Vrije Universiteit Amsterdam

Bachelor Thesis

Towards Analyzing Developer Efficiency in Enterprise-Driven Open Source Projects through Commit and Issue Tracking Data

Author: Ayman Errahmouni 2774028

1st supervisor:	Sieuwert van Otterloo	
daily supervisor:	Lodewijk Bergmans	(SIG)
2nd reader:	Joost Schalken	

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 30, 2025

Contents

Abstract

Developer efficiency is an important property of the development process. However, existing quality evaluation systems, like SIGRID, do not use process-related information to do their analysis. Because there is no client (closed source) data available to create a representative data set, this thesis attempts to create a representative data set from open source data instead, which can be used to create an initial benchmark for developer efficiency.

A sample set of 40 open source enterprise-driven software projects was created from a larger set of 17000 software projects. All of them are on GitHub. Using the GitHub API, issue tracking and commit data was extracted from the GitHub Repositories used by each software projects, and was used to extract metrics which could be used to create benchmarks relating to efficient use of developer time. The GitHub Repository "https://github.com/aer205/vu-thesis-0" contains the which can be used to perform the required steps.

Analysis of the data shows that both issue cycle time and time before start-of-work on an issue are heavily right-skewed. Their correlation with each other is very strong for larger values of the time before start-of-work on an issue, with a Pearson correlation of 0.62 and -0.95 for regular Issues and Pull Requests respectively, when the time before start-of-work on an issue is longer than 2 weeks. But for smaller values, their Pearson correlation is much weaker, with 0.04 and -0.09 for regular Issues and Pull Requests respectively, when the time before start-of-work on an issue is shorter than 2 weeks.

GitHub Issues tend to be short-lived, with 73% of all Issues closed within 2 weeks and 31% within the first day. Most of these short Issues are Pull Requests, which account for 95% of the Issues closed within the first day. Regular Issues are longer-lived, with 44% taking being closed within the first 2 weeks.

All repositories in the sample follow this pattern: Issues with a long issue cycle time tend to have a proportionally long time before the start-of-work. Although regular Issues are only 15% of the Issues in the sample, a third of Issues lasting longer than a month are regular Issues. In general, Issues are mostly short-lived, with most of these short-lived Issues being Pull Requests. Pull Requests themselves also tend to be short-lived.

1 Introduction

1.1 Background and Context

Software is a very important part of modern life. It is used in many domains, ranging from social media platforms to critical infrastructure. Because software must be maintained and improved, it is important that the development process makes effective use of developmental resources. Rapid resolution of issues ensures that software bugs are fixed quickly [Kho+12], thus making efficient use of developers' time is important. The measurement of developer efficiency through development artifacts (e.g. issue cycle time, code quality metrics) is a research field that is still being researched [TB22].

SIG, or Software Improvement Group, is a company that specializes in providing clients with advice on how to get the best out of their software projects, by helping them improve the security, maintainability, greenness, and overall architecture of their software projects. They provide software to their clients which can be used to score the client's software project, as well as to advise their clients on how to proceed. In this way, SIG helps

bridge the gap between software developers and project managers, helping them to gain understanding of each other's work.

SIG's software is bundled in SIGRID, which uses the source code of their client's software project to evaluate its qualities. It uses a benchmark which compares the source code against many other examples of source code from other software projects to produce a scoring. This scoring can be used to find the strengths and weaknesses of the software project and thus ways to get the most out of the software project.

1.2 Problem Statement

The current implementation of SIGRID looks mostly at the source code of the software projects when evaluating them. Although this is enough information to judge the current quality of the source code, it cannot be used to judge the process used to create the source code. The future quality of the source code depends on how the software project will be developed and managed, something that SIGRID does not touch on. By benchmarking the development process itself, clients can receive more applicable advice for their software projects.

The development process is to evaluate the versioning system of the software project. The versioning system contains lots of useful data such as commit rates, code churn, issue cycle times, among others. These data can be used to create a benchmark, which is how the versioning systems will be evaluated. However, this is where the issues with this approach start: Lack of data.

One way to solve this issue is by collecting the versioning system information from SIG clients. This would yield a data set that is most representative of the clients of SIG, as it comes from them directly, and SIG has had enough clients for the data set to be big enough to work with. However, creating such a data set would require SIG clients to provide their versioning system information to SIG. SIG clients have no incentive to do this: They would be giving away potentially sensitive information and risking it being leaked for no short-term benefit. However, if SIG had a benchmark to evaluate versioning system information, that would be enough incentive for SIG clients to provide versioning system information. This is a case of the chicken-and-egg problem: SIG clients will not provide their versioning system information without a way to benchmark it, but that information is required to create such a benchmark. Another source of data representative of SIG clients is needed.

An open source data set containing enterprise-driven open source projects was provided to us by SIG to use as a set of software projects representative of SIG clients [Spi+20]. The data set comes from a project whose goal was to address known generalizations on the data of 17264 open source software projects, mainly funded by enterprises. The projects are staffed mainly by insiders from the companies themselves [Spi+20]. For each project, the data set provides 29 features which describe the project and how it was obtained. Examples include the number of lines of code (The "lines" column), the number of files (The "files" column) and the GitHub repository URL (The "url" column). The reason why this data set is useful is that we can use it to limit our analysis to only enterprise-driven open source projects. The software projects are all GitHub repositories, which can be accessed via the "url" column.

Examples from the data set (showing only 5 columns):

url ("https/github.com/" omitted)	star_number	commit_count	pull_requests	author_count
Microsoft/vcpkg	5957	10557	3299	712
Netflix/iep	49	1074	371	14
makandra/consul	247	182	24	19
okta/samples-nodejs-express-4	57	143	34	13
Azure/azure-uamqp-c	38	1447	200	65

1.3 Research Questions

This study focuses on finding the relationships between the cycle time of the issue and metrics that could be linked to efficient developer use. The cycle time of an issue is defined as the time it takes for an issue to go from the start-of-work to completion, where work is defined as the actions which are directly related to successful completion of an issue. Thus, work on an issue begins when all tasks have been assigned, and code changes can begin to be made, and work on an issue ends when the changes are visible in the software project and the issue is closed. If meaningful relationships between them can be found, they can be included in a benchmark for developer efficiency within software projects.

Main RQ: How is issue cycle time with issue usage in enterprise-driven open-source projects?

- **SRQ1: How can issue cycle time be determined using GitHub Issue data?**
- **SRQ2: How is issue cycle time associated with the time it takes to initiate work?**

2 Related Work

2.1 Mining GitHub

GitHub offers huge and valuable data, and has more than 100 million users and 800 million open data files as stated at [Rom+23]. There is a tool called Pydriller to make mining software repositories easier which we will also use in our work. [SAB18]. Also, SIG, the company we are working with on our thesis and Tu Delft researchers made this maintainability model [Di +19] where we will be checking our findings with.

Tornhill and Borg [TB22] analyzed multiple industrial codebases and found only a weak correlation between lines of code added and average issue resolution time. However, CodeScene's Code Health tool correlated higher with the issue resolution times. They reported that "resolving issues in low quality code takes on average 124 more time in development" and "issue resolutions in low quality code involve higher uncertainty manifested as 9 times longer maximum cycle times".

Researchers at Leiden University [MV23] used a method called BERT "to perform the bug report validity classification task" and worked on "Which bug reports are valid and why?" which can be helpful in our research to classify the bugs.

2.2 Developer Engagement

There is a framework called SPACE that attempts to capture the developer productivity. They concluded "that productivity cannot be reduced to a single dimension (or metric!)." [For+21]. "This framework, called SPAC+E, captures the most important dimensions of developer productivity: satisfaction and well-being; performance; activity; communication and collaboration; and efficiency and flow."

In addition, a recent study in 2024 [Raz+25] finds a relation between Developer eXperience and developer productivity.

2.3 Maintainability

There are some maintanibility measuring models like Maintainability Index from 1997 [WOA97], SIG model [HKV07], and recently Delta Maintainability Index. In addition, there are some researches and comparisons between maintainability models such as [SCR20], and [Opp].

2.4 Code Review Practices and Software Quality

Alberto Bachelli at Zurich University has conducted various researches on this topic, in which he mostly collaborates with others. In 2013, at the 35th International Conference on Software Engineering (ICSE), they published a paper on "Expectations, Outcomes, and Challenges of Modern Code Review" [BB13]. Their study found that "while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems."

In 2020 [Kov+20] they found that "automated reviewer recommendation algorithms leaves out important aspects: perception of recommendations, influence of recommendations on human choices, and their effect on user experience" and "reviewer recommendations rarely provide additional value for the respondents" indirectly showing the importance of code review practices as it is not easy to replicate by an algorithm and they gave tips for a new algorithm.

[CG20] Chen and Goldin created new thrashing frequency metric and found strong evidence that it's a valid measure of code quality.

[DLR10] In Commit 2.0, they created "IDE enhancement to enrich commit comments using software visualization. Commit 2.0 generates visualizations of the performed changes at different granularity levels, and lets the user annotate them."

[Sad+18] at modern code review study with Google, they found that indeed "code review is an important aspect of the development workflow at Google. Developers in all roles see it as providing multiple benefits and a context where developers can teach each other about the codebase, maintain the integrity of their teams' codebases, and build, establish, and evolve norms that ensure readability and consistency of the codebase. Developers reported they were happy with the requirement to review code.", showing the importance of code reviews once again.

In 2018 again [Spa+18] they made a detailed analysis of the review practices and proposed recommendations for automated testing tools.

As a result, the existing study shows a positive correlation between robust code review practices and improved software quality.

3 Methods

The GitHub Repository "https://github.com/aer205/vu-thesis-0" contains files and docs to perform the all the steps described in this chapter.

3.1 Sampling

The data set containing the enterprise-driven open source projects [Spi+20] was last updated in April 2020, with data points from May 2019. At the time of writing (July 30, 2025), over 6 years have passed since the collection of the data inside the data set. In those 6 years, many of these software projects may have been closed and the information about them in the data set might be wrong. In addition to that, many enterprise-driven software projects are not representative of SIG clients. To create a set of software projects that are representative of SIG clients, we need to ensure that only actively developed software projects are selected. This sample will be used as the basis for our analysis: we extract our metrics from this sample.

- **Extract # of commits made in the last 90 days for each GitHub Repository**
  For each software project, we obtain the number of Git commits made in the last 90 days. The number of commits will be used as an indication of activity for the purposes of selecting software projects based on activity. This is because software projects which have no commits made in a certain time also have not added, removed, or changed any code in that period. This makes it a simple way to tell whether or not a software project is active or not. The reason a 90-day period was selected is that it is long enough for a low activity period to occur in the software project without excluding it from the analysis. The number of commits can be extracted via the GitHub API [Mic], by extracting the commits made in the last 90 days.

- **Remove Projects with 0 commits made in the last 90 days**
  After collecting the number of commits in the last 90 days for each software project, we remove those that do not have any commits made in the last 90 days. If the number of commits for a software project could not be obtained, it is also removed from the data set, as getting the commits from the GitHub Repository is required for computing metrics.

- **Keep Projects where the # of commits made in the last 90 days is in the 90th-percentile**
  Of the remaining software projects, a sample should be created that is representative of SIG clients. This is needed as many of the software projects, while still having activity recently, are not active enough to be considered representative. Most software projects are low in activity, which can be seen when the ‘last90‘ values are plotted out, like in Figure 1.

- **Select n Projects with the largest # of commits made in the last 90 days** The final sample is created by selecting the n software projects with the highest 'last90' value in the 90th percentile of 'last90' values. n was chosen to be 40, as the resulting sample consists of actively developed software projects that are representative of SIG clients, while still small enough to be handled with the present resources.

3.2 Issue Representations

From the set of selected software projects, issue tracking data can be extracted to calculate the metrics of interest. GitHub Issues are used as the issue tracking system on GitHub. GitHub Issues are used as an aid to development, being frequently used for bug reports, planning and discussing work and coordination of work. By analyzing these GitHub Issues, it might be possible to find patterns in development which can be used to evaluate the development process. The data to be extracted will relate to the timing of indicators of work being done or work being finished.

GitHub Issues are split into 2 categories: Pull Requests and regular Issues. Every GitHub Issue has an associated list of events, called the Issue Timeline, where they are stored in chronological order. Every GitHub Timeline Event has a type, the event identifier, and auxiliary information depends on the type. Most event types also include the time it occurred as auxiliary information. An Issue Timeline represents the history of a GitHub Issue, which can be used to estimate when work has begun or ended.

The Event Timeline is available for both Pull Requests and regular Issues, but there are some differences between them. Not all event types are available to either of them. Some event types, like "committed", are only available to Pull Requests, while others, like "transferred", are only available to regular Issues. Pull Requests also have a second timeline of events, namely the Git commits on the Git branch they are associated with. A Git commit is an indicator of actual work being done: the addition/removal of code or other tracked information. This information can be extracted from each GitHub Repository via the REST API provided by GitHub, using the PyGithub library [PyG].

3.3 Metrics

3.3.1 Issue Cycle Time

The issue cycle time of an issue as a metric describes the amount of time spent on a given issue. This metric is used in the research questions as a way to represent the time spent doing work on a given issue. This will be compared with other metrics to answer the research questions. The issue cycle time, as defined in the research question, can be defined as the formula:

$$t_{\text{cycle}} = t_{\text{end}} - t_{\text{start}}$$

Where t_{start} and t_{end} are defined as the date of the start and the end of work on an issue respectively. To do this, we require data that chronologically catalog the work done on an Issue from start to end. To this end, we use the Issue Timeline and, if the issue

is a Pull Request, the associated commits. These structures chronologically record many events that may occur in an issue, including work being turned in (Git Commit) and an Issue being resolved, closed, or discarded (Timeline Events). This data is extracted from the Issue or Git commit from the GitHub Repository.

3.3.2 Start-Of-Work

The start-of-work date of an Issue is defined by the first event associated with the Issue representing the start of work. Whether an event is considered to be the start of work, is dependent on the type of the issue. After looking through all issue event types, 3 are been considered to indicate the start of work on an issue: "committed", "connected", and "assigned". Git Commits are also considered start of work, and are denoted as "<commit>" [Mic].

The "<commit>" event type is not an event type defined by GitHub, but is still considered one for the purposes of analysis. The "<commit>" event type is triggered when any Git Commit is made to the HEAD of the branch of a Pull Request. Logically, if a Git Commit is made, then work has been done, as any code change is considered work by our definitions. Thus, the "<commit> "event type indicates the start of work on an issue.

Because a Git Commit is usually made after changes are made locally, the start of work date will be off. This is because a Git Commit can only be made when changes to a (local copy of) the Repository are made beforehand. The date of the first code change is not recorded, so the second closest date to the start of work is the date the Git Commit was made. In this study, the "author.date" attribute of the Git Commit is taken as the date the Git Commit was created.

Because a Pull Request can be created from a branch which has existed before the creation of the Pull Request, and Git Commits can be made to that branch, it is possible that start-of-work happens before the creation of the Pull Request. This means that, in theory, the issue cycle time can be longer than the total duration of the Issue on GitHub.

The "committed" event type, similarly to the "<commit>" event type, is triggered when a commit is made to the HEAD of the branch of a Pull Request, but unlike the <commit> event type, it must be after the creation of the Pull Request. This is because "committed" shows up in the GitHub Timeline, and requires the Pull Request to already be created. The "committed" event type, in effect, is simply the "<commit>" event type, but after the Pull Request is made. We can thereby differentiate between Git Commits made before and after the creation of the Pull Request. The "committed" event does not specify the date it was triggered, so the date is computed in the same method as <commit>: By getting the commit associated with the "committed" event and extracting the "author.date" attribute. The commit can be retrieved via it's hash, which is extracted from the URL of the event.

The "assigned" event type is triggered when an issue is assigned to a user. On GitHub, you can use assignment to assign work to a user. This user is then responsible for the Issue. This system is meant to clarify that the assignee is working on the Issue, so occurrence of this event represents an assignment of work and is thus considered a start of work event. The "assigned" event type specifies when it was triggered via the "created_at" attribute.

The "connected" event type is triggered when a Pull Request is linked to another issue. Linking a Pull Request to an Issue is used to signify that the linked Pull Request is meant to resolve the Issue. This is counted as starting work on the Issue, because it is an

assignment of work to the people working on the linked Pull Request. The "connected" event type specifies when it was triggered via the "created_at" attribute.

3.3.3 End-Of-Work

The t_{end} of an Issue is defined by the last event associated with the Issue representing the end of work. The "closed", "merged" and "deployed" event types are used as the end of work event types.

The "merged" event type is triggered by merging a Pull Request. This merge takes the associated commits from the Pull Request and adds them to another branch. Adding finished code to the software project and closing a Pull Request means that work on the Pull Request is finished, as the merge cannot be undone, and is for that reason counted as the end of work on the Pull Request. The merging of the Pull Request that triggers the event can be done via one of three merge methods: merge commits, rebase and merges, and squash and merges.

The merge commit method will create a merge commit to merge the Pull Request. This method preserves the date of the first commit. The rebase and merge method copies all the commits from the Pull Request to the target branch, which also preserves the date of the first commit. But the squash and merge methods destroy the commit history of the Pull Request and "squashes" them into a single commit.

Knowing this, all Pull Requests that are merged using squash and merge are not valid, as start-of-work may be off because the original commits have been destroyed. Thus, any Pull Request suspected to be squashed and merged is removed. This includes all Pull Request where there is only one commit, and that commit only has a single parent. This also includes Pull Requests that are rebased and merged with only one commit, which means that some valid Pull Requests may be lost. However, to be safe, the decision was made to remove a Pull Request in case of doubt.

the "deployed" event type is triggered by the deployment of a Pull Request. Deployments are used to automatically publish software updates. If a Pull Request gets deployed, it means that it is made available to users and thus complete. It is thus counted as the end of work on the Pull Request.

If multiple end-of-work events occur in the same Issue, the last one is chosen. This can happen when, for example, an Issue is reopened and then closed again. Another problem is that the "closed" event type is always the last end-of-work event type in the timeline. Because the "closed" event type is always present, it is not as informative as the other event types. For that reason, it is only selected when no other end-of-work event type precedes it.

Issues ending in an "convert_to_draft", "converted_to_discussion" or "marked_as_duplicate" event before they are "closed" are discarded. This is because they either convert an Issue into a topic to talk about or invalidate the work defined by the Issue.

"convert_to_draft" converts a Pull Request into a Draft, making it unable to be merged until it is converted back. "converted_to_discussion" converts an Issue into a discussion, making it unable to be assigned to user or linked to other issues. Discussion on GitHub are more like forums than a defined piece of work, and are thus ignored. "marked_as_duplicate" is triggered by marking an Issue as a duplicate of another Issue, and thus the duplicate Issue should be ignored.

3.3.4 Ignored Event Types

GitHub API REST version 2022-11-28, which is used in this study, defines 42 event types in total. However, most of these are not suitable as indicators for starting or ending of work on an issue:

- *"automatic_base_changed_failed", "automatic_base_changed_succeeded"*
  These event types are triggered automatically by GitHub, when the target branch of a Pull Request has been merged with another branch. GitHub will change the target branch to the branch the original target branch was merged into. These are not triggered by work on the Pull Request in which the event is triggered, but by work done on another Pull Request. These event types are, therefore, not considered to start of end work on an Issue.

- *"base_ref_changed"*
  This event type is triggered when changing the target branch of a Pull Request. This event type does not indicate the start or end of work, but rather where the work will end up, and is thus ignored.

- *"commented", "cross-referenced", "labeled", "unlabeled", "pinned", "unpinned", "renamed", "subscribed", "unsubscribed", "milestoned", "demilestoned", "user_blocked", "mentioned", "unmarked_as_duplicate"*
  These event types fall into the same category of being only indirectly related to work. Talking about, mentioning, and referencing an Issue can all happen without any work following or preceding it. To know whether any of these are directly related to work requires analysis of the contents of the messages themselves, which is not feasible due to resource limitations. Adding label, renaming, and milestone event types also fall into that category, where their content needs to be examined to come to any conclusions.

- *"disconnected", "unassigned"*
  The "disconnected" and "unassigned" event types are triggered when a previously linked Issue has been unlinked and a user assigned to work on an Issue has been unassigned, respectively. These events are related to work being done, as they are related to assignment of work to user and issues. However, they are always preceded by the "connected" and "assigned" event types, which are already start-of-work event types, described in Section 3.3.3.

- *"transferred"*
  The "transferred" event type is triggered when an Issue is moved to another repository.

- *"head_ref_restored", "head_ref_deleted", "head_ref_force_pushed"*
  These event types are triggered when the HEAD branch of a Pull Request is restored, deleted, or force pushed, as the names imply. "head_ref_force_pushed" and "head_ref_deleted" are always preceded by a merge or deletion, which means that they are always after work has ended or been discarded.

- *"locked"*
  This event type is trigger when locking an Issue. Because this could be done for any reason, with no indicator of why, this event type is ignored.

- *"reopened", "ready_for_review"*
  This event types both are both not considered start of end of work event types because, an Issue being reopened or ready for review never logically occur at the start of end of work. They are both events that happen during the work, and do not indicate that work has started or ended on an Issue.

3.3.5 Total Issue Duration

The Total Issue Duration is defined by the time passed between the start of the GitHub Issue and the end of work on the GitHub Issue. This would normally be the creation date of the GitHub Issue. However, the start date is defined as the minimum between the "created_at" attribute and the start-of-work date. This is done to correct scenarios in which a start-of-work event is triggered before the creation of the Issue, which causes the issue cycle time to be longer than the total issue duration, which is incorrect.

3.4 Data Set Summary

The final data set is an aggregate of all the aforementioned metrics. For every issue in each repository, we extract the following:

- The issue-number (GitHub way to uniquely identify an issue)
- The creation date ("created_at" attribute)
- The closure date ("closed_at" attribute)
- The start-of-work event type
- The start-of-work date
- The start-of-work ID
- The end-of-work event type
- The end-of-work date
- The end-of-work ID
- Whether the issue is a Pull Request

2 Example rows from the data set (IDs are omitted for brevity):

number	4472	6371
created_at	Fri Jun 07 2024	Sun Jun 08 2025
closed_at	Thu Feb 13 2025	Mon Jun 09 2025
start-of-work	Mon Jun 10 2024 ("assigned")	Sat Jun 07 2025 ("<commit>")
end-of-work	Thu Feb 13 2025 ("closed")	Mon Jun 09 2025 ("merged")
Pull Request?	No	Yes

4 Results

4.1 Sampling

The data used in the analysis were collected on 30 June 2025. After extracting the number of commits made in the last 90 days for each project, it became clear that most projects are no longer active. This phenomenon of most open-source GitHub Repositories being inactive has been observed in the past [Kal+16].

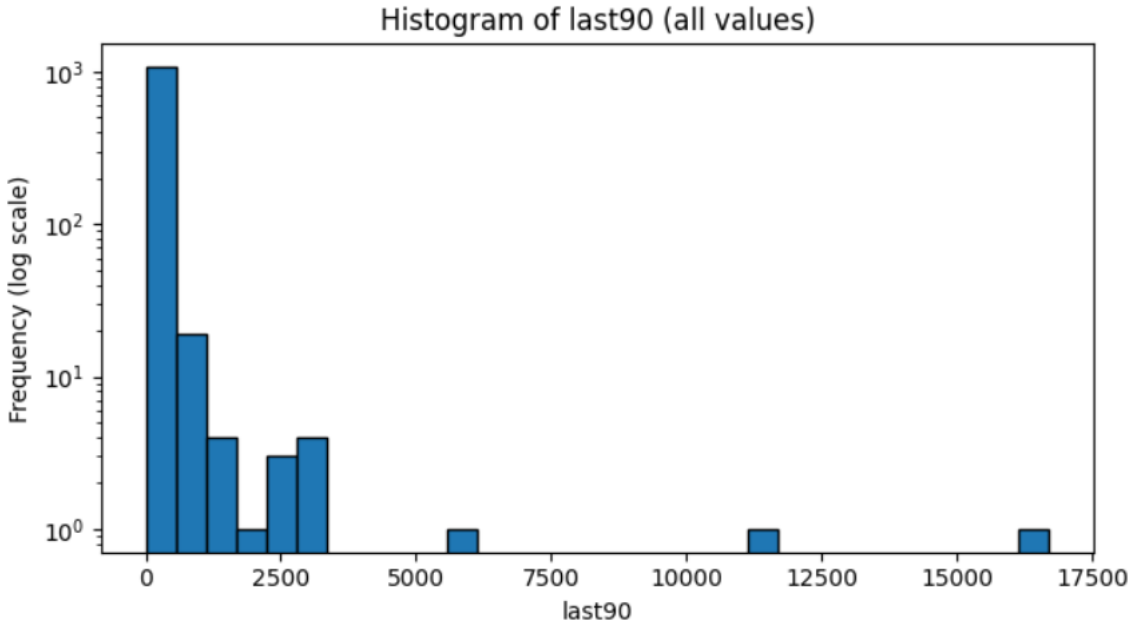

Figure 1: Most software projects in the data set are low activity

After removing inactive projects, only about 1200 software projects remain, as seen in Figure 2. This implies that almost 95% of the 17000 software projects in the data set are inactive. The data The software projects left are projects that are not fully inactive and can be filtered further. 1200 software projects are also more software projects than present resources can handle. The distribution of 'last90' values is heavily skewed due to outliers. The final two sampling steps ensure that these outliers, which are not representative of SIG clients, are removed to produce the final sample.

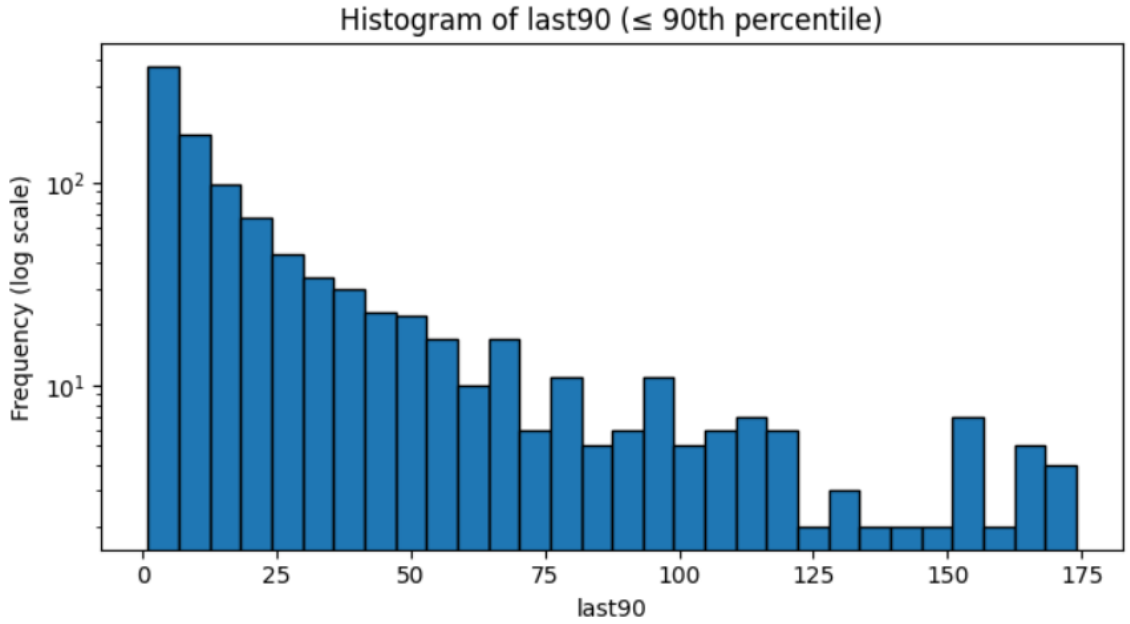

Figure 2: Values of ‘last90‘ not in the 90th-percentile. Notice that the right-skew is much less pronounced than before the sampling

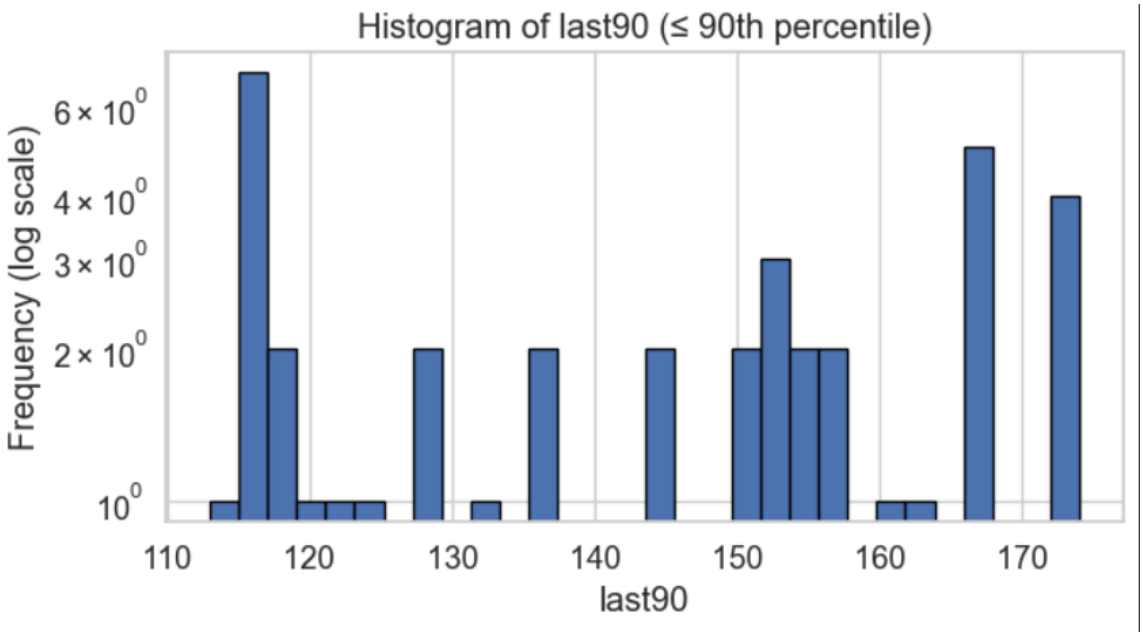

Figure 3: Values of ‘last90‘ in the final sample.

SRQ1: How can issue cycle time be determined using GitHub Issue data?

After extracting all relevant metrics from GitHub, a final data set was created for analysis. Since this study pertains to modern practices related to issue tracking, only issues closed

within the last year and created within the last one and a half years were included. This data set contains 15640 Issues across 40 GitHub Repositories, with 13311 Pull Requests and 2329 regular Issues. Of the 15640 Issues, 8650 Issues had no start-of-work. This happens when no start-of-work event or end-of-work event is triggered in the Issue, like when an Issue is worked on but never explicitly assigned, and thus no issue cycle time can be determined. This means that only 6990 Issues have a defined issue cycle time.

Of the remaining Issues, 85% were Pull Requests, which can be seen in Figure 4. Around a third of the Issues are finished within 1 to 2 days of creating them. The distribution of issue cycle times is heavily right-skewed because of this. This seems related to the fact that 99.82% of the Pull Requests in the data set are triggered by "<commit>" (98.33%) or "committed" (1.48%), and that most Pull Requests are closed within 1 to 2 days.

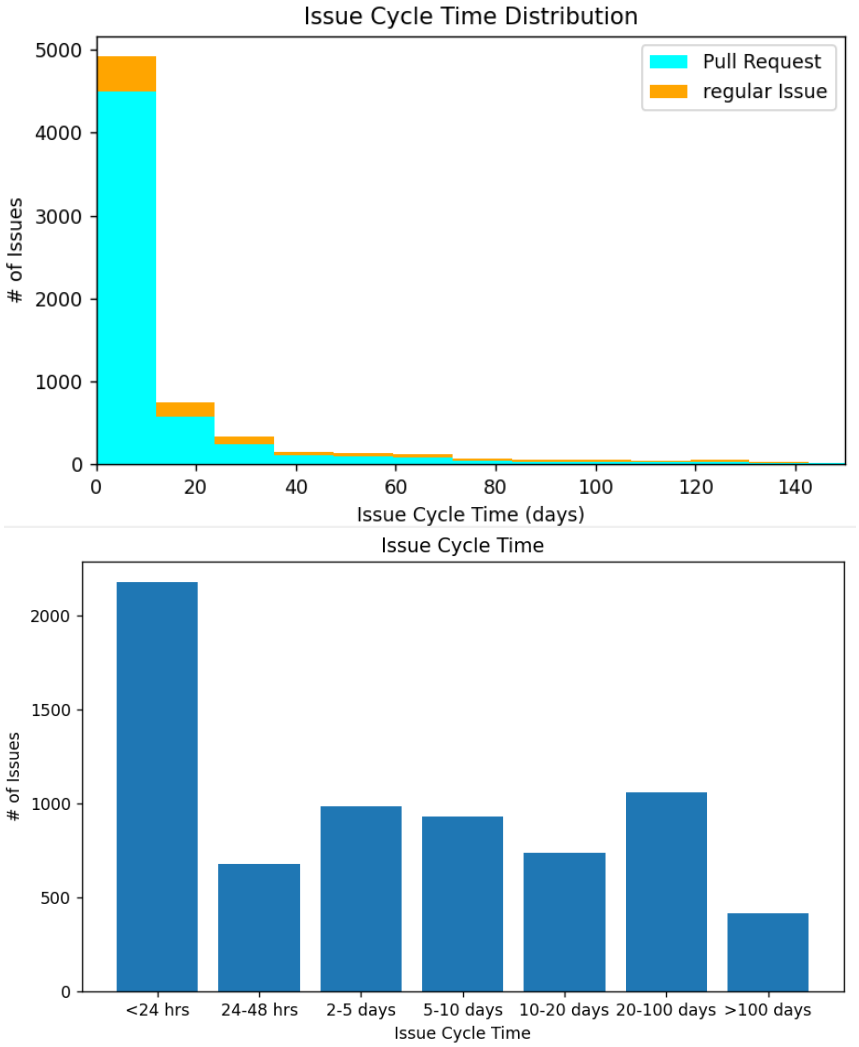

Figure 4: Distribution of Issue Cycle Times. Most Issues are resolved within 2 weeks, with ≈2500 Issues closed within the first 2 days.

Summary of Issue Cycle Time (Days):

	Median	Mean	Standard Deviation
Pull Requests	1	11.12	63.09
Regular Issues	17	50.63	71.89
Mixed	2	17.04	64.90

The frequencies for each start-of-work event type are: "<commit>" (34%), "committed" (1%), "connected" (0.3%), "assigned" (8%) and 55% is missing, as seen in 5. A missing start-of-work event means that the Issue did not trigger any of the start-of-work event types, and thus has no discernible start-of-work date.

Almost all Pull Requests have "<commit>" as their start-of-work event type, with 95% of Pull Requests having this event type as their start-of-work event type. Because most Issues are Pull Requests, this also means that the most common event type overall is also "<commit>". "<commit>" never occurs for regular Issues, as commits to them cannot be made.

At a distant second is "assigned". Similarly to the "<commit>" event type in Pull Requests, almost all regular Issues have "assigned" as their start-of-work event type, with 96% of regular Issues having this event type as their start-of-work event type.

The remaining Issues with a defined issue cycle time have the "connected" or "committed" event type as their start-of-work event type. "committed" rarely occurs for the same reason that "<commit>" is so common: most Pull Requests have their work started before they are created, and "committed" can only occur after creation of the Pull Request. "committed" never occurs for regular Issues, as commits to them cannot be made. "connected", which is triggered by linking to other Issues, only occurs 34 times in the entire data set.

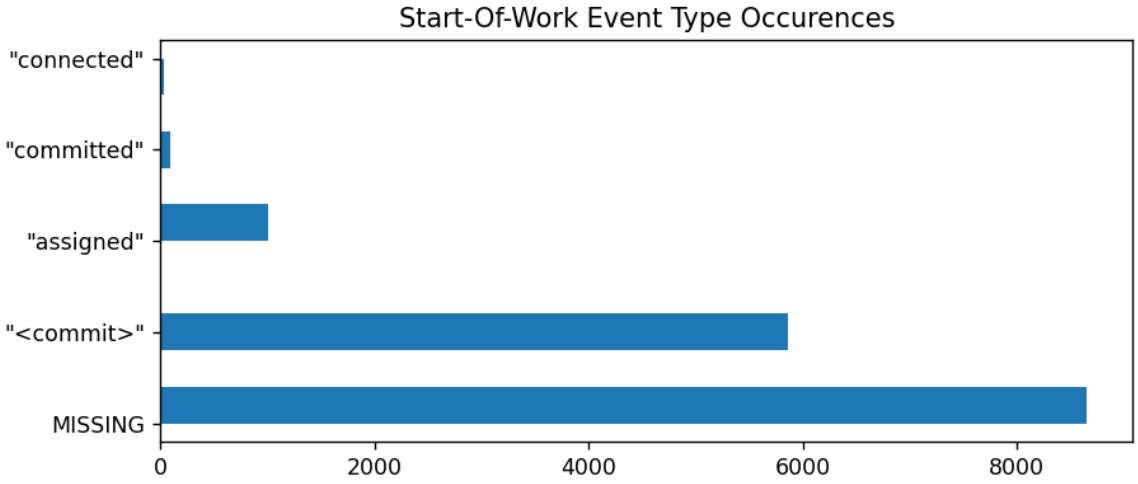

Figure 5: Start-Of-Work event type occurrences. For most Issues, the Start-Of-Work could not be determined.

The frequencies for each end-of-work event type are: "merged" (74%), "closed" (24%) and all other end-of-work event types occupying the remaining 2%, as seen in Figure 6. Unlike the start-of-work, the end-of-work is always defined for a given Issue. This is because all Issues in the data set are completed and are thus in the "closed" state. Being

in the "closed" state implies that the Issue was closed, which would have triggered the "closed" event type, meaning that there are no missing end-of-work events.

The "merged" event type is the most common end-of-work event type. This even type is only triggered by Pull Requests, with 87% of Pull Requests having this as their end-of-work event type. 99% of Pull Requests ending in a "merged" event start with a "<commit>" event. The "closed" event type acts as a catchall for Issues that do not have another more specific end-of-work event type preceding it. 95% of regular Issues end in "closed". The "deployed" event type also occurs only in Pull requests with a frequency of 1%. The remaining event types are event types that remove the Issue if it is present. 1% of the Issues analyzed are disqualified by ending in "marked_as_duplicate", "convert_to_draft" or "converted_to_discussion".

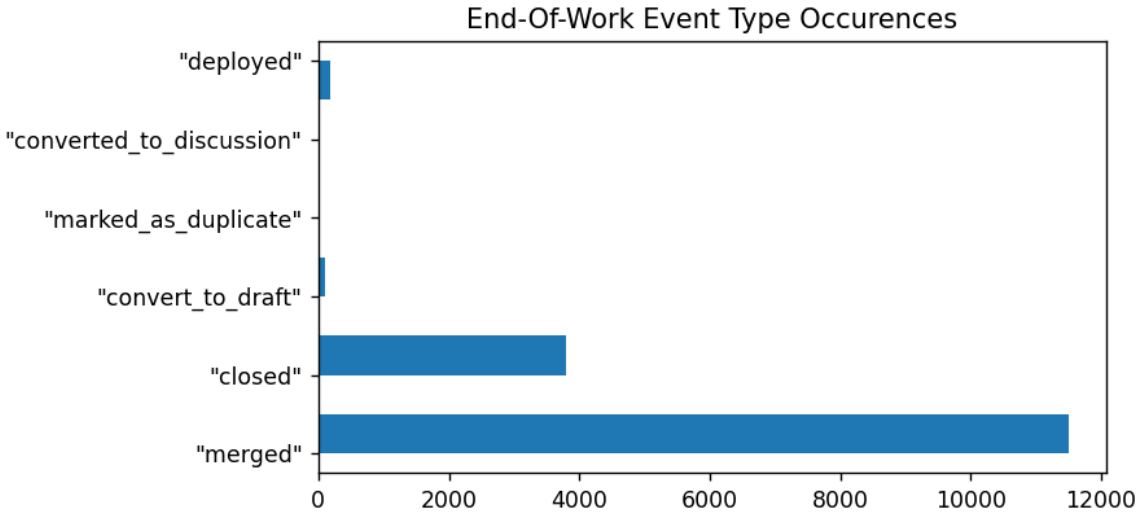

Figure 6: End-Of-Work even type occurrences. Unlike Start-Of-Work event types, all Issues have an End-Of-Work.

SRQ2: How is issue cycle time associated with the time it takes to initiate work?

The time it takes to initiate work is defined as the difference between the start-of-work date (start-of-work) and the creation date of the Issue. This means that for Pull Requests, this value is extremely likely negative, due to the fact that the most common start-of-work event is "<commit>" and that the start-of-work date for "<commit>" is always before the creation date of the Pull Request, as seen in Figure 7.

The resultant Pearson correlation is -0.8468. The further away for 0 we go in terms of time before start-of-work, the stronger the correlation. If we exclude all the points where the time before start-of-work is in the range [-50, 50], which is where all the spread of the points is the largest, the Pearson correlation jumps to -0.966064.

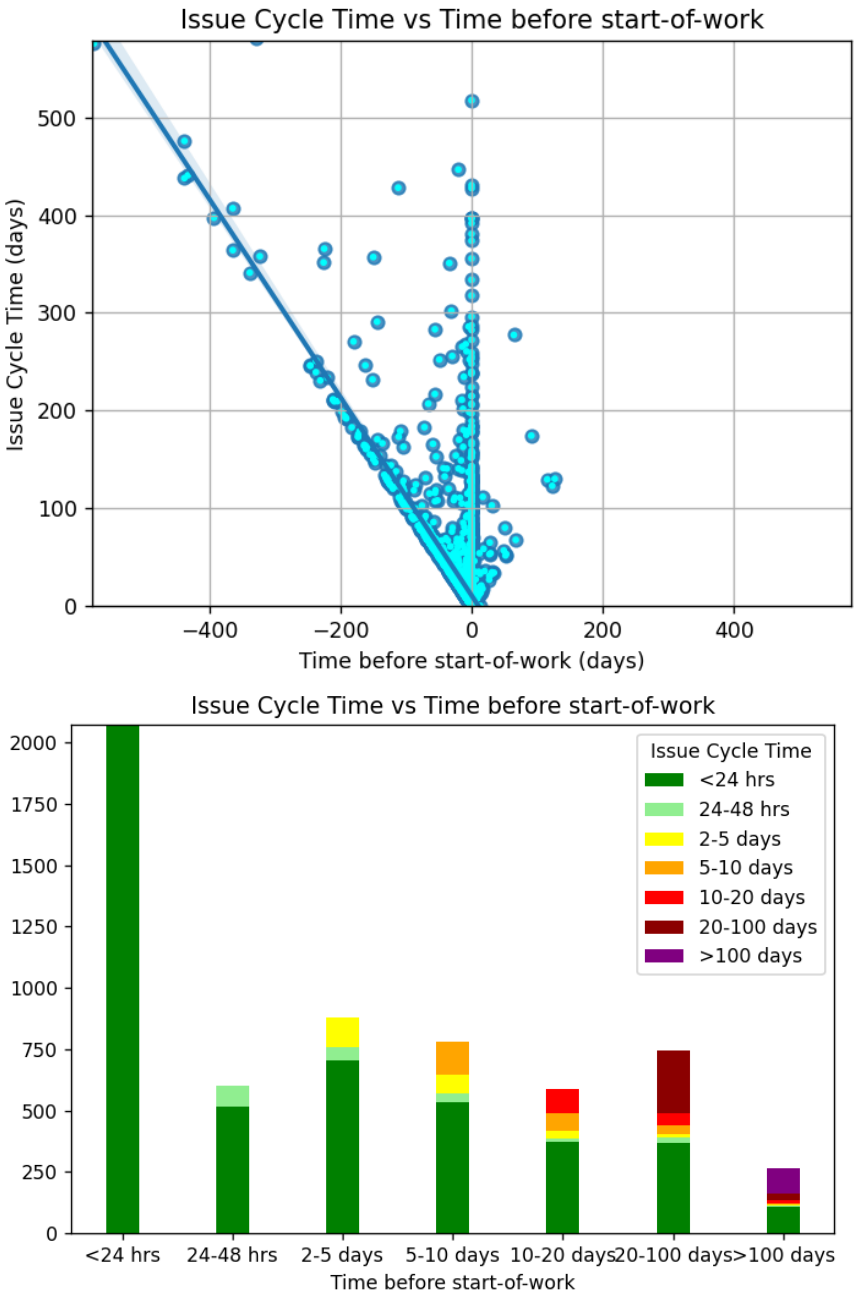

Figure 7: Scatterplot and Bar chart of Issue Cycle Time vs Time before start-of-work for Pull Requests. Most Pull Request are worked on before they are created.

Plotting regular Issues against the time before start-of-work, shows a mirrored version of the plot made with the Pull Requests. For regular Issues, the time before start-of-work cannot be negative, as none of the start-of-work can be triggered before the Issue is created.

The Pearson correlation between the issue cycle time and the time before start-of-work is 0.4314. If we exclude all the points where the time before start-of-work is in the range

[0, 50], which is where all the spread of the points is the largest, the Pearson correlation jumps to 0.8064.

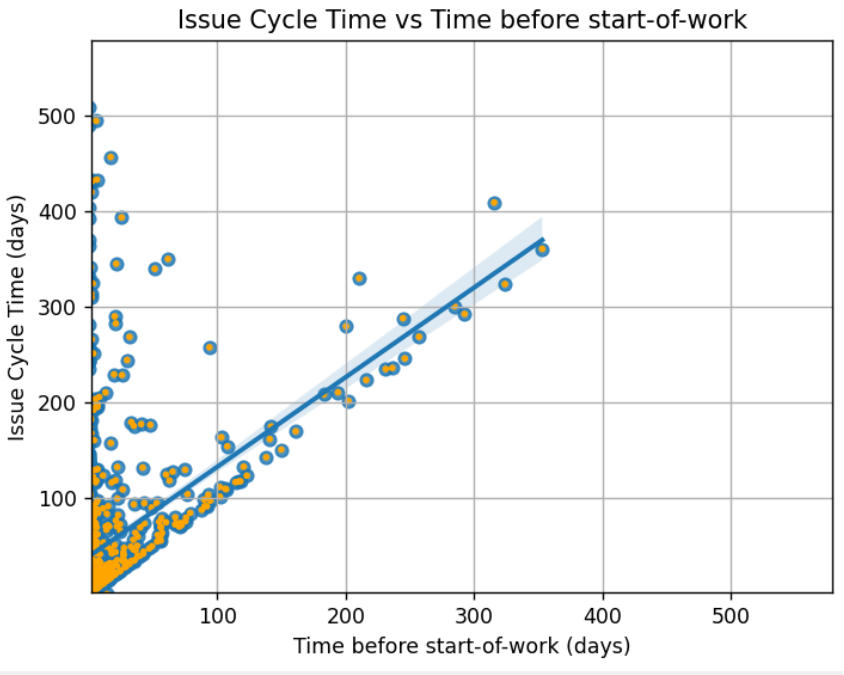

Figure 8: Scatterplot of Issue Cycle Time vs Time before start-of-work for regular Issues.

5 Discussion and Conclusion

The issue cycle time was, for most Issues, less than 2 weeks. The first 2 days alone account for more than a third of all issue cycle times. In addition, there are a very large number of outliers that take much longer to complete. These long-lived issues have an issue cycle time which is close to equal to the total issue duration. This is because the time before the start-of-work is does grow at the same rate as the total issue duration.

The issue cycle time and time before the start-of-work were strongly correlated in our sample. The correlation was positive or negative for Pull Requests and regular Issues respectively. When considering points where the time before the start-of-work was close to zero ($+-50$ and below), then correlation weakens significantly. For those issues where work starts closely before or after the creation of the Issue are not strongly correlated.

The data suggest that the issue cycle time and the total issue duration start to vary less and less as the the total issue duration increases. This can be seen in Figures 8 and 7, where a large number of points near the y-axis turns into a near-straight line as the time before start-of-work increases and decreases for regular Issues and Pull Requests respectively.

The predominant issue type used in the sample was the Pull Request. For Pull Requests, most of them have the start-of-work event type set to "<commit>", meaning that Pull Request don't properly represent work to be done, but rather work that has already

happened. But this does not mean that regular Issues are used to always represent work to be done.

Regular Issues suffer from another problem: It is unclear when work on them is initiated. Since there is no way to report work to a regular Issue directly, the only available methods to approximate the time of the start-of-work are via events that link to other work, like Pull Requests, or via events that assign work to users. Regular Issues are much more likely to be created before work on them starts.

In further research, data on more repositories should be extracted over a longer period of time. This study touches on everything that happens on a yearly basis, which is important. A lot of things happen over the course of a year, like holidays and periods of crunch, which should be included in analysis of the issue tracking data. This reason why this was not included was because of resource constraints. It was not feasible to collect this much data in a reasonable amount of time when the metrics are tweaked over the course of the study. The sample was also not very large (40 GitHub repositories), so this can also be increased in further research.

The large number of missing values was also a limiting factor. If the start or end of work on an issue could not be determined, it was removed. In further research other methods of extracting these metrics can used alongside Git Commits and the Event Timeline to determine them more accurately.

The contributions of this thesis are:

- a detailed analysis of the GitHub events and their role in determining issue duration and cycle times.
- a model for determining issue duration and cycle time that combines the GitHub issue event timeline and the GitHub commit history
- the creation of a dataset with issue data on 40 GitHub repositories and 15K issues
- determined the issue duration and issue cycle times for this dataset
- analyzed the relation between issue cycle time and time before the start of work; these are strongly correlated for larger values of the time before the start-of-work, and weakly correlated for smaller values of the time before the start-of-work

The division into buckets of <24 hrs, 24-48 hrs, 2-5 days, 5-10 days, 10-20 days, 20-100 days, 100+ days seems to work. The first three buckets capture all the Issues that are finished quickly after being opened, with the first 2 capturing most Pull Requests. These Pull Requests seem to represent day-to-day work on the software project as part of their workflow. The "20-100 days" bucket is where time before start-of-work tends to become much longer proportionally to the issue cycle time. In general, projects should aim to resolve their Issues within the first 3 weeks of creating. If a third of the Issues are resolved in that time-span, that is already better than the average.

While issue cycle can be approximated using Git Commit and GitHub Timelines, the accuracy of the analysis can be improved by using more contextual information, such as the number of commits, comments, and contributors.

References

[BB13] Alberto Bacchelli and Christian Bird. "Expectations, Outcomes, and Challenges of Modern Code Review". In: *2013 35th International Conference on Software Engineering (ICSE)*. May 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617. (Visited on 04/07/2025).

[CG20] Dan Chen and Sally E. Goldin. "A Project-level Investigation of Software Commit Comments and Code Quality". In: *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*. Nov. 2020, pp. 240–245. DOI: 10.1109/ICOIACT50329.2020.9332086. (Visited on 04/07/2025).

[Di +19] Marco Di Biase et al. "The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). Montreal, QC, Canada: IEEE, May 2019, pp. 113–122. ISBN: 978-1-7281-3371-3. DOI: 10.1109/TechDebt.2019.00030. URL: https://ieeexplore.ieee.org/document/8785997/ (visited on 04/07/2025).

[DLR10] Marco D'Ambros, Michele Lanza, and Romain Robbes. "Commit 2.0". In: *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*. Cape Town South Africa: ACM, May 2010, pp. 14–19. ISBN: 978-1-60558-975-6. DOI: 10.1145/1809198.1809204. (Visited on 04/07/2025).

[For+21] Nicole Forsgren et al. "The SPACE of Developer Productivity: There's More to It than You Think." In: *Queue* 19.1 (Feb. 2021), pp. 20–48. ISSN: 1542-7730, 1542-7749. DOI: 10.1145/3454122.3454124. (Visited on 04/09/2025).

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability". In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. Sept. 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8. (Visited on 04/10/2025).

[Kal+16] Eirini Kalliamvakou et al. "An In-Depth Study of the Promises and Perils of Mining GitHub". In: *Empirical Software Engineering* 21.5 (Oct. 2016), pp. 2035–2071. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-015-9393-5. (Visited on 04/04/2025).

[Kho+12] Foutse Khomh et al. "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox". In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 2012 9th IEEE Working Conference on Mining Software Repositories (MSR 2012). Zurich: IEEE, June 2012, pp. 179–188. ISBN: 978-1-4673-1760-3. DOI: 10.1109/MSR.2012.6224279. URL: https://ieeexplore.ieee.org/document/6224279/ (visited on 04/07/2025).

[Kov+20] Vladimir Kovalenko et al. "Does Reviewer Recommendation Help Developers?" In: *IEEE Transactions on Software Engineering* 46.7 (July 2020), pp. 710–731. ISSN: 1939-3520. DOI: 10.1109/TSE.2018.2868367. (Visited on 04/07/2025).

[Mic] Microsoft. "GitHub REST API documentation". In: URL: https://docs.github.com/en/rest?apiVersion=2022-11-28.

[MV23] Qianru Meng and Joost Visser. "Which Bug Reports Are Valid and Why? Using the BERT Transformer to Classify Bug Reports and Explain Their Validity". In: *Proceedings of the 4th European Symposium on Software Engineering*. Napoli Italy: ACM, Dec. 2023, pp. 52–60. ISBN: 979-8-4007-0881-7. DOI: 10.1145/3651640.3651648. (Visited on 04/04/2025).

[Opp] Frank R Oppedijk. "Comparison of the SIG Maintainability Model and the Maintainability Index". In: ().

[PyG] PyGithub: Typed interactions with the GitHub API v3. "PyGithub". In: URL: https://pygithub.readthedocs.io/en/stable/reference.html.

[Raz+25] Abdul Razzaq et al. "A Systematic Literature Review on the Influence of Enhanced Developer Experience on Developers' Productivity: Factors, Practices, and Recommendations". In: *ACM Computing Surveys* 57.1 (Jan. 2025), pp. 1–46. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3687299. (Visited on 04/10/2025).

[Rom+23] Anthony Cintron Roman et al. *Open Data on GitHub: Unlocking the Potential of AI*. June 9, 2023. DOI: 10.48550/arXiv.2306.06191. arXiv: 2306.06191 [cs]. URL: http://arxiv.org/abs/2306.06191 (visited on 04/07/2025). Pre-published.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "PyDriller: Python Framework for Mining Software Repositories". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, Oct. 26, 2018, pp. 908–911. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3264598. URL: https://dl.acm.org/doi/10.1145/3236024.3264598 (visited on 04/04/2025).

[Sad+18] Caitlin Sadowski et al. "Modern Code Review: A Case Study at Google". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. Gothenburg Sweden: ACM, May 2018, pp. 181–190. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183525. (Visited on 04/07/2025).

[SCR20] Peter Strečanský, Stanislav Chren, and Bruno Rossi. "Comparing Maintainability Index, SIG Method, and SQALE for Technical Debt Identification". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. Brno Czech Republic: ACM, Mar. 2020, pp. 121–124. ISBN: 978-1-4503-6866-7. DOI: 10.1145/3341105.3374079. (Visited on 04/10/2025).

[Spa+18] Davide Spadini et al. "When Testing Meets Code Review: Why and How Developers Review Tests". In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 677–687. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180192. (Visited on 04/07/2025).

[Spi+20] Diomidis Spinellis et al. *Enterprise-Driven Open Source Software*. 2020.

[TB22] Adam Tornhill and Markus Borg. "Code Red: The Business Impact of Code Quality - a Quantitative Study of 39 Proprietary Production Codebases". In: *Proceedings of the International Conference on Technical Debt*. TechDebt '22: International Conference on Technical Debt. Pittsburgh Pennsylvania: ACM, May 16, 2022, pp. 11–20. ISBN: 978-1-4503-9304-1. DOI: 10.1145/3524843.3528091. URL: https://dl.acm.org/doi/10.1145/3524843.3528091 (visited on 04/07/2025).

[WOA97] Kurt D. Welker, Paul W. Oman, and Gerald G. Atkinson. "Development and Application of an Automated Source Code Maintainability Index". In: *Journal of Software Maintenance: Research and Practice* 9.3 (1997), pp. 127–159. ISSN: 1096-908X. DOI: 10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S. (Visited on 04/10/2025).