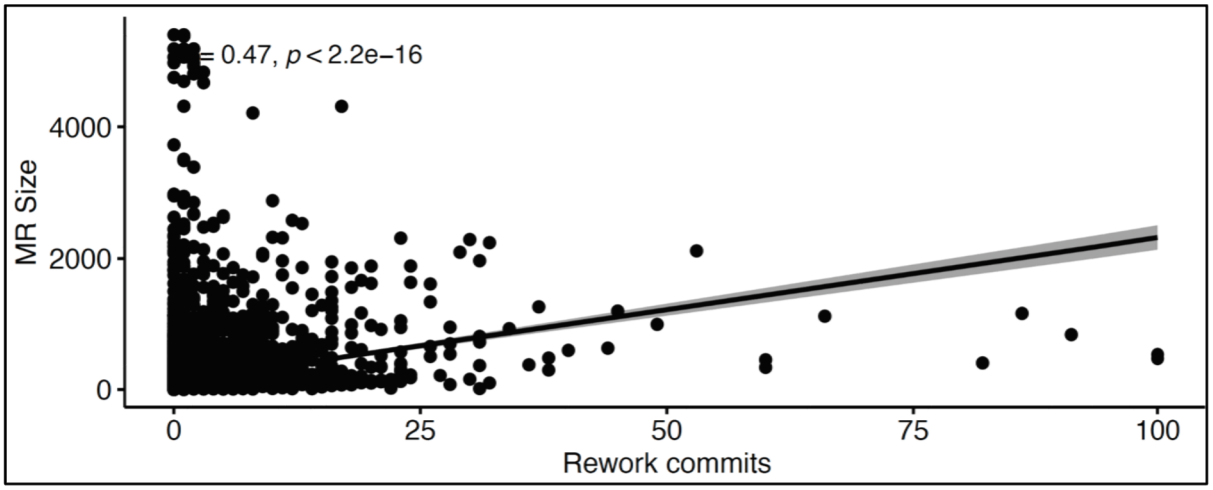

Figure 4.7 Spearman correlation between the MR size and the rework commits

Take-home message: While rework is time-consuming as it takes a significantly higher amount of time to be reviewed and merged, while time is not the only factor to consider when studying rework as there are merge requests with a large lead time and no rework due to merge requests dependencies. In fact, reviewers provide their feedback at the same pace for either low, medium, or large rework categories of merge requests. Our results suggest the need to reduce the rework so reviewers can spend more time on reviewing multiple merge requests rather than the same merge request.

4.3 RQ3. How does the variation in the number of files associated with a MR affect the number of rework commits?

4.3.1 Motivation

The goal of this research question is to investigate how the changed files are related to the rework. For instance, we wish to identify whether developers miss dependencies between files so they change a subset of the expected files and miss other relevant files for a merge request. We aim at investigating the opposite as well; whether developers change extra files that they did not have to change. Our results will indicate whether there is a need for tools that suggest which files do developers have to change to reduce the rework or the required tools to reduce the rework needs to target the source code lines to change instead.

4.3.2 Approach

To investigate the files that people changed during the code review of merge requests, we measured the delta file, as discussed in a previous section. The delta file represents the differences between the submitted files via a merge request and the final files after the code review. We consider a positive delta as cases in which developers forget to change certain files, while negative delta when they change some files that they did not have to change. In this research question, we first quantify the amount of merge requests with a non-null delta, the amount of files that exists in such a delta, and whether these files were missed or did not have to change.

4.3.3 Results

Finding 1: 32% of all the studied merge requests have a non-null delta of files suggesting the need for developing a solution that suggest to developers which files to change. 17% of the merge requests that exhibit a small rework have a non-null delta file. Such a percentage is higher for the other rework group, as 64% of the medium rework group has either a missed

file or file that did not have to change. Such a percentage is even 87.5% for the large rework group. Overall, the problem of rework is about inaccurately changed files.

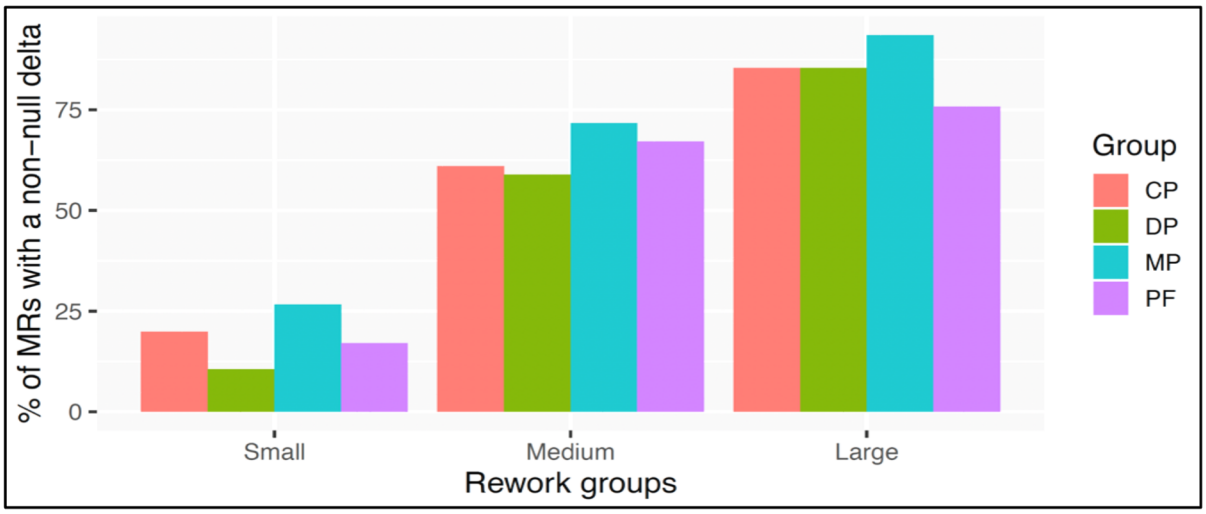

Figure 4.8 The percentage of MRs that exhibits each of the rework sizes with a non-null delta

Finding 2: **Our results indicate the need for tools that optimize the rework at the source code level for small rework group and at the file level for medium and large rework groups.** As shown in Figure 4.9, merge requests with a small number of rework commits have, as expected, a median of zero delta files. So, these merge requests would benefit from tools that optimize the rework at the source code level. Whereas, the medium rework group has a median delta of one file (either missed or did not have to change). Such a median number of files is five for the large rework group. After manual investigation of a dozen of MR to find a pattern or general rule to identify MR that are at risk of having a non-null delta file, we found a few scenarios that lead to creating a difference in the file count. For example, yang data model files were sometimes automatically added by mistake because they were autogenerated. The same type of files was also found to be required but forgotten in some MR, in the case of file renames. Another example includes new features that touches multiple areas of a component, where additional files needed to be changed were discovered during review. Major refactors where code moves around also lead to the review detecting more files needing to be changed. These examples shows that files can be added or removed as part of a rework for

reasons that can be part of the normal review processor not. An optimisation tool would have to be able to determine such scenarios to make good recommendations.

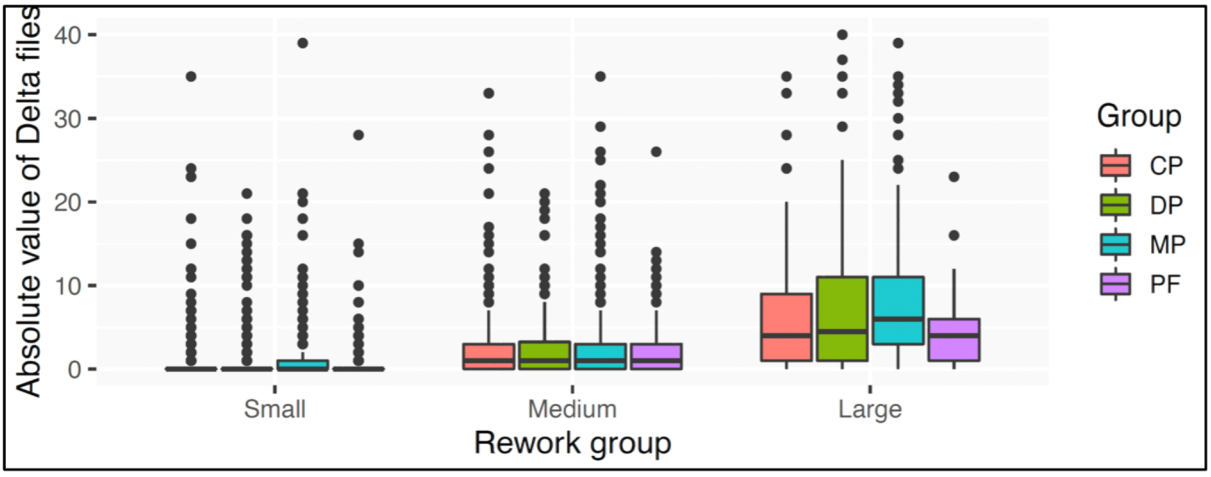

Figure 4.9 The percentage of MRs that exhibits each of the rework sizes with a non-null delta

Note that we use the absolute value of the delta files to get an idea about the files that people either did not have or forget to change. Not considering the absolute value can accidently end up with a median delta value of zero.

Finding 3: Developers mostly forget to change files rather than change files did not have to, as shown in Figure 4.10. In fact, 85.1% of the merge requests with a large rework have missing files compared to only 2.3% have files that did not have to change. Similarly, 15.9% and 59.6% of the files were missed for the merge requests that belong to the small and medium rework groups, compared to only 1.1% and 5.2% for the files that did not have to change.

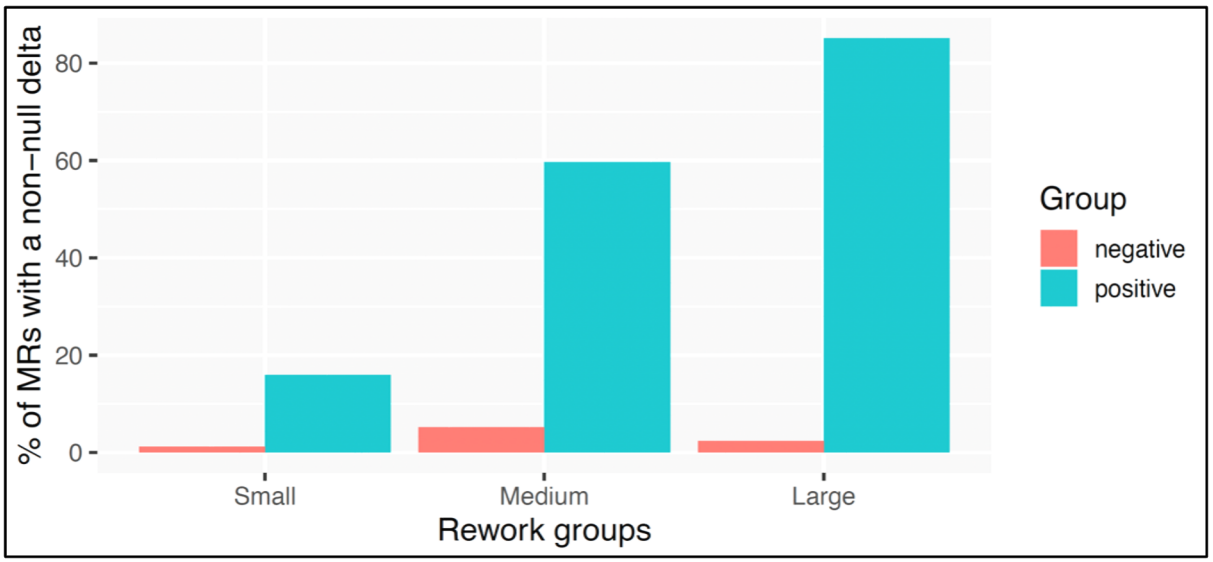

Figure 4.10 Positive (files that are missed in a merge request) vs negative (files that did not have to change in a merge request) delta files for the three rework groups

Take-home message: The rework is about missing dependent files, which can be a median of 5 files when there are a large number of rework commits. Our results suggest the need for tools that identifies missing dependencies before opening merge requests to minimize the number of reworks.

CHAPTER 5

DISCUSSIONS AND CONCLUSION

5.1 Summary

The goal of this study was to improve the Review phase and processes while using MR data available to us. To accomplish this, we studied the impact of the commits made after the code review inside a MR, that we called rework commit. We explored the relationship between the rework commits and selected metrics, to explain how they affected the review process. To perform our data collection as well as help other researchers involved in the Kaloom/Telus DevOps research chair in their research, we designed and developed our own GitLab MR Data Extraction tool. The tool was iteratively improved and validated through feedback with our industrial partners. This technical contribution was then used to obtain our own results for analysis.

We observed that the no rework and small rework categories are the majority across all groups at Kaloom, representing a cumulative total of 85.5% of all the MR. The no rework category forms 47.5%. We also observed that the medium and large rework category accounts for 903 and 376 merge requests, respectively. This translates to an average of 58 merge requests with medium or large rework per month. The review grouping feature of Gitlab might explain some of the large reworks, it doesn't account for quality related issues in the review process. We also observed that those MR exists in all the sprints. We do not observe that such a problem is decreasing. This analysis prompted to our questioning of the impact of MR with more rework on the lead time and MR size.

While rework is time-consuming as it takes a significantly higher amount of time (17 times higher for MR with large amount of reworks) to be reviewed and merged, time is not the only factor to consider when studying rework as there are merge requests with a large lead time and no rework due to merge requests dependencies. In fact, reviewers provide their feedback at the

same pace for all rework categories of merge requests. Therefore, reducing rework would allow reviewers to focus on more merge requests rather than the same merge quest.

32% of all the studied merge requests have a non-null delta of files, bringing out the need for a solution to suggest to developers which files to change. Our results indicated the need for tools that optimize the rework at the source code level for small rework merge requests and at the file level for medium and large rework groups. 85.1% of the merge requests with a large rework have missing files (with a median of five files added during the MR process), suggesting that developers mostly forget to add files rather than having to remove changes that weren't needed. Verifying that all required files are indeed modified when opening a MR could help reduce the amount of rework.

5.2 Research Limitations

This research was conducted within the Kaloom-Telus Industrial Research Chair at École de Technologie Supérieure (ÉTS). In this context, we operated with constraints to the scope of software delivery cycle at Kaloom, our access to traceability data and our commitment to respect ethical boundaries related to data from individuals.

The scope of the research was focused on the Review Phase and more specifically at the GitLab code review process in place at Kaloom. This local optimization approach shies away from the DevOps methodology which encourages global improvements over local ones. This limitation was self-imposed both to scope the research to a manageable level and because our industrial partners had an immediate interest in improving this phase of their CI/CD pipeline.

We were also limited to using only the MR data available from GitLab to conduct our research. Although we had full access to every part of the software, we did not have a direct access to the confidential source code, so our data collection tool was run under supervision by Kaloom managers. We also did not have access to any other source of data, including the Trouble Report data from JIRA. Our goal was to find out everything we can using only the MR data.

Because code review is a fundamentally human activity, we had to be careful to respect ethical boundaries with the data we collected. We opted out of using any personal or anonymous data from individuals in our research. We understand that experience and familiarity of a reviewer with the code is an important factor in code reviews.

5.3 Threat to Validity

A self-evaluation of threat to validity is critical to any scientific research. We mitigated the risks to validity to the best of our abilities, but some challenges were hard to control for. Notably the accuracy of some metrics was hard to establish, mostly because of the variability in the existence of the data used to calculate some metrics like the specific merge commit and the MR size. The different groups at Kaloom have different ways of working and this forced us to make some technical assertions which are explained in detail in Appendix 1. For example, the number of rework commits is calculated from commits made after the initial MR commit creation. Because the merge commit ID is dependent on how the branch was merged, we had to expand the commits of some MR to include all commits made on the MR branch when the precise commit ID for the MR creation didn't exist anymore. This may have had an impact on the calculated amount of rework commits for MR where branching policy differed from one branch per MR.

Outlier identification was challenging because of the non-normal distribution of the data. A simple statistical definition of outliers based on the standard deviation might have been an insufficient discriminator for outlier detection. We mitigated this problem by focusing on the majority of the dataset rather than extremities. In the extended context of the industrial research chair, another master thesis (Mashari 2022) took an in-depth look at the outliers to better explain them.

We cannot generalize our results to other software or code review tools. This research was specific to the industrial context in which it was performed and showed the importance of context when establishing relationship between metrics. This differs from open-source project

research where all projects are treated the same and context is sacrificed for generalization. However, the results of this research may be transferable to another context similar to our industrial partner.

5.4 Future Work

If anything, our research explored and mapped the limits of what can be studied using only the MR data. We went as far as we could with the limited information we had. Although it would appear trivial to say that traceability is critical to conduct code quality analysis on MR and comparing them with bugs later found in the code, this research showed us the importance of traceability at every step of the way. As a researcher, this study proved to us the value of traceability between the code review, MR and documented issues. To explore the code quality impact of code review in depth beyond the MR data, traceability with bugs, issues and related MR should be added in future work. The process of doing research without such information was a humbling experience that should remind us that research isn't always about finding major results but about exploring what we can accomplish with what is available.

In DevOps, we are always looking for ways to improve the flow of work and minimize delays. Looking at the Review Phase is useful, but leads to some unanswered questions: Is it more efficient to perform longer code reviews and avoid unnecessary rework later, or is it more efficient to do quicker code reviews to progress faster and fix the smaller issues as they arise? The DevOps approach would be to fix issues earlier as possible, but modern code review practices sometimes favor a quick code merging approach. It would be interesting to compare both approach to see what kind of code quality issues can be found using these two paradigms and the benefits they have on the organization in terms of experience and knowledge synchronization.

5.5 Final word

Reaffirming what we know to be true in a different context might not bring the most exciting results, but it is still a valid result. To be in line with scientific research, ulterior motives such

as article publication and industrial financing must never become a directing force in a study. We hope that the scientific community in software engineering continues to mature to understand the value of constructivist research over the classical positivist paradigm. Science and industry has just as much to gain with transferable results, which can drive our research field out of blind open-source data mining into applicable context-relevant research

APPENDIX I

DETAILS ON TECHNICAL CHALLENGES OF THE MR DATA EXTRACTION TOOL

To understand how we retrieved the *MR Size* information, we must first explain how a commit chunk header works. Each commit in Git can have one or multiple chunks of modified code. Each of these chunks of modified code is preceded by a header which contains useful information to calculate the MR Size. The general format of a commit chunk header contains the first line where code was added and how many LOC were added and repeats the same information for deleted LOC. The following figure exemplifies the use of the commit chunk header information to calculate the size of the MR.

```
diff --git a/src/calculator.py b/src/calculator.py
index 3822668..371c301 100644
--- a/src/calculator.py
+++ b/src/calculator.py
@@ -167,2 +167,3 @@ class Calculator:
167 167         #                       headers=self.headers).json()
    168 +
168 169         result = self.cache_mr_commit
@@ -417,3 +418,4 @@ class Calculator:
417 418
418     -       MR = [str(self.getProjectID()), MRID, str(self.getCreationDate()),str(self.getSprintDay()), str(self
    419 +       MR = [str(self.getProjectID()), MRID, str(self.getCreationDate()), str(self.getSprintDay()),
    420 +             str(self.getState()), str(self.getEndDate()),
419 421             str(self.getSprintEndDay()),
@@ -428,3 +430,3 @@ class Calculator:
428 430             str(self.getTrueReviewer()), str(self.getAssignedReviewer()), str(self.getMRSize(MRID)),
429     -           str(self.getSprint()) ]
    431 +           str(self.getSprint()), str(self.getReworkCommits())]
430 432
@@ -620 +622,12 @@ class Calculator:
620 622         return sprintDay
    623 +
    624 +   def getReworkCommits(self):
    625 +       """returns the amount of commits within a MR that are created after the MR creation date"""
    626 +       commits = self.cache_mr_commit
    627 +       MRCreationDate = self.general_details["created_at"]
    628 +       reworkCommits = 0
    629 +       for commit in commits:
    630 +           commitDate = commit["created_at"]
    631 +           if commitDate > MRCreationDate:
    632 +               reworkCommits = reworkCommits + 1
    633 +       return reworkCommits
```

Figure-A I-1 Commit chunk header example

In figure A, we see part of a commit made in Git. The grey lines are the commit chunk headers, the green lines are the LOC that were added, the red lines are the LOC that were deleted, and the white lines are lines that weren't modified but provide context around the part that was changed. The chunk headers are delimited by enclosure between the '@@' symbols. The deletion information is in the segment starting with the '-' symbol, followed by the first line that was removed, and the total number of lines that were removed. The addition information is similarly represented with the '+' symbol, the first line that was added, and the number of added LOC. If we look at the first chunk header in this figure, we see: " @@-167,2 +167,3 @@. This means 2 LOC were deleted and 3 LOC were added. In this specific case, both deletion and additions are counted starting at line 167. However, if we take a closer look at the code represented by that chunk header, we notice that there is only a single actual green LOC that was added. Line 168 was added, and no lines of code were removed. This seems to contradict the chunk header, which says 3 LOC were added and 2 were removed. This mismatch is due to the chunk headers always including white lines of code, or unmodified contextual LOC, in the calculation. Since we have line 167 and 169 as contextual LOC in this chunk, they are both counted as well. This explains how we get the numbers seen in the chunk header. The problem is that these white contextual lines of code are not constant and varies depending on how we choose to visualise the commit. Some tools allows to tweak the number of contextual lines to get a better picture of the changes. Because this number can vary, we don't get the real number of modified LOC. We would need to specify a value of zero lines of context to get the exact MR Size. We don't have control over how GitLab displays commits when we use the Commit API, meaning we can't control the number of contextual lines in the result of the API call.

This clarifies why we say the *MR Size* is a Derived Variable. We use our best way of calculating the size, following GitLab's specifications on how they return chunk header information. For our study, this is accurate enough to measure the MR Size, if we assume that the error caused by this method is systematic across all MR. We still get an accurate enough representation of the scale of the MR.

Alternatively, the MR Size could have been calculated by counting every single LOC that begins with a ‘+’ or ‘-‘symbol. This would considerably slow down the algorithm to count the MR Size. Accessing the chunk header is done directly in O(1) in the API Result. Counting N LOC to get the maximum accuracy would raise the algorithm to O(N) where N is the LOC. This would be significantly slower as we would have to count every single line of code ever committed in the GitLab Repo across every commit.

As a possible improvement, an asynchronous architecture design capable of running multiple API call at the same time is a planned feature that could improve the total execution time of the tool. The following UML activity diagrams are used to show how the order of the API calls and their results would change between the current synchronous design and a future asynchronous architecture.

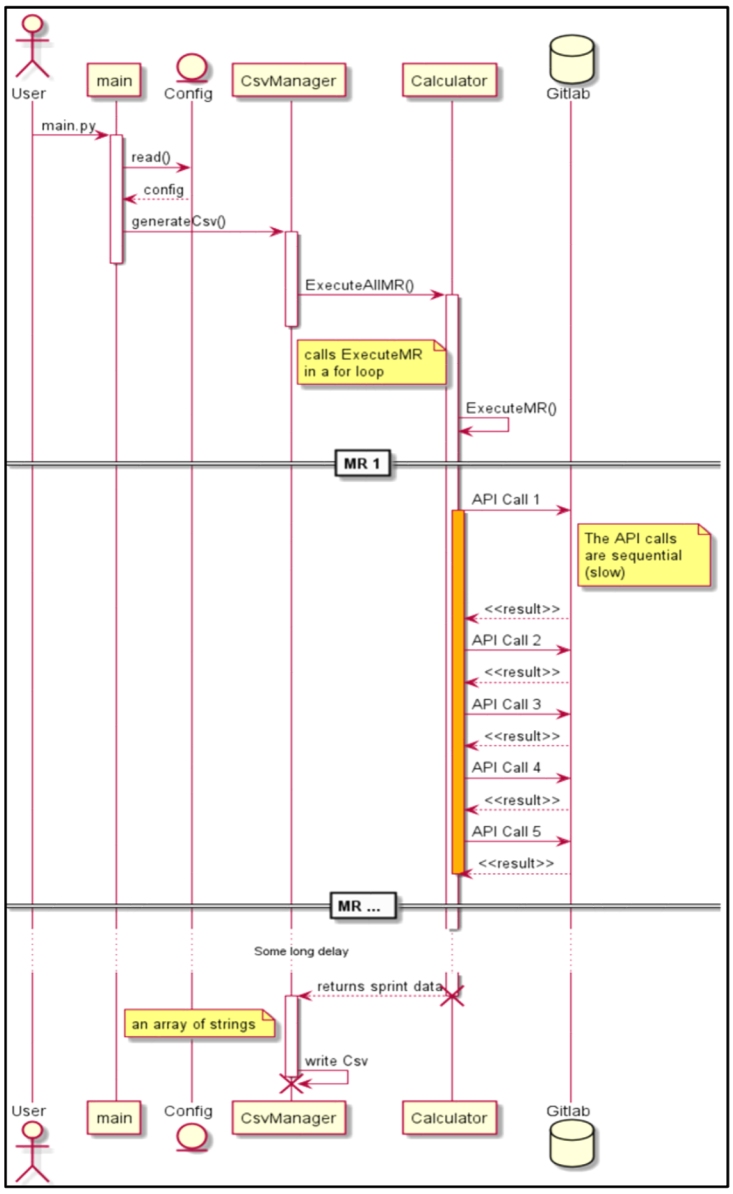

Figure-A I-2 Synchronous architecture activity diagram

The Synchronous UML Activity diagrams shows that we have a human User Actor, the main class Actor which uses a Config Object at the start of the application before the real work starts in the CsvManager class Actor. The CsvManager uses a loop in the ExecuteAllMR() method to call the ExecuteMR() method for each MR. We then see that the API call to the Gitlab Actor are made in sequence for each MR, and we wait for the API Call to return it's result before making the next call. This waiting step is the key element which slows down the execution of the tool.

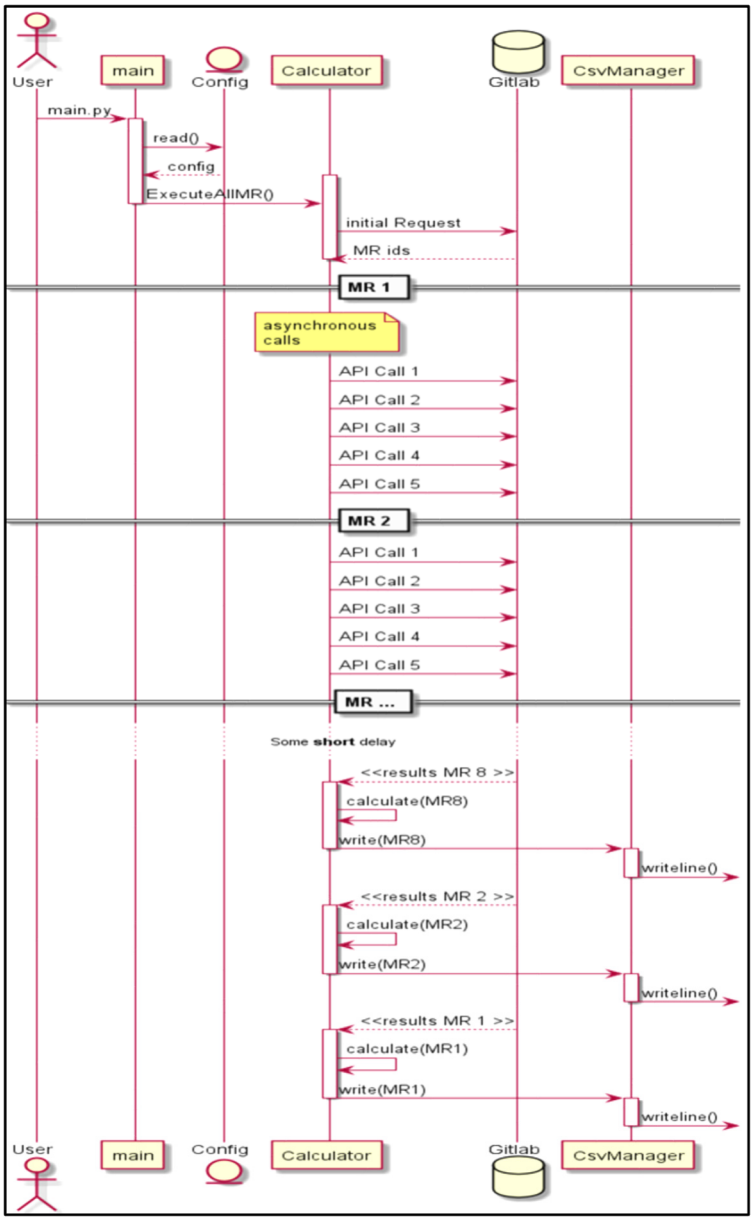

Figure-A I-3 Asynchronous architecture activity diagram

In this asynchronous version of planned architecture, the technical limitation is removed. We see the same actors as in the figure C with a similar start of the diagram. The difference is that the loop doesn't wait for the API results. Each API Call is still made sequentially, but the results can arrive in any order across multiple MR. in this diagram, we see results from API Calls for MR 8 arrive before MR 2, then the data is handled in the calculate method and written to the CsvManager before moving on to the received results of MR2. This allows us to make

all the API calls and handle results as they come in, skipping the wait between calls. The trade-off to this approach is more memory use and management, as we need to format unordered data before writing the csv file.

LIST OF REFERENCES

Amit, I., & Feitelson, D. G. (2021). Corrective commit probability: A measure of the effort invested in bug fixing. *Software Quality Journal*, *29*(4), 817–861. https://doi.org/10.1007/s11219-021-09564-z

Chatley, R., & Jones, L. (2018). Diggit: Automated code review via software repository mining. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 567–571. DOI: https://doi.org/10.1109/SANER.2018.8330261

Chouchen, M., Ouni, A., Mkaouer, M. W., Kula, R. G., & Inoue, K. (2020). Recommending peer reviewers in modern code review: A multi-objective search-based approach. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 307–308. https://doi.org/10.1145/3377929.3390057

Davila, N., Nunes I. (2021). A Systematic Literature Review and Taxonomy of Modern Code Review. *Journal of Systems and Software.* DOI: https://doi.org/10.1016/j.jss.2021.110951

Ebert, F., Castor, F., Novielli, N., & Serebrenik, A. (2019). Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 49–60. https://doi.org/10.1109/SANER.2019.8668024

Forsgren, N., Humble, J., Kim, G. (2017). *Forecasting The Value of DevOps Transformations*. DORA Devops research and assessment.

Ghadhab, L., Jenhani, I., Mkaouer, M. W., & Ben Messaoud, M. (2021). Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, *135*, 106566. https://doi.org/10.1016/j.infsof.2021.106566

GitLab. (2022). Retrieved from www.gitlab.com.

Hirao, T., McIntosh, S., Ihara, A., & Matsumoto, K. (2022). Code Reviews with Divergent Review Scores: An Empirical Study of the OpenStack and Qt Communities. *IEEE Transactions on Software Engineering*, *48*(1), 69–81. https://doi.org/10.1109/TSE.2020.2977907

Hong, Y., Tantithamthavorn, C. K., & Thongtanunam, P. P. (2022). Where Should I Look at? Recommending Lines that Reviewers Should Pay Attention To. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 1034–1045. https://doi.org/10.1109/SANER53432.2022.00121

Kaloom. (2022). Retrieved from https://www.kaloom.com/

Kim, G., Humble, J., Debois, P., Willis, J. (2021). *The Devops Handbook. How to create world-class agility, reliability & security in technology organizations*. United States of America. IT revolution Press LLC.

Mashari, S. (2022). *Extraction and analysis of behavior practices based on GitLab MR information*. Mémoire de maîtrise électronique, Montréal, École de technologie supérieure.

McIntosh, S., Kame, Y., Adams, B., Hassan, A.E. (2015). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5). DOI: http://dx.doi.org/10.1007/s10664-015-9381-

Thongtanunam, P. (2016). Studying Reviewer Selection and Involvement in Modern Code Review Processes. Doctoral dissertation, Graduate School of Information Science, Nara Institute of Science and Technology

Thongtanunam, P., Kula, R. G., Cruz, A. E. C., Yoshida, N., & Iida, H. (2014). Improving code review effectiveness through reviewer recommendations. *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, 119–122. https://doi.org/10.1145/2593702.2593705

Thongtanunam, P., McIntosh, S., Hassan, A. E., & Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. *Proceedings of the 38th International Conference on Software Engineering*, 1039–1050. https://doi.org/10.1145/2884781.2884852

Thongtanunam, P., Tantithamthavorn, C., Kula, R. G., Yoshida, N., Iida, H., & Matsumoto, K. (2015). Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 141–150. https://doi.org/10.1109/SANER.2015.7081824

Yan, M., Fu, Y., Zhang, X., Yang, D., Xu, L., & Kymer, J. D. (2016). Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software*, *113*, 296–308. https://doi.org/10.1016/j.jss.2015.12.019