Waiting with Architectural Technical Debt

Hanne Bosch

An empirical analysis of the relation between architectural technical debt and software development speed

TU Delft

Waiting with Architectural Technical Debt

An empirical analysis of the relation between architectural technical debt and software development speed

by

Hanne Bosch

Author:	Hanne Bosch	hbosch355@gmail.com
Supervisors:	Mark de Reuver	g.a.dereuver@tudelft.nl
	Nitesh Bharosa	n.bharosa@tudelft.nl
	Zenlin Rooseboom-Kwee	z.roosenboom-kwee@tudelft.nl
	Lodewijk Bergmans	l.bergmans@sig.eu
	Dennis Bijlsma	d.bijlsma@sig.eu

Project duration:	February 1, 2022	until	July 31, 2022
Course:	MSc Management of Technology		MOT 2910 Master Thesis Project

TU Delft

Acknowledgements

I am extremely thankful to the rather large group of people that worked with me to make this project possible. On the academic side of things I am grateful to my graduation committee for their willingness to be flexible. Additionally, Nitesh has always been a source of interesting angles and unending positivity. Mark was able to add structure and create clarity when the situation asked for it. And Zenlin was always a source of valuable criticism and interesting alternative perspectives.

On the business side of things I want to thank the Software Improvement Group as a whole for providing the environment in which this project took place as well as many kindred spirits showing unending interest in my work. Even more importantly, my gratitude goes out to Lodewijk and Dennis, who have spent many hours discussing methods, results and possible new avenues of research with me. I can without a doubt say that without their input this project would not have reached its current level of quality.

Finally, a thank you to my friends an family who have suffered my unending talk about this project. Especially as very valuable input would keep coming, even when the subject became more and more technical. More specifically, I would like to thank my parents who have always taught me that questions can be answered by more than just Google, and my partner who has shared in many frustrations and jubilations along the way.

Executive Summary

Generally speaking there is a certain expectations that if a business relies on something for its core business processes the quality of this something is strongly guaranteed. As digitisation keeps moving forward it is more and more likely that some form of software lies at the core of these essential business processes. Despite this, it is extremely difficult to communicate effectively and clearly about the quality of software.

Software quality is an illusive concept to properly grasp an understand, especially for people without technical knowledge. To combat this (architectural) technical debt is used extensively to measure and manage the quality of software architecture in both business and research. Many scientific authors and experts in the field mention benefits of improved software architecture. For example, it would result in improved reliability, developer motivation and speed of development. However, there is very little empirical evidence to support these claims, making it difficult to quantify what investments in architecture quality are worth, and to convince management why such investments might be necessary. In this project the relation between architectural technical debt and speed of development are investigated. only one of the benefits of reduced architectural technical debt (reduced software maintenance) has empirically been shown to exist. This project aims to fill part of this gap in knowledge by empirically quantifying the relation between architectural technical debt and software development speed.

This makes this project interesting for both researchers and ICT management looking to understand the ways in which software development teams work. There exists a lot of different beliefs in the world of software development about what methodologies improve team productivity. However, it is important, especially with the size and impact that software has on the modern world, to rigorously investigate these beliefs and actually prove their existence. Both for the scientific and the ICT management community.

In order to allow this project to take place in close contact to current practices and to make use of the expertise of a broad range of this project is undertaken in collaboration with the Software Improvement Group. The Software Improvement Group is an Amsterdam based consultancy firm that specializes in helping companies improve and manage the quality of their software. This expertise makes them especially well suited for a project such as this one.

Based on the literature review performed in this project, it becomes clear that there is currently no appropriate metric for speed of development in use in the scientific community. Because of this a new metric is developed within this project. This new metric, by the name of median time per pull, was developed according to the goal question metric approach. Median Time per Pull captures the time it takes to implement a single feature, or fix an issue in the existing code. The metric is validated in a number of different ways including a workshop with developers and is able to capture speed of development of a project which can be further split into the time spent on coding and time spent before the merge.

Median Time per Pull represents a novel way of measuring software development speed. The method is easily used and can be used with data which is readily available for many open source projects and should be kept by almost any other software development company. It results in high granularity and reliable data which can be easily aggregated to make meaningful statements about specific development processes. Median Time per Pull could prove useful in many different research projects looking to measure the productivity of software development processes. Managers might be tempted to make use of the median time per pull metric as a key performance indicator. This is strongly discouraged, as using one-dimensional metrics like this one can have serious adverse effects on development teams as developers attempt to game the system. Specifically it might encourage programmers to create more shorter branches to decrease the median time per pull, rather than actually working on ways to improve the productivity of their team.

Median Time per Pull is used to empirically show that there exists no significant correlation between the architectural technical debt (represented by the architecture rating developed by the Software Improvement Group) of a system and its speed of development (represented by median time per pull) using Kendall correlation tests to establish (the lack of) a significant correlation.

This is unexpected, as many managers report making decisions about their software architecture quality based on the promise of increased speed of development. Many scientific authors mention these effect as well. The fact that this effect was not measured within this project shows that improving architecture quality is not a reliable strategy for improving speed of development.

Additionally, it calls into question the other benefits of software architecture which currently lack a solid empirical underpinning. Nevertheless, it seems extremely unlikely that the effects of software architecture have been so grossly misinterpreted over the past decades. It is more likely that other effects are dampening any effects of architecture quality on speed of development.

Furthermore, this project could seem to undermine the importance of proper software architecture. This is not the interpretation of this author. As was mentioned earlier there are many other effects at play relating to architectural technical debt and speed of development. Empirically and quantitatively investigating these effects could seriously increase the capacity of software development teams to weigh the costs and benefits of improvements to architecture quality. Additionally, determining the exact effects of architecture quality on the development process would seriously aid communication about these more abstract aspects of software development with non-technical people.

Finally, there are a few limitations of this project. Perhaps most importantly the assumption is made that a pull represents a somewhat constant unit of work. While there are a number of reasons why this is likely the case it still remains an assumption and might warrant future research. Additionally, there is some coding work which is not included in the Median Time per Pull. This makes it impossible to say with absolute certainty how long is spent on each pull. Finally, many of the project considered in this project are open source. While there is reason to believe that this should not change the results it would be even more certain if the project was performed using non-open source projects.

Contents

List of Abbreviations

API Application Programming Interface

AR Architecture Rating

ATD Architectural Technical Debt

BSC Balanced Scorecard

CSD Custom Software Development

CC McCabe's Cyclomatic Complexity Metric

FP Function Points

GQM Goal Question Metric

ICT Information & Communications Technology

KPI Key Performance Indicator

LOC Lines of Code

MTpP Median Time per Pull

PPK Public-Private Key

SIG Software Improvement Group

SoD Speed of Development

TD Technical Debt

Definitions

Architectural Technical Debt

Architectural Technical Debt (ATD) is technical debt that pertains to the architecture of the system. This means it is the deviation of the architecture of a system from its ideal state. Depending the context, the scope of ATD can vary widely. For example, in some cases it is useful to talk about ATD as something that is actively taken on (deliberate ATD) while in other settings ATD is something that happens without actively bringing it on (inadvertent ATD), and should be kept low by reacting to it. In this project the SIG architecture model (see section 3.2) will be used in order to gain a measure of ATD, meaning it includes both deliberate and inadvertent ATD, as well as structural ATD, communication methods, data access, technology usage, evolution and knowledge distribution. Practically this means that for this project the measure of ATD will be 5.5 (the maximum architecture rating) - the architecture rating.

Branch

In software development it is common for developers to make a copy of the software system before they start working on implementing a new feature. This gives them a unchanging version to work in, making the finding of bugs and testing behaviour much easier. Additionally, this way whenever a mistake is made during development other developers do not experience any problems because of this mistake. Such a copy of the code is known as a branch. Additionally, the production version of the code is often known as the master or main branch

Commit

A commit is a form of saving progress while programming. After making certain changes to the code a programmer can make a commit, which saves the changes they made withing the file(s). The main benefit of using (GIT) commits over more traditional saving of folders is that it is very easy to revert to an earlier state of the system (in case something breaks) and that it allows for multiple different versions of the same folders to exist in parallel so that many different features can be developed without interfering with one another.

Pull

Whenever a feature, bugfix or other change to the code is completed the branch in which it is created needs to be merged with the production version of the code. This process of merging the updated version of the code with an existing version is known as pulling the branch into the main branch.

Software Architecture

Software architecture is the abstract organisation of a software system. It includes the different components that make up the system, like databases, load balancers or interfaces. Additionally, software architecture describes how these different components interact. Finally, these architectures also define specific paths by which new functionality can be added to the system in the future.

Software Maintenance

Software Maintenance encompasses any activities undertaken on a software project after it's initial delivery. This includes bug fixes, updates and implementation of new features.

Speed of Development

In general, speed of development is the speed with which a team is able to create new features, or fix problems that arise over time. The exact scope of this speed depends in large parts on what measure for speed is used. In this project (as will be expanded upon in section 5) the time between the first commit associated with a pull and the final merge of said pull will be used. This means that the speed of development includes any coding activity, so long as it is developed in a separate branch before merging back into the main branch.

Technical Debt

Technical debt is the deviation of the state of of a software system from its ideal state. This debt has associated with it a principal, and an interest. Principal is the amount of effort or resources that needs to be spent in order to bring the state of a software system to its ideal state, thereby eliminating the architectural technical debt. Interest is the amount of effort or resources that are expended as a direct consequence of keeping the state of a system in a non-ideal state.

1

Introduction

It is over ten years ago that Andreessen made his now famous quote: ”Software is eating the world.” (2011, p1). Despite its age, this quote has shown itself to be more true over time. For the better part of the past century software has started playing a more and more important role in essential processes, such as 24/7 online banking, healthcare (especially during the recent COVID-19 pandemic) and global logistics. As a result society has become increasingly dependent on software for essential processes.

Software quality concerns many different aspects of quality, including the perceived quality by the client but also the capacity of the software to evolve as needs change over time (Fawareh, 2020). This capacity of software to evolve and remain up to date is known as the maintainability of the software. With the current shift of software to software as a service and 24/7 availability this latter part of software quality becomes even more important. After all, if software is unable to evolve it needs to be rewritten in its entirety every time a change in needs arises.

Communicating about the quality of this software is quite difficult. Software is difficult to understand for those who aren’t versed in the actual coding that makes these processes work. Especially so because how software looks and feels to the user does not have to reflect the quality of the code underpinning it. To illustrate why this might be an issue, consider the following. While the average person has little technical understanding of woodworking they are able to see the general construction of a table and will be able to judge approximately how sturdily it has been built. If someone is confronted with a poorly constructed table they will generally not trust it to hold heavy objects and will certainly not construct entire companies on top of it. With software, it is entirely possible that people will incorporate software into their company in such a manner that it becomes critical to their business processes without having any concept of the reliability of this software.

To combat this, a metaphor was created by Cunningham (1992, p30) when he wrote:

> ”Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.

This concept is since known as Technical Debt (TD). This metaphor nicely captures a few core concepts of software quality in a manner that people without a coding background can understand well. There are situations in which it is worth it to implement imperfect solutions. For example to reach certain deadlines or to quickly get to a state where people can play with a product and give feedback. Taking such shortcuts allows developers to be quicker and more flexible. They are not without cost, however, and ideally they will be replaced with more robust solutions sooner rather than later. This is why Cunningham calls these shortcuts a form of debt, yielding short term benefits which will need to be repaid at some point in time. Generally speaking, TD is defined as the difference between the ideal state of a software system, and its current state (Kruchten et al., 2013). The cost of resolving this difference is called the

principal of the debt. Similarly to how debt carries a certain interest, TD also carries interest. As is mentioned by Cunningham any costs associated with this imperfect state of the system (the TD) can be considered interest on this debt. This is perhaps the most communicatively important aspect of the analogy, allowing people without technical know how to understand the cost of poor software quality.

An especially elusive aspect of software quality is software architecture. Software architecture is the abstract organisation of a software system; The different components and how they interrelate. A good architecture is intuitive to understand, allows for room to grow and guarantees other aspects of software quality like security. Generally such an architecture will be defined before developing a system and is updated as the project grows / changes. Software architecture is rather difficult to manage however. Generally speaking, there is nothing forcing a developer to adhere to a certain architecture. And often it might be tempting to implement a simpler solution which does not adhere to the intended architecture. This can slowly cause software projects to deviate more and more from their intended architecture.

At the same time, speed is the name of the game in software development. The market of software is especially quick to evolve and competitive, meaning that Speed of Development (SoD) is often considered extremely important for the success of a software product. Here SoD is the speed with which software maintenance tasks are carried out. As a result in encompasses many different tasks like implementing new features, but also fixing bugs or ensuring software keeps working. It is quite difficult to force faster development. For example, adding additional developers to a team does not necessarily increase the productivity of that team due to the increased communication overhead (Brooks, 1974; Dingsøyr & Moe, 2013). As a result, other aspects that can be focused on by managers which result in increased SoD are extremely interesting.

1.1. Knowledge Gap & Problem Statement

As will be expanded upon in chapter 2, Architectural Technical Debt (ATD) is an especially interesting form of TD. Nevertheless there is little research about the impact that ATD has on aspects of software development other than the maintainability of the software. An often cited reason to improve architecture is to improve SoD, or to increase security and reliability (Besker et al., 2017). But there is very little research into whether improved software architecture actually achieves these goals. Additionally, a lot of research that does attempt to analyse these more tacit aspects is based on single case studies rather than larger empirical datasets. As a result there is little knowledge about how exactly ATD influences software.

Because of this gap in knowledge it is difficult to judge what sort of investments are warranted to improve software architecture. There seems to be a strong consensus that architecture is very important for healthy software development, but the exact benefits are vague and unquantified. This can already cause issues for experts with experience in software development but is even more troublesome when working with managers without this kind of experience. It is especially difficult to express why time and money should be spent on a certain improvement if the benefits are unclear, especially so when competing with new features for paying customers.

Perhaps one of the most interesting aspects that are thought to be influenced by ATD is SoD. As will be further expanded upon in section 2.2, there are no appropriate metrics available which allow for large scale analysis and comparison of development speed across different projects. Instead, many of them are either rather fundamentally tied to the product and thus its architecture quality or they are very difficult to compare across different projects and development teams. Perhaps it is because of this that the relation between SoD and ATD remains underinvestigated.

The relevant problem statement for this project has two major components. Firstly, ATD is widely recognised as a very valuable tool for communication and decision making about the architectural state of a software system. Nevertheless there is little empirical evidence about the effects ATD has on many aspects of software quality other than maintenance cost. Secondly, there are very little methods available to software development teams to improve their SoD.

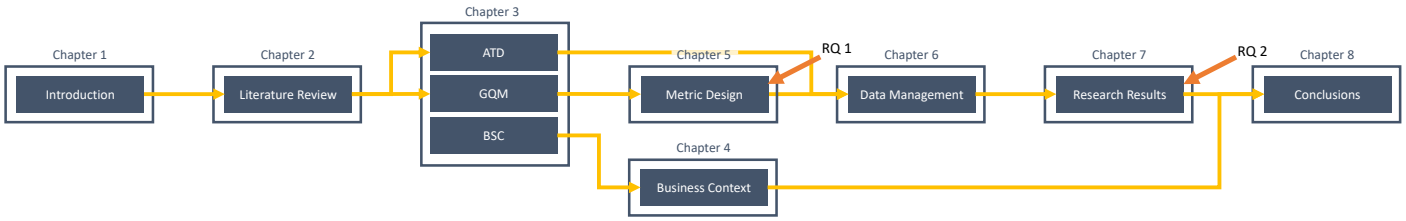


Figure 1.1: Overview of how the different chapters in this project relate to one another

1.2. Research Questions & Thesis Outline

The main research question for this project focuses on the lacking understanding of the effects of ATD and is mentioned below.

> "What is the effect of architectural technical debt on software development speed?"

To answer this question two sub-questions have been established. Answering these will provide the means to answer the main research question. These sub-questions are:

1. What are measures of speed of development that allow for comparison between different projects?
2. What kind of relation exist between the amount of ATD in a project and its SoD?

Figure 1.1 shows the different chapters in this report, as well as how they effect one another and where certain sub-questions mentioned above are considered. In the introduction the general subject, as well as current gaps in knowledge and problems arising from these gaps are introduced. This is followed by a literature review of the two major concepts relevant to this project; ATD and SoD. Based on these findings a method for creating a metric for SoD are chosen in chapter (section 3.3). In the same chapter, the Balanced Scorecard (BSC) framework is introduced as a method for assessing the business implications of this project and Software Improvement Group (SIG)s Architecture Rating (AR) rating is explained. This is followed by chapter 4 where an overview of the different kinds of companies operating in the software development market is given. Additionally, an analysis of the strategies of such companies is performed. Based on the Goal Question Metric (GQM) method outlined in chapter 3.3 a metric for SoD is designed in chapter 5. This metric is then used in conjunction with the metric for ATD to collect and process the appropriate data in chapter 6 which can then be analysed to produce the research results in chapter 7. This is where the second sub-question is answered. Finally, these results are combined with the business context from chapter 4 to analyse the broader implications of these findings in the conclusion.

1.3. Scope

This project concerns itself with the relation between SoD and ATD. No other aspects of code quality will be taken into account in the quantitative analysis in this project (though some will be mentioned as possible explanations for certain behaviour etc.). This means that a number of other aspects that could influence either ATD or SoD are not considered. For example, code maintainability has been found to have a large effect on SoD (Bijlsma et al., 2012), but these effects are not considered in this project. There are a number of strategies which are used to minimise the noise these effects can cause. The exact methodologies for doing so are explained in section 7.1.

Furthermore, this project aims to be an empirical project. Because of this the datasets used are (by design) very large and do not lend themselves well for detailed analysis. While some details will be analysed in order to ensure that the metrics used are indeed working as intended this is not the main method of analysis. As a result the project will not concern itself with individual anomalies in the data, but rather look at larger trends.

Additionally the ATD and SoD metrics themselves are also subject to a particular scope. Regarding ATD a metric is chosen based on the code body. A lot of the more explicit aspects of ATD like code smells are considered in the metric. Some more tacit aspects are also considered, like how knowledge is distributed within the team. All different aspects that are taken into account are described in section 3.2 and appendix A.1.

As is always the case by setting a scope for the project some aspects that could be relevant might fall outside of the scope. For SoD a metric is used based on the commit history of the project. This means the metric only considers coding activity on a feature, not other activities like planning or communicational overhead. An analysis of what threats to validity this might introduce and their effects is performed in section 8.5.2.

1.4. Research Context

This project is performed to gain the title of Master of Science at the Technical University Delft. Additionally, the project takes the form of an internship at the SIG. SIG is a consultancy company based in Amsterdam with a large amount of expertise in measuring and improving software quality. They have developed a maintainability model which has been applied to, and is benchmarked against, a massive library of source code. To expand their services SIG has recently developed an architecture quality model to supplement their existing model. In addition to filling the gap in scientific knowledge mentioned before this project will also serve to validate their model, and assess it's usefulness.

SIG has a long and proven track record as an expert in software quality, and their experience with common practices in industry makes them a perfect partner for this particular type of research. Similarly, their architecture quality model is based upon best practices that are supported widely across the software development industry. Additionally, the collaboration with SIG grants access to the source code and experience of a number of companies, allowing a level of validation that would otherwise prove difficult to achieve within the scope of this project.

2

Literature Review

To ensure that this projects is properly grounded in existing literature and does not ignore previous findings a literature review is performed. For both SoD[1] and ATD the scientific activity surrounding these subjects clearly shows that there is a lot of interest in both of these subjects. In this review, it is found that there is little scientific work attempting to quantify the broader implications of weak or strong architectures. No research was found looking at the relation between ATD and SoD despite the fact that this is mentioned as one of the main reasons to invest in improved software architecture. Additionally, for both ATD and SoD there is no strong consensus on metrics or ways of measuring. In both cases these seem to strongly depend on for what purpose measurements are taken and do not allow for easy comparison across multiple projects. This makes large scale empirical analysis very difficult.

This review will be split into two general topics. Firstly, measuring the cost and benefits of ATD. And secondly, measuring the SoD of either individual, or groups of, software developers. ATD as a scientific field is relatively new, with most of the work being done within the last decade. This means that it is feasible for the single author of this project to perform a systematic literature review. In contrast, SoD is a subject with a much longer history, going back as far as the 1960s. Because of this, performing a systematic literature review for this portion of the literature review is beyond the scope for this project. This longer history does, provide some benefits. The long history means the field is more mature, and there are more generic papers available. This lends itself well for a pearl growing methodology which will be used for this portion of the literature review.

A large number of different methods for assessing both ATD and SoD were considered in this literature review. To keep the size of this report reasonable the explicit discussion of these methods will be relegated to appendix A.

2.1. Review of ATD Literature

it is worthwhile to perform a thorough examination of the state of the art in ATD identification and quantification. As was mentioned in section 1.4, however, one of the implicit goals for this project was to assess the usefulness of SIGs architecture quality model. As such, making use of this metric is a requirement for this project. The review is performed to gain an understanding of the benefits and drawbacks of using this method compared to some of the other methods currently in use.

The concept of TD has proven an invaluable tool for communicating and making strategic decisions about software quality since the early nineties. Since then it ATD is a more recent addition but has nevertheless proven just as useful as conventional TD. Because of this a healthy body of scientific

[1] Rather than discussing SoD directly, much of the literature surrounding speed of development talks about productivity rather than speed. The two concepts are closely related, however, as SoD can be seen as productivity over time. Because of this the literature review that follows concerns itself mostly with developer productivity. If an adequate measure of productivity is found this can be easily expanded to SoD by looking at productivity over time.

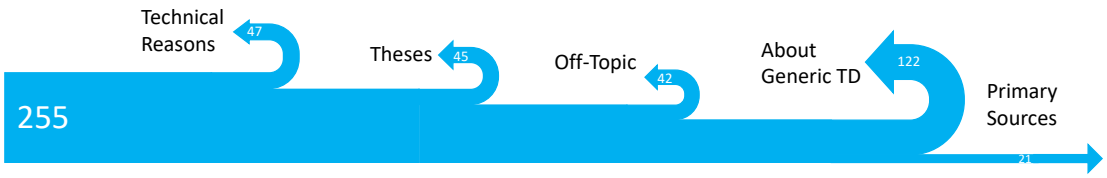


Figure 2.1: Breakdown of how many of the original 255 sources were disregarded for which reason resulting in the 21 primary sources used in this part of the literature review

knowledge already exists about identifying and quantifying ATD. In the following chapter a systematic overview of the different methods of ATD assessment will be provided. To this end two research questions have been formulated, which will be answered in this part of the literature review. These questions are:

1. What are current approaches to assessing and quantifying ATD?
2. Do holistic approaches for assessing the cost of ATD and benefit of ATD repayment exist?

Because this review takes the form of a systematic literature review it is essential to formulate a good search query. This query is then used to find the initial list of papers using Google Scholar as a search engine. The search query was refined over multiple iterations to find a good number of results about the intended subject and is as follows:

”architectural technical debt””cost””benefit”[2]

The quotations around architectural technical debt ensure that this phrase shows up in this manner in the paper and not as separate terms. Cost benefit did not receive the same treatment as this project is also interested in cost or benefits of ATD repayment (not exclusively both at the same time). Therefore allowing the words to show up independent of one another is valid. This yielded a total of 255 search results.

A sub-selection of these papers was then made based on analysis of the title and abstract of the paper. Only a single researcher was available to review the results, while this somewhat reduces the reproducibility it was impossible within the context of this review to add additional reviewers. The exclusion criteria which were used to filter out search results which were not usable can be found in table 2.1. An overview of how many sources were disregarded for which reason can be found in figure 2.1. 47 sources were disregarded because of technical reasons like availability, 45 sources were disregarded because they were master’s or PhD theses, 42 sources were off topic and 122 sources were disregarded because they were about generic TD without explicitly diving into ATD. After this deselection procedure 21 sources remained, which form the primary sources for this review.

Finally, the papers were read in full, and classified according to their findings .A more elaborate overview of the different methods in existence can be found in appendix A.1. The papers can roughly be split into three different categories. Namely: Identification; Growth of ATD and Cost and Benefit of ATD Repayment.

Especially in the first of these categories there are a number of frameworks which help employees identify problematic sources of ATD (Eliasson et al., 2015; Li et al., 2015). As these methods are qualitative in nature they do not lend themselves well for the type of comparative analysis this project aims to perform.

There are a number of quantitative methods in use as well (Diaz-Pace et al., 2020; Mo et al., 2018). Most of these look at how a system evolves over time to find files which change often and with specific

[2] It is worth noting that there are a number of synonyms for ATD in use, like design debt or code debt. However, technical debt is the most common term and the number of instances where sources could only be found under the term of design or code debt are few. While these papers have not been included in the systematic literature review they have been found during less structured reading of material and they do not significantly change the conclusion of the literature review.

Table 2.1: The criteria by which sources were excluded from the original search results.

Criterion	Rationale
Not in English or Dutch	As this author does not speak languages other than English or Dutch to a high enough standard to perform proper scientific analysis sources in other languages were discarded.
Inaccessible	Sources that were not accessible through the TU Delft licence or were found to be corrupted were discarded.
Duplicate	Some sources were found to be among the search results twice, the second results was discarded in these cases.
Secondary Source	Secondary sources such as other review papers and books were discarded to remain as close as possible to the original writing of authors.
Theses	While some PhD and Master's these were very interesting regarding this subject the single author of this review did not have the time to properly analyse the contents of their often hundreds of pages. Therefore these were discarded.
Not on the topic of TD	A portion of the results mentioned (architectural) TD in passing while discussing a different subject. These sources were disregarded.
Not Explicitly about ATD	The lion's share of results did concern TD, but only mentioned architectural TD as a part of the field without being explicitly about this type of debt. Because of this these sources were also discarded.

other files indicating that there is ATD present. Some of the more advanced of these models also incorporates how ATD grows over time (Xiao et al., 2021).

None of these methods lend themselves very well for comparison across different projects. Rather, they seem to be more designed for analysis and decision making within a single projects. Comparison across different projects becomes difficult when they differ significantly in size. For example, if a system with millions of Lines of Code (LOC) has the same amount of ATD as a system with only a few thousand LOC, the former is in a much more troublesome situation. This is where the metric designed by SIG, the AR, becomes especially useful. As will be elaborated upon further in section 3.2 this AR is especially well suited for comparison across different projects and teams.

What is perhaps even more interesting is that, to the knowledge of this author, there are no methods for more holistically assessing the cost and benefit of ATD repayment. It is impossible to get a reach a good estimate of the worth of an investment if not all the benefits of that investment are known. Therefore this represents a significant and important gap in knowledge, which this project aims to help fill.

Based on the findings in the previous paragraphs two things become clear. Firstly, the metrics currently in use for assessing ATD in the scientific community can provide detailed information about the architecture quality in a system, but are less well equipped for comparing systems among one another. Additionally, there appear to be no attempts to fill the knowledge gap outlined in section 1.1 available in the scientific literature. Not with regards to SoD, but also not in other aspects of quality which might be influenced by architecture quality, such as increased security or reliability.

2.2. Review of Development Productivity Literature

The other main aspect of this research considers the speed with which development takes place. In essence the aim is to find a measure which can express the productivity of a team. Once such a measure is found, the amount of productivity over time can be considered a measure for the SoD. Nevertheless, to carry out this project it is necessary to establish a measure of productivity. This field of research is much more mature compared to the field of ATD. Because of this a pearl growing approach is used.

A number of methods for measuring software size or complexity are used in a number of different settings (Honglei et al., 2009). An overview and explanation of these metrics can be found in appendix

Table 2.2: An overview of the different methods of determining productivity in a development team and their attributes. A method is considered automatable if it can be applied without human interaction. A method is broadly usable if it has little to no requirements of the system to work. And finally, a method can either be based on the process, or on the product.

Method	Automatable	Broadly Usable	Process Based	Product Based	Comparable
LOC Count	X	X		X	X
FP		X		X	X
CC	X	X		X	X
Cocomo 2		X		X	X
Story Points	X		X		
Lead Time / Time to Market	X		X		
Issue Resolution Time	X	X	X		X

A.2. A list of the different metrics which were considered is also presented in table 2.2. These metrics range widely in complexity, ranging from simple LOC counts (Fenton & Neil, 1999) to much more complex metrics like McCabe's Cyclomatic Complexity Metric (CC) (Boehm et al., 1995), as well as what they are based upon; the code itself like Function Points (FP) (Albrecht, 1979) or some other aspect of the coding process like story points (Coelho & Basu, 2012). All of these metrics have advantages and disadvantages regarding their use in this project.

It is worth discussing which of these metrics can automatically be applied to large numbers of systems. For example, the Cocomo 2 model requires an expert opinion to provide a good estimate of size, making it extremely difficult to reliably and reproducably apply to large numbers of systems. Similarly, some metrics are easy to calculate if the required data is readily available. This might not always be the case, however. For example, open source projects make their entire version history available, but data from issue trackers is generally not so readily available. An overview of which metrics have which characteristics can be found in table 2.2. Because the aim of this project is to perform a large scale empirical analysis the metric for SoD used must be both automatable and broadly applicable. Finally, it is important that the metric allows for comparison across different projects lest any comparison made between different systems not be valid.

Additionally, within all of these metrics, this author recognises a new distinction in productivity / velocity metrics for software development. Namely, product based metrics and process based metrics. The former of these represents methods which are based directly on source code, performing some analysis on the source code, while the latter is based on the process through which the code is written.

Product based metrics are those that in some way rely on the written code have a rather fundamental problem in the context of this project. One of the main reasons to make use of well defined software architecture is because it keeps the structure of the system clear. Generally this means that implementing a feature in a system with proper architecture will rely on much less (complex) code compared to implementing the same feature in a system with poor architecture (Hohmann, 2003). This means that using a metric based on (or strongly correlated with) LOC counts to measure a correlation between architecture and SoD could run into issues.

The methods that in some manner rely on the process surrounding the development of code don't run into the same problem as the previously mentioned product based measures. Generally speaking the human interpretation of a feature (or similar unit) is irrespective of it's implementation. Nevertheless, this method also runs into issues. There exists no standard size for any of these metrics. For example, how much work is represented by a single story point can vary widely from team to team. As a result it is difficult to compare these metrics across different projects, or even teams. Nevertheless, this issue is not intrinsically linked to ATD making it the preferred type of metric for this project.

Based on the previous paragraphs it is possible to determine which metric is best suited for this project. An ideal measure for SoD would be based on the process of development rather than the

product itself. Additionally, the metric should be easily automatable and broadly applicable. As can be seen in table 2.2 there is only one metric which adheres to all of these requirements, namely issue resolution time. As is further mentioned in section A.2.5, it is possible (or even likely) that issues will be kept on the plank for a long time before actually being worked on. This means that most of the time represented in issue resolution time is not actually spent on coding. Because of this, a new metric similar to issue resolution time will be created for the purpose of this project (see chapter 5).

As a final note, it is important to mention that since recent years using the productivity of a (team of) programmers as a Key Performance Indicator (KPI) is generally considered a bad idea (Ko, 2019), as it can encourage unproductive behaviour. For example, measuring the number of LOC coded could encourage programmers to write overly verbose code, while looking at closed issues could discourage developers from helping colleagues with their tasks. For the metric developed in this project programmers could work to create very small (and short branches) to improve their statistics. Because of this it is unwise to use the metric developed in this project as a KPI.

3

Research Design

Before actually diving into the research performed in this project it is worthwhile to provide an overview of how the research will be performed as well as which pre-existing methods will be used within it. First, an overview of the different aspects in this research will be provided. Additionally, the different research questions, as well as how and where they are answered can be found in table 3.1.

3.1. Overview

As was mentioned in section 1.4, one of the implicit reasons to perform this project in collaboration with SIG was to assess the usefulness of their new architecture model. Because of this, the architecture model will be used as a metric for ATD. As was already mentioned in chapter 2 , the metrics currently broadly in use for ATD are not well suited for this particular kind on analysis. SIGs architecture rating is based on solid first principles and experience from many different customers and experts. This metric is especially well suited for quantitative comparison across different projects making it perfectly suited for this project and is explained in section 3.2.

Similarly, most of the metrics for SoD currently in use are not well suited for this particular type of project. As was discussed in section 2.2, the only metric currently in use which is suited for the project has serious drawbacks. Because of this, the choice was made to create a new metric based on the same core concept. The metric is developed according to the GQM methodology. The explanation of this methodology follows later in this chapter in section 3.3 while the actual implementation of the GQM method resulting in a suitable metric is detailed in chapter 5.

Once these metrics are established, suitable data can be selected, collected, cleaned and validated. This process is laid out in chapter 6. This provides a strong foundation to build any correlation tests on top of. For these correlation test the Kendall correlation coefficient will be used. The different ways in which the data can be analysed to show the (lack of) existence of a correlation will be detailed in section 7.1. Based on this analysis a number of higher level findings can be determined, which will be explained in section 7.2. The business context in which these findings are rooted is detailed in section 4. Here, the BSC framework will be used to assess how company strategies might change to incorporate

Research Question	Chapter	Method
SoD Metric (RQ 1)	5	GQM Metric Design
ATD Metric	3.2	Literature Review
Data Validation	6.4	Data Analysis & Workshop
Relation between SoD and ATD (RQ3)	7	Data Analysis
Main Research Question	8	All of the Above

Table 3.1: Overview of the different research questions, where they are discussed and the methods used to answer them.

Table 3.2: A list of the aspects of quality, and the SIG metrics of architecture quality, and which of these metrics contribute to the which quality aspects.

	Code Breakdown	Component Cohesion	Component Coupling	Code Reuse	Communication Centralisation	Bounded Evolution	Data Coupling	Technology Prevalence	Component Freshness	Knowledge Distribution
Structure	X	X		X	X					
Communication			X	X	X	X	X			
Data Access							X			
Technology Usage								X		
Evolution	X	X	X			X			X	X
Knowledge								X	X	X

the conclusions drawn by this project. After which the project is concluded in the conclusion, including other considerations like threats to validity and possible avenues of future research.

3.2. Measuring ATD

For this project the AR metric developed by SIG will be used. This metric is especially well suited for this project due to its capacity to compare architectures of many different projects. This is a quality many other methods of determining ATD lack. In the following section a brief overview of how SIGs architecture quality metric establishes a quality score is given. A more detailed overview of the different underlying metrics, as well as how they are quantified can be found in Appendix B.

SIG recognises six different components which contribute to architecture quality; structure, communication, data access, technology usage, evolution and knowledge. To quantify these aspects, SIG has developed ten metrics on which to score software to quantify the architectural soundness of the system. How each metric contributes to one of the different aspects of quality can be seen in table 3.2. A brief overview of what each of these metrics represents, as well as how they are measured, will be given in the following section.

3.2.1. Components

Fundamentally, the architecture quality model of SIG is built upon the concept of components. Conceptually a component is similar to a folder, in that it can contain systems, but also other components. This allows SIG to analyse the architecture quality of a single small component. Architecture scores of larger components are generated by aggregating the architecture scores of the smaller components within them as well as scores on how they interrelate until an architecture score of the largest possible component; the entire system, is calculated. This means that all the metrics (see appendix B) are calculated across all different components of the system, and then aggregated into a single architecture score.

3.2.2. Benchmarking

To make sure that the architecture scores can easily be related to what is actually common in the industry all the underlying metrics of the AR are benchmarked. After this benchmark the average score of a metric will be 3, and 90% of the cases will fall between 1,5 and 4,5.

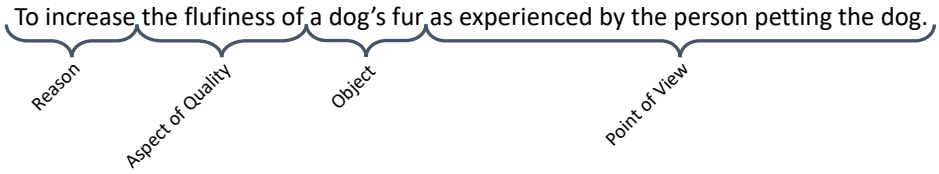
3.3. The Goal Question Metric Methodology

As was mentioned in section 2.2 there are serious drawbacks to the metrics of SoD currently in use. Because of this a new metric will be created to specifically meet the needs of this project. To create this metric in a structured manner the GQM methodology is used. This methodology was originally developed at NASA (Basili & Weiss, 1984) and provides a comprehensive framework for developing metrics especially catered to software development (Caldiera & Rombach, 1994; Van Solingen et al., 2002). In the following section a brief overview of the GQM approach will be given.

The GQM model is a hierarchical model that works in three levels; the titular goal, question and metric. It is built upon the core concept that metrics need to work towards a goal to be useful. This is why the model begins with an abstract goal after which the following questions and metrics are increasingly more concrete.

3.3.1. Goal

As was mentioned in the previous section the goal is the most abstract step in the GQM approach. generally speaking, a goal is defined for a specific object of interest for one or more reasons, aspects of quality, points of view and environments. As an example, a dog breeder might want to improve the fluffiness of their dog's fur, and define the following goal:



Here the object of interest is an actual physical object, a dogs fur. It could also be a more abstract object like a process. The aspect of quality that is being considered is the fluffiness of the fur, which should be increased (the reason). Finally in this case the viewpoint of the person petting the dog is taken. Note that if the perspective of the dog was taken instead what it meant for fur to be fluffy might change. By establishing all of these aspects of a goal it is much more straightforward to create a proper metric to fulfill said goal.

3.3.2. Question

Next, one or more questions can be asked which, when answered provide the means to determine whether the goal is being reached. Questions should be created in such a way that they relate the object of interest to the aspect of quality and the chosen viewpoint. Fluffy furs tend to be thick but not dense. Therefore a second question that could be posed is how heavy (a defined area of) the fur is. There is no reason this needs to be limited to a single question per goal, so to capture the density of the fur as well an additional question could be asked, asking how thick the fur is. These questions relate the object to the aspect of quality stated in the goal.

It is important, to also consider the chosen perspective. Because in this case the perspective of the person touching the dog is chosen the undercoat of the dog's fur, which is usually much denser might not be considered in this thickness as it would almost be considered to be part of the dog's skin. From the dog's perspective, however, this undercoat is definitely important. This shows how important it is to specify a perspective and keep it in mind while formulating the questions.

3.3.3. Metric

Finally, one or more metrics are defined for which contribute to quantitatively answering the questions. Here a distinction can be made between objective metrics, which provide the same result irrespective of the chosen viewpoint, and subjective metrics, which can change depending on the viewpoint. Once

again sticking with the previous example the first question can be answered with a rather simple and objective metric. For example, weight of a cm^2 of fur. Note that this value will always be the same and therefore this is an objective metric. This question serves to answer the first of the questions from the previous section.

The second question leaves more room for interpretation. As was already mentioned at the formulation of the question, where the fur ends might change depending on who is actually attempting to determine its thickness. As the perspective of the person petting the dog is chosen, the undercoat which is much thicker should probably not be considered. Therefore the following metric could constructed; How thick is the dog's fur if measured by balancing a ruler upright on the dog's back. Just like before this is an objective metric. It is possible to conceive of metrics which change in value depending on the perspective. For example, the length of an email exchange could be very long for the person sending the first email and very short for the receiver of this email if the receiver leaves the mail unread for a long time. It is important to clearly establish whether a metric is objective or subjective, and in the latter case, to establish a clear perspective of measurement.

3.4. Kendall Rank Correlation Coefficient

Many metrics within software engineering, and this project, are not distributed normally. Instead, they take to the long tail type of data. This means that many correlation test which rely on the data being normal no longer work. In essence, a long tailed distribution will have a sharply increasing number of occurrences until a peak is reached, after which occurrences become less common, tending to, but never quite reaching zero. A normal distribution on the other hand, has a more gradual increase in occurrences, until the maximum is reached, after which the number of occurrences symmetrically decrease again. Because the data in this project is not normal, a correlation test which can handle this will need to be applied. For this the Kendall rank correlation coefficient will be used.

3.5. Balanced Scorecards

This project concerns itself with the processes surrounding software development. Because of this the project can be very interesting for businesses operating in this field. Special care is taken to discuss the implications for business in the conclusion. To properly do so, it is important to have a clear image of the context in which the project takes place. This context will be established in chapter 4 according to the BSC framework.

Balanced scorecards are a method for ensuring that all different aspects of a business are considered when determining company performance and setting goals. While it still retains financial performance as the main goal of a company there are many other goals which can support the financial performance of a company. Four different aspects of company performance including financial performance are considered. The other three are customer, learning & growth and internal business processes.

There are a number of reasons for choosing the BSCs over other theoretical models. Firstly, BSCs are widely recognised and used in both business and research contexts. This means that literature on BSCs is widely available. Additionally, their usefulness has been shown in many different disciplines (Bieker et al., 2003; Van Grembergen & De Haes, 2005; Zelman et al., 2003).

Additionally, BSCs were developed with the express purpose to supersede exclusively financial goals, but to instead also include the value of processes within a business (Kaplan, 2009). This makes the model perfect for this particular project, as the explicit purpose is to determine whether certain development processes add additional value for the company.

3.5.1. Central Vision

The concept of the BSC is built around a central vision. When first introducing the BSC Kaplan and Norton (1992) found that many very successful companies have in common that they had a very clear vision which returned across all important business processes. As an example, Audi has the following

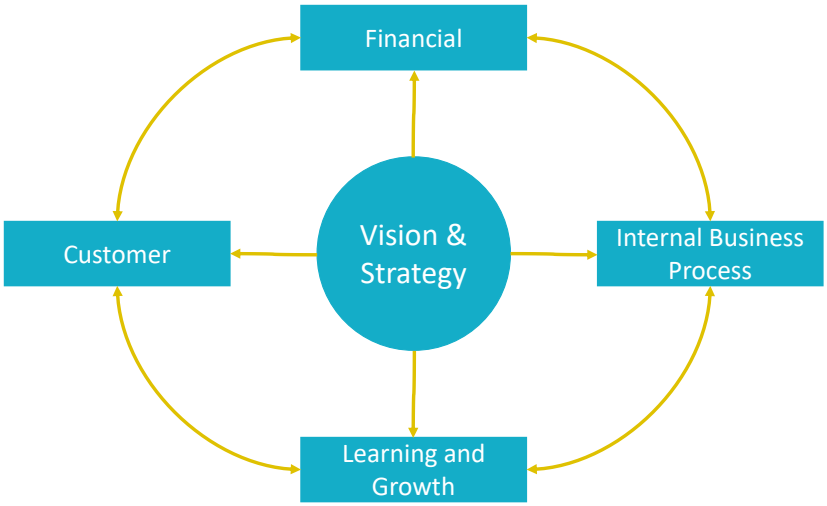


Figure 3.1: An overview of the different concepts in BSCs and how they relate to one another. Reproduced from (Kaplan, 2009)

mission statement:

> ”Vorsprung durch Technik.” (Audi AG, 2022)

Which translates roughly to ”Ahead because of Technology.” This vision should be present all throughout Audi's business processes, which is what BSCs attempt to capture. These processes are categorised into four categories: Financial, Internal Business Process, Learning and growth and customer. A visual overview of how these different concepts influence one another can be found in figure 3.1.

For each of these different categories of processes BSC should establish which objectives should be reached, metrics to measure the progress towards these objective, targets for these metrics to reach and finally initiatives to drive these metrics towards their targets. These become increasingly more concrete and therefore more catered to the specific company in question. Because this project is empirical in nature no single companies are considered in isolation. Therefore, only the more abstract aspects of the BSC are considered.

By combining these four aspects of a business, a balanced overview of what is and is not important to the company can be established, as well as strategies on how to measure, and improve upon these processes. In section 4 the impact of the findings on these different aspects will be elaborated upon, allowing for a clear view of how companies can better incorporate software quality into their strategic decision making.

3.5.2. Financial

As is the case with many frameworks like a BSCs, financial performance remains the most important metric. After all, companies generally function in a for-profit manner. Every other business process should work towards the financial performance of the firm. Generally speaking the financial scorecard looks to answer the question: ”To succeed financially, how should we appear to our shareholders?” (Kaplan and Norton, 2007, p1254). What a financially healthy company looks like might change drastically depending on the product and age of the company. As an example, many companies like Uber and Airbnb don't make a profit, but do quickly amass market share, and are therefore generally considered successful. What exactly healthy financial performance looks like will vary widely and creating financial policy that is in line with the vision and strategy of the company is therefore very important.

3.5.3. Internal Business Process

Internal Business processes represent all the processes within the company that aren't visible from the outside, for example certain quality assurance processes. Some of these can be essential while others

might take a lot of time without actually generating much additional value. The question this card is designed to answer is: ”To satisfy our shareholders and customers, what business processes should we excel at?” (Kaplan and Norton, 2007, p1254). This can, of course, vary widely depending on the vision and strategy of the company at hand. The quality assurance process mentioned above might be essential to companies producing luxury goods, while those competing on price might only ensure that the product falls within the formal safety requirements. Determining which processes are important and which are a drain on resources according to the strategy of the company is essential for its continued success.

3.5.4. Learning & Growth

Learning and Growth is essential to maintain a sustainable competitive advantage. There are numerous examples of companies that go under after failing to properly adapt to (technological) advances in society. Perhaps most famously, Kodak failed because of its inability to adapt to the introduction of digital cameras (Lucas Jr & Goh, 2009). To avoid this it is essential to have a clearly defined strategy on how to learn and grow as an organisation. This scorecards answers the question: ”To achieve our vision. how will we sustain our ability to change and improve? (Kaplan and Norton, 2007, p1254)”. Out of the four different categories, this one is perhaps the most difficult to make measurable due to the tacid nature of knowledge. Nevertheless it is of paramount importance to establish metrics and targets in line with the strategy and vision of the company.

3.5.5. Customer

The final perspective represented in the BSC is that of the customer. Naturally, all companies have some customer, which needs to be catered to in order to remain effective. This particular card asks the question: ”To achieve our vision, how should we appear to our customer?” (Kaplan and Norton, 2007, p1254). Once again exactly how important the relationship with customers is and what this relationship should look like can vary widely depending on the company. For some continued loyalty of customers might be essential while for others customers are lost and gained on a daily basis, which should then be reflected in the scorecard regarding customers of the company.

4

Business Context

To be able to analyse the implications for business it is important to establish a clear context in which the project will land. In this case this means a clear indication of the different strategies being employed by different Information & Communications Technology (ICT) companies. To perform this analysis the BSC methodology (explained in section 3.5) will be employed.

Particularly for software companies it can be useful to make a distinction between ICT companies who sell software as a finished product or as a service and ICT companies that develop custom software which is then handed over to the ICT department of the customer. The former will be called ICT with support companies, while the latter will be called Custom Software Development (CSD) companies from here on out.

Before diving into the specifics of software development it is worth noting that the analysis above mostly focuses on the aspects of software development that are different from other industries. For example, within software development there are companies competing for the lowest price, while others implement additional features to be able to ask a higher price. Because these distinctions exist in all industries, they are not considered of much interest here and are omitted.

4.1. Vision & Strategy

to gain an understanding in the vision and strategies a number of mission statements will be outlined. By looking at a number of these statements and looking for common threads among them it is possible to glean a number of defining characteristics of software companies.

4.1.1. CSD Mission Statements

For CSD companies the choice is made to specifically analyse custom software companies, as these represent the extreme end of the custom software market. The following mission statements of large custom software companies were found:

- **Intellectsoft**: ”Our mission is to help enterprises accelerate adoption of new technologies, untangle complex issues that always emerge during digital evolution, and orchestrate ongoing innovation. Whether it is a consumer-oriented app or a transformative enterprise-class solution, the company leads the process from ideation and concept to delivery, and provides ongoing support through its IS360 framework” (Intellect Software Development Company US, 2022).
- **Oxagile**: ”We enable progressive businesses to transform, scale and gain competitive advantage, through the expert delivery of innovative, tailor-made software. Our elegant, data-driven solutions help organizations and people around the world to perform more effectively and achieve better outcomes” (Oxagile, 2022).

- **OpenXcell**: ”We aim at transforming the digital experience of our customers into cost effective, functional, user-centric and innovative technical solutions. OpenXcell recognizes and adapts quickly to the changing digital landscape thereby empowering clients to uplift their presence in the market” (OpenXcell, 2022).

The main point which seems to be present in all of these statements is an aim for innovative and technologically excellent solutions. This makes sense, as companies like these are often approached for the development of a new tool. If someone is about to build something new they likely want it to make use of modern technologies to ensure that the product remains usable for as long as possible. Additionally, all of them in some manner mention that the product should be well tailored to, and developed in collaboration with, the customer. Once again, this makes sense, why would someone approach a company to build them a solution which is already available to the market. In conclusion, the focus for **CDC!** (**CDC!**) companies layson custom and technologically advanced solutions.

4.1.2. ICT with service Mission Statements

Just like with CSD companies, is is worthwhile to take a look as the visions and strategies often employed in the ICT with service software development world. Below a number of different missions as stated by large software companies are listed.

- **Microsoft**: ”Empowering others; Our mission is to empower every person and every organization on the planet to achieve more” (Microsoft, 2022).
- **Oracle**: ”Our mission is to help people see data in new ways, discover insights, unlock endless possibilities” (Oracle, 2022).
- **Adobe**: ”Changing the world through digital experiences. Great experiences have the power to inspire, transform, and move the world forward. And every great experience starts with creativity” (Adobe, 2022).

There are two threads that seem to permeate most of these missions statements. Firstly, they resolve around providing tools to (a large number of) people. Secondly, they all revolve around the use of forward moving technology. Based on these two threads A statement can be made regarding the strategy these companies likely employ. Innovation seems to be very important for these companies. Because individuals can easily and cheaply switch to other software providers if the competition provides more features, it is very important in ICT with service software development to remain technologically superior to their competition. Additionally, there is a focus on reaching many people. Once again this intuitively makes sense, since the variable costs associated with an additional licence of a product are very low compared to the fixed development cost. This makes it so that additional customers very directly contribute to the additional profits of the company. The conclusion can be drawn that these companies focus on providing technologically advanced solutions to a large public.

4.2. Financial

Naturally, financial performance has two main aspects; the costs and the income of the company. The costs for software development are generally mostly the costs of employing programmers. This means that the name of the game for reducing cost in software development is efficiency. Strategies which could help to increase efficiency are discussed in the internal business processes section.

For the other aspect of financial performance; income. There are a few things that can be done to increase performance as well. Naturally, a higher price can be asked for a product which is higher in quality. Specifically for ICT with service software development it should be noted that a rather unique strategy becomes possible. The variable costs of an additional licence for a software product are often very small as the product has already been developed and creating a new copy of it is almost free. This allows software companies to serve truly massive numbers of customers. As a result, relatively small improvements to the product resulting in a very small relative, but massive absolute, increase in

market share can be worth large investments. As an example, Alphabet inc., most well known or their Google search engine, has well over 1000 employees actively working on the core search algorithm of the service (Fox News, 2018). Despite this, the way their search service works (to their customers) has not changed significantly since its first introduction in 1998. In this manner, Alphabet inc. can receive tiny profits per customer but massive absolute profits.

4.3. Customer

Because the customer of a CSD business is fundamentally different from a ICT with service business it is worth discussing the two entirely separately.

4.3.1. CSD Customers

Whenever a business makes the decision to acquire a piece of custom software it is likely they will carefully weigh their formal requirements and the specific competencies of the company developer against their cost. This means that while reputation will still play a role in these transactions it will be less important than when an individual makes a software purchasing decision.

What likely will make a big difference is whether the software supplier can meet the formal requirements of the customer in a reasonable timeframe. Simultaneously, modern software development practices tend to make use of agile working methods. This means that the requirements for the final product do not need to be finalised at the very beginning of the project, but can be changed as new insight emerge as the project develops. This requires close collaboration between the customer and supplier.

This also means that the switching costs for businesses are generally quite high. The software being developed can be highly customised. Additionally, a joint development process as mentioned above requires investment in good relations between the customer and developer. If the customer switches supplier these investments will have to be made all over again. On top of this, the kind of custom software that is developed in these kinds of settings is generally highly specialised, meaning no alternatives can be used without heavy modifications. This is further exacerbated by the fact that changing software in a company often involves many employees having to adapt to the new software, as well as needing to integrate the new software into all existing systems. This creates a very strong incentive to remain with the same provider.

4.3.2. ICT with service Customers

Individual consumers of software are quite different. Individuals might be more likely to base their choice on the reputation or brand of the company in question (Mao et al., 2020). At the same time it is not feasible for ICT with service companies to build the same intimate relationship with their customers as CSD companies can. They are much more dependant on (mass) marketing to maintain the relationship with their customers.

Partially because of this, but also because ICT with service tend to sell much less specialised software, it is much more easy for an individual consumer to switch to a different supplier. The less specialised nature of this software means that there are likely ready made alternatives available. Additionally, switching to a different product is often as simple as installing a single new software package on a computer and removing another, as individuals rarely make use of the high level software integration larger companies do. This means keeping existing companies requires much more effort compared to **CDC!** companies.

Finally, as was already mentioned in section 4.2, it is feasible to serve enormous amounts of people with a single software product due to the low variable cost associated with additional software licences. This means that large investments in small improvements can be worthwhile if it results in slightly larger market shares. This aspect of strategy is also something that can be found in the mission statements for ICT with service companies.

4.4. Learning & Growth

In the mission statements of CSD companies, the importance of innovation is mentioned very prominently. For the ICT with service companies this is slightly less clear, but these statements also talk about ”achieve more” and ”unlock endless possibilities”, hinting strongly at the fact that innovation is very important in this part of the market as well.

This is not so strange, as software is a field with many groundbreaking innovations rapidly entering the market. For example, AI is quickly becoming more and more common in the industry (Aggarwal et al., 2022) and quantum computing is starting to enter the market as well. Both of these technologies have massive impacts on what is possible with software. Quantum computing could break the Public-Private Key (PPK) encryption on which much of the internet relies some time in the near future (Mavroeidis et al., 2018). This is a good indication of how large the impact of these technologies, following one another in rapid succession, can be.

to remain competitive in such a market it is essential that employees are encouraged to learn about new techniques. To make sure this happens efficiently, it might be beneficial to allow developers to build up certain areas of expertise. Additionally, it is important that employees do not get stuck in large amounts of bureaucracy while working, as this can stifle internal innovation and learning (Janka et al., 2020). If such learning does not take place, companies will quickly lose the knowledge required to remain competitive.

Having discussed innovation, growth is also an important point of interest in software development. Reliably growing ICT companies without massively increasing communication overhead cost can be very difficult. One possible measure to reduce this is to heavily modularise the software produced (Alshuqayran et al., 2016). In line with this, creating small teams with dedicated tasks could further increase efficiency, which would also improve the financial performance of the company, as was mentioned in section 4.2. Maintaining such efficient teams becomes especially difficult, but even more important, when growing (large)c companies.

Another interesting aspect of software development is that the product often needs to be able to continue growing as well. In the case of custom software products, when these products are handed over, it is important that the customer’s ICT team is able to understand and work with the software. While it might be feasible for a team who has slowly gotten to know a system to work with the quirks a system has gained over time, this becomes rather troublesome if an entire team is changed. Additionally, it is important for developers to be able to understand how code works before working with it, which is much easier if the code is well structured. Both of these aspects hint that well structured code is important to allow the software to keep evolving over time.

For ICT with service companies on the other hand, products could remain in development for quite some time. As an example, Microsoft Word has existed since 1983 and is still in active development (though there have been major re-releases in between). This means that the software needs to be able to evolve without becoming too burdensome to work in. To do this it is important to invest in aspects of code quality that keep maintainability high (Bijlsma et al., 2012). Architectures which contain defined avenues through which new functionality can be added to existing software can also greatly increase the evolvability of software.

4.5. Internal Business Processes

Based on the previous sections the main goals of the internal business processes should be to remain efficient while guaranteeing high software quality and allow for growing the company. Keeping ICT companies operating smoothly as they grow is notoriously difficult, however. Coding requires many different components to work together well. As more employees start working on a project communication between these employees becomes more and more of a time sink. This is one of the reasons that keeping teams small and giving them dedicated responsibilities was mentioned as essential in section 4.4. Additionally, efficiency can be increased by shortening communication lines, or improving developer wellbeing (Forsgren et al., 2021). Forgoing such measurements could quickly slow down development to a snails pace.

At the same time, software development tends to come with a number of boundary conditions for a piece of software to work well. For example, a product that is unreliable or unsafe can cause massive problems for the customer. As a result these aspects of quality should always be kept in mind while developing, even if they are not explicitly stated as requirements for the product.

As was already mentioned in section 4.3, for CSD companies it is often very important to maintain good contact with their customer throughout the design process to ensure that the product is indeed what is required by the customer. Generally it is very difficult to determine the exact form a software product should take beforehand. This is why agile software development methods have gained such popularity, as they account for this by allowing for many course corrections throughout the development process. To do this, good communications with the customer is essential.

5

Metric Design

As has been mentioned in previous sections, a metric to measure the speed of development within a project will be developed to better suit the needs of this project. To do so the GQM approach will be used to create a solid foundation for this new metric. The GQM methodology has already been discussed in section 3.3.

5.1. Goal

The first step in developing a new metric is to establish a goal which the metric aims to help achieve. The following goal is chosen:

> To measure the time it takes to develop a new feature, or resolve an issue as perceived by management.

In this goal the reason for measuring is simply to measure. While in a business context it might be more useful to cast judgement on whether the quality of the object is adequate there are a number of downsides associated with such lines of reasoning (Forsgren et al., 2021; Prechelt, 2019). These mostly fall in the category of people being encouraged to work on certain tasks to increase their personal performance in favor of the performance of the whole team. Because in this case the goal is only to measure the statistic, these aspects of measuring programmer productivity won't be as important. The aspect of quality of interest is the speed with which tasks are resolved. Speed is measured as the difference in time between the start of the task and the resolution of the task. In this case the object of interest is the process through which tasks are carried out. Tasks can both be an issue with existing capabilities of the software, or the time it takes to develop a new feature. The point of view of management is chosen. The main reason for doing so is that this most closely represents the business aspects of software development. What is generally important for the health of the business is that a strategic decision can be made to move the product in a specific direction, and that this decision can quickly be brought to fruition within the product. Because of these reasons, this goal closely represents what is generally meant with development velocity.

5.2. Question

The question chosen to ask to answer the goal mentioned above is as follows:

> How long does a developer typically perform active work on a single feature or issue before its completion?

This might seem like a very straightforward question, but there are some aspects of it that are worth discussing. Firstly, a feature or issue is quite a broad line of possible bodies of work. Nevertheless, it

is extremely difficult to create a clear distinction or representative standard of such a unit. And even solutions to very small problems can be extremely important in the right circumstances and are therefore still of interest. Because of this the decision is made to include all of them.

Secondly, the choice is made to focus on the time taken while working on the task. An alternative might be to measure the time between the creation of the task, i.e. the point where someone decided that this was a task that required attention and it completion. One significant downside for such a metric, is that it is possible that a task is created but not considered high priority. This can cause issues to exist for years on end because they are considered ”nice to have” but other aspects have priority. Additionally, if a task is sufficiently important it is possible for management to decide that other tasks should be postponed in favor of a new task. This means that the time spent actually working on the task is a better representation of the flexibility in performing tasks for the development team and therefore more interesting for this project.

5.3. Metric

The final step in the GQM method is to create one or more metrics to answer the posed question. In this case a single metric is chosen, as the aim is to show the (lack of) existence of a correlation. Because of this it is very useful to have a single metric of interest[1].

As was mentioned in section 2.2 the different methods for measuring productivity can be broken down into two categories: Product and process based methods. The former has a very fundamental problem for this particular project. Because the volume and / or complexity of the code needed to implement a feature increases as ATD increases it is very difficult to construct a convincing argument about whether the two are or are not related. As an illustration, consider two identical programmers (A and B) implementing the same new feature in similar systems. Programmer A works in a system with very low ATD and implement the feature quickly and elegantly. Programmer B, works in a system with rampant ATD and needs to work around a large number of sub-optimal solutions of their predecessors. Because of this, it takes a long time and requires the solution to keep in mind a large number of edge cases and exceptions. If the productivity of these two programmers is measured by the code they have produced, one could conclude that while programmer B took longer they did more work in that time compared to programmer A. This is exactly the kind of interactions this project aims to clarify. Therefore, metrics that rely on the product are rejected because of their very fundamental relation with ATD.

This leaves the other method for measuring productivity based on the process. The perception of a single feature in the human mind is largely independent of the context in which it is implemented. The methods mentioned in section 2.2 all have serious drawbacks. Because of this the choice was made to create a new metric which attempts to avoid these issues. Because of this, a method is chosen based on the process of implementing the solutions. The fundamental concept of the issue resolution time metric is taken as a basis. Instead of measuring the full duration of the issue, the decision was made to base the metric on the commit history of a project. This allows the metric to take into account most of the actual development that has been done. It is worth noting that the time taken to code before the first commit is not included, making the metric slightly lower in value than it should otherwise be. Additionally, commit histories of open source projects are generally entirely public. This allows for the collection of high quality and granularity data.

How long it takes to develop a feature can also be extracted directly from the commit history. Generally when a new feature is developed, an isolated copy of the project is made to allow for its development without interacting with any other development that might be taking place. Such a copy is called a new branch. When the feature is finished and ready to be incorporated into the main project this branch is pulled back into the main branch. The time between the creation of the new branch and the pull of the branch back into the main branch is taken as the length of a pull. This is then assumed to be a reasonable proxy for the time taken to develop a feature. The full definition of the metric is then as follows:

[1] The final result of this process is a single metric (median time per pull) which can be further split into two sub-metrics (coding time and merge time.).

> The difference in time between the first commit in a branch, and the final commit in that same branch.

What exactly represents the final commit in a branch is not entirely set in stone. The merge commit is the point where the the branch is truly merged into another branch, but this commit generally contains no code changes. Rather this represents the time in which quality control and other such processes take place. The commit before the merge commit then (usually) represents the last commit where actual coding takes place. The Median Time per Pull (MTpP), which includes the merge commit represents all work taking place in a branch. This can then be split into commits in which coding takes place, denoting the coding time, and the time between the merge commit and last coding commit, which denotes the merge time. This final time is taken as a indicator of review time in this projects. This can then be used as a proxy for the amount of communicational overhead for a project. Because of this way of working pulls with only two commits (one coding commit and one merge commit) are excluded from the analysis. This is no great loss, as these pulls only contain data on the merge time, and have a coding time of zero. This makes it difficult to judge what the actual effect of these pulls on the working speed for a project is. Together the merge time and coding time provide detailed and high level data on the coding process.

5.4. Measuring the Length of a Pull

In order to measure these metrics a broadly applicable method for determining them for many projects needs to be developed. How this information can be gathered from the GIT history is explained below.

5.4.1. Determining the Length of a Pull

Because of the way GIT is built, references only point to parents and never to children. This allows GIT to build upon existing information without changing said information. This means that finding all commits must start at the (chronologically) last point in the pull. Luckily this is easily found by searching commits which have more than one parent. From this merge commit a string of parents can be constructed. In figure 5.1 this process follows the feature branch.

It is slightly more difficult to determine where to end the pull. This is because there are no attributes of the split commit designating it as a commit where a branch is split off. As can be seen in figure 5.1, there are two commits in the commits history which refer to the split point. This fact is used to determine where to terminate the pull. For each commit it is counted how often their parent is referenced as a parent in the entire git history. A diagram of this is shown in figure 5.1. This fact is used to determine which commit in the feature branch should be the last.

This allows for the collection of a number of aspects of a single pull. In particular, the points of interest which are collected are: The number of commits in the pull, the timestamp of the first, merge and last coding commit, and the number LOC added and deleted. The script used to perform this analysis can be found in appendix C.2. For this analysis R is used (R Foundation, 2022). The reason for this is the fact that R has been designed from the ground up to be well suited for statistical analysis and visualisation. Additionally, this author already has reasonable experience working with R and the ”dplyr” and ”ggplot” libraries. After performing this analysis, aggregating the results per project provides interesting insights into the development process of these projects.

5.4.2. Nested Branches

It is worth mentioning what happens if a branch is split off from a branch which is not the main branch. Depending on the working method this could be quite a common occurrence, for example if a new version of software containing many different new features is developed. An example of what such a situation might look like is shown in figure 5.2. In this case there are two pull requests, one merging the feature branch into the version branch, and one pulling the version branch into the main branch. This would be an example of a nested branch.

The algorithm would treat such a nested pull as follows: The starting point for each pull within the

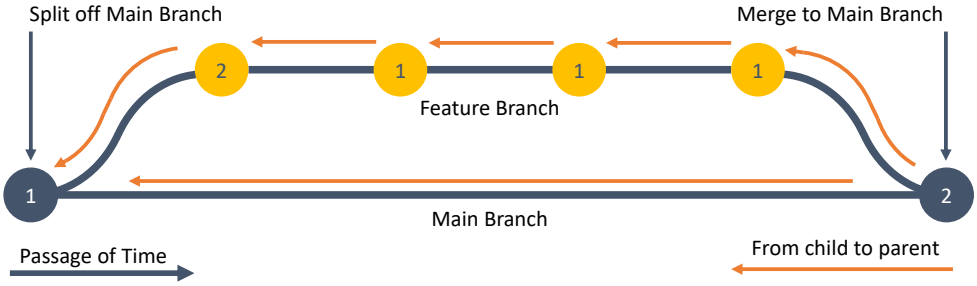


Figure 5.1: Diagram representing a pull in a GIT history. Note that the passage of time and paper-trail of child-parent relations flow in opposing directions. The commits coloured in yellow represent those that are counted in a pull. The number in each commit represents the number of times its parent shows up as a parent in the GIT history.

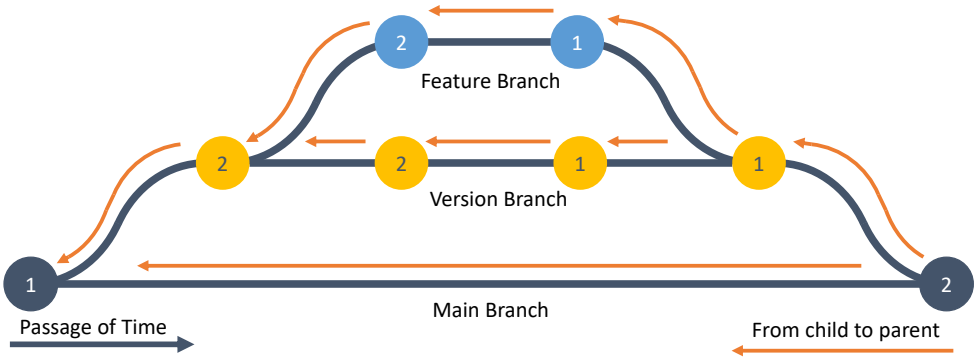


Figure 5.2: Diagram representing a pull and a ”nested” pull in a GIT history. The number in each commit represents the number of times its' parent shows up as a parent in the GIT history.

tool is a commit with more than one parent. In figure 5.2 these are the rightmost blue and grey commits. From here the parent is taken, such that the string of commits moves up one branch. From there the string of commits is followed until a parent is found which is referenced as a parent more than once. In this case that is the same point for both branches, the left-most yellow commit. This means that the version branch is terminated early compared to the actual length of the branch. This slight error is acceptable because the number of version branches compared to the number of feature branches should be very small the difference in final statistics should be rather small.

5.5. Overview of Median Time per Pull

To provide a more easily interpretable overview of how the MTpP metric is built up, as well as how it should be interpreted the diagram shown in figure 5.3 was created. The figure illustrates how the data collected are the commits associated with the project. These commits are then aggregated into sequences of commits representing a single pull (or branch). The statistics of these pulls (like initial commit merge commit etc.) are aggregated once more into a single datapoint representing the mean values for a system in a specific year. This means that a single system has multiple periods in the dataset. One such period then becomes the unit of analysis for this project. This diagram also makes clear what exactly the MTpP score represents. Namely, the median time for which a branch exists for a system within a particular year. A longer time means that branches (usually) exist for a longer time, indicating that implementation of a single feature takes longer.

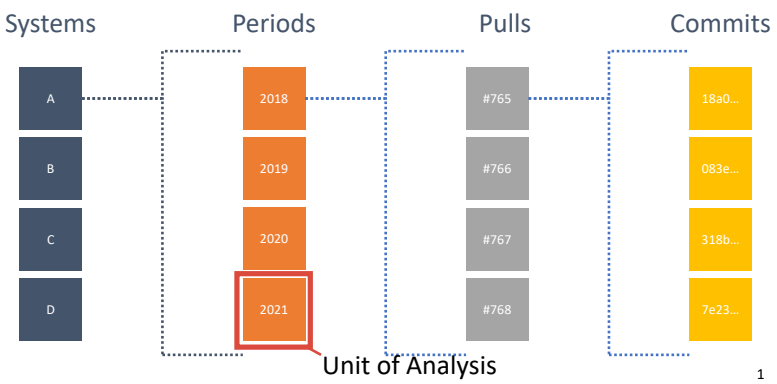


Figure 5.3: Overview of how the MTpP metric is built up, including what forms a single unit of analysis.

6

Data Management

During the literature review in chapter 2 it became clear that there is little empirical research into ATD. Because of this, this project aims to provide a solid empirical entry into the body of knowledge on ATD. To do so it is necessary to create a dataset to analyse. The following section describes how this set is created, as well as what additional data manipulation is carried out. Finally, a number of validation steps are carried out to verify that the metrics are working as intended.

Before discussing more detailed specifics it is worth discussing how the data under analysis is created on a broader and more abstract level. An overview of how the data processing takes place can be seen in figure 6.1. A repository is cloned from a service like GitHub. Based on the git history, the MTpP can be extracted, and further delineated into coding time and merge time. In order to establish the AR code of a project (and how it evolves) is used. This allows the establishment of ten different aspects of quality which are aggregated into a single AR. To determine whether a relation between AR and SoD exists MTpP, coding time and merge time are plotted and correlated against the AR. This provides the basis for this empirical project.

6.1. Data Selection

To guarantee that enough data about a project is available to run an analysis the choice was made to analyse Open Source projects (Perens et al., 1999). These project provide their source code free to all to allow anybody to use and improve the software.

Because Open Source projects are chosen there are some possible differences between these projects and a normal project which is developed more traditionally within a company. Open Source projects are much more likely to have contributors who only work occasionally on the project, which can have quite serious effects on the workflow within a project. There are, however, projects which are open source, but are maintained by defined teams within a company such as Netflix's Zuul or Apple's Swift. This means that these projects are much more representative of non open source projects. Additionally, a number of systems developed by SIG and two of their clients (these particular systems are anonymised) are included in the analysis to ensure that there is indeed no significant difference between these open source and ”normal” software development. A full list of projects can be found in table 6.1. These projects should provide a representative dataset for the project.

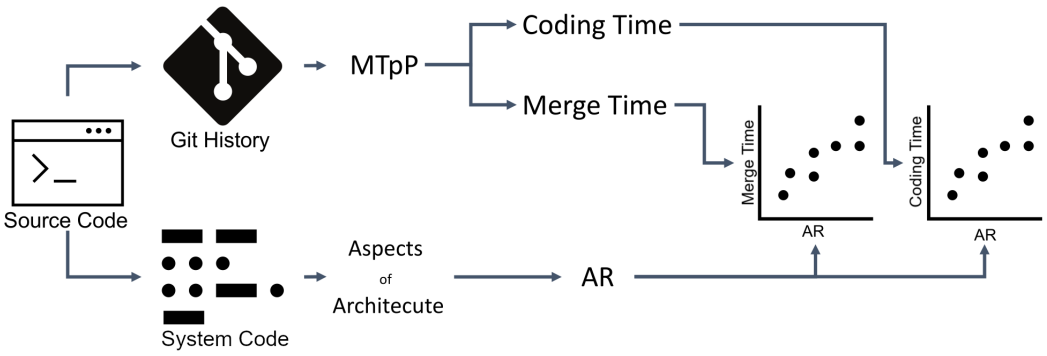


Figure 6.1: Diagram of how the analysis of this project is established. Based on the project repository, the git history of a project can be extracted, which can be used to determine the MTpP. This can then be further split into coding time and merge time. Based on the actual files in the repository ten different aspects of architecture can be analysed. These are then aggregated into a single AR. These different metrics are then plotted and correlated against one another to determine whether a relation between them exists.

Owner	(Number of)System(s)	Number of Periods
Netflix	Zuul	9
Bitcoin	Bitcoin	12
Github	Atom	10
Apache	Echarts	8
Alibaba	FastJSON	10
Oracle	Graal	10
Gradle	Gradle	13
Jiti	Jitsi Meet	8
JRuby	JRuby	20
libGDX	libGDX	11
Puppet	Puppet	16
Rasa	Rasa	5
Supertux	Supertux	18
Apple	Swift	10
Webpack	Webpack	9
Axios	Axios	8
Deno	Deno	4
FFmpeg	FFmpeg	21
Microsoft	Visual Code	6
	Typescript	8
Red Hat	Quarkus	3
	Vert.X	8
.NET Foundation	Roslyn	7
	Roslyn Analysers	5
Google	GSON	12
	Tensorflow	6
Company A	33	58
Company B	54	185
Company C	7	23
Total:	120	534

Table 6.1: Overview of the different systems included in the analysis. In the case of SIG and her customers a number of systems rather than a specific system is named.

Only having access to the source code is not enough. It is also important that the branch structure is still present in this history. There are a number of actions that can be taken that would alter the commit history in such a way that this is no longer the case. One example which is often used is to squash before merging a branch. This creates a single commit containing all the changes that were made over the entire branch, which is then the merge commit. This means that if this method is used for the majority of branches the project is not usable[1].

6.2. Data Collection

to collect the data a previously developed tool by the name of PyDriller is used (Spadini et al., 2018). This python tool makes use of githubs REST Application Programming Interface (API) (GitHub, 2022), or similar APIs by other services to scrape the commits of a particular project. A number of different metrics are then collected for each metric and saved into a .csv file containing all commits for a particular project. This file can then serve as the input for the tooling created in this project to determine which commits belong to which pulls. The scripts used to run PyDriller can be found in appendix C.1. The main reason for making use of Python for the data collection is mainly the availability of the PyDriller tool, as well as the fact that this author is already familiar with Python as a language. This method allows for quick and easy scraping of commit histories.

6.3. Data Cleaning

The commit history of a project was chosen to ensure high quality data could be retrieved. Nevertheless, there are still some aspects of the data that need to be cleaned before proper analysis can take place.

Because git allows users to manipulate quite some aspects of the git history, the order in which some operations take place can change. This can result in a series of commits, where the parent of a commit was committed later than its child. Naturally these cannot be the actual timestamps,so such commit chains are removed from the analysis.

There are a number of projects where development does still take place, but at a rather slow pace. To make sure that data points are not based on very little actual activity, any data point which is based on a small number of underlying (series of) commits is disregarded. In a similar vein, series of commits which contain a very large (> 100) number of commits are disregarded. Such series are extremely unlikely to represent single features, and are more likely to represent a separate version of a system.

Additionally, the analysis will be carried out over multiple years. This means that each project will be analysed multiple times. To create clear boundaries between these snapshots only commits within the calendar year will be considered in any series of commits. This means that if a series of commits starts in a different calendar year than it ends, that series of commits won't be taken into account in either of the snapshots. While this might favor smaller series of commits (as these are less likely to cross the 1st of January) the alternative creates a lot of ambiguity in what is and is not included in a year of analysis. While this approach disregards some data it allows for clearer delineation between years.

The aggregate of these pulls (per year) are then joined with the results from the architecture tool developed by SIG. Because the point of interest for this project is the effect proper architecture has on development speed the choice was made to measure the AR at the start of the year (based on information from the year before) and combine this with the MTpP data from the year after. Figure 6.2 shows a diagram of what a single datapoint represents.

[1] It is worth noting that the information which is lost by squashing or otherwise manipulating the commit history of a project is not entirely gone. It is still contained in the reflog. The reflog by default only keeps three months of information before removing entries, making it impossible to retrieve data over long periods of time in this manner.

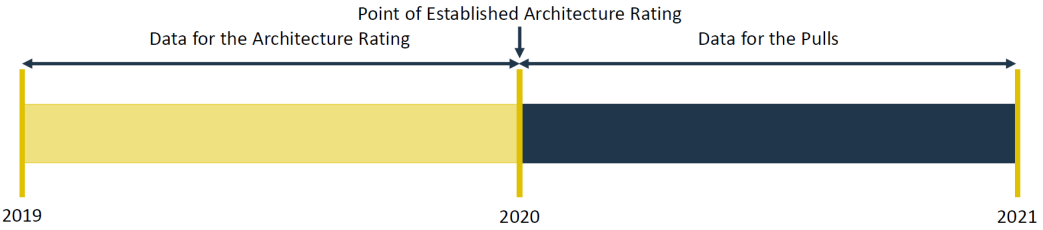


Figure 6.2: Diagram showing on which data different aspects of the analysis are based. The AR is established at the start of the year under analysis with the previous year as data to build upon. The pulls associated with that AR are then drawn from the year following the establishment of the AR

6.4. Validation Of Median Time Per Pull

Before moving on to investigating the relation between ATD and SoD, it is worth spending some time on validating the MTpP metric.

6.4.1. Workshop with Company A

to validate the findings a workshop was organised with employees of one of the two companies whose data was analysed. If the data agrees with the image employees of the company have about their software this is a strong indication that the metrics are working as expected. If they are unable to explain certain anomalies this might indicate that the metrics are not working correctly. The employees in question are three employees working in a mixed research and development division of a large multinational operating in well over 100 countries.

The workshop was structured using a PowerPoint presentation, asking a number of questions. For example, the employees were shown a number of systems and asked to identify which systems were faster to work with, or they were shown a number of different systems with associated MTpP scores, and asked to explain differences between the different years. The conscious choice was made not to have a very structured workshop. This allowed the topic of discussion to follow the issues that jumped out to the developers, allowing this author to retrieve a lot of feedback on MTpP over a period of one hour. Data was collected by taking notes. These were converted in the text below, which was sent to the participants of the workshop for validation.

The first question asked to all three employees was why they care about the architecture quality of their system. A number of reasons were mentioned, among which that systems with a well defined architecture are easier to understand and more fun to work in. Additionally, it allows the system to remain evolvable as it increases in size and complexity and avoids the necessity for work to be done more than once. On top of this it allows for clearer division of labour and responsibility between teams while also making it easier to judge whether a certain task is indeed achievable within a reasonable timeframe. Finally, it makes it easier to perform maintenance tasks on a system. These answers are a clear indication that software architecture is considered an important aspect of the development process.

Next, the employees were asked whether the MTpP metric aligns with their way of working or whether they could think of processes that might be misinterpreted with the used definition of MTpP. They recognised that there were teams within the company that had a different way of working which relied more heavily on automated testing processes for quality control, allowing developers to push directly to a branch rather than first creating a separate feature branch. Within this team, their way of working aligned very well with the definition of MTpP.

The employees were also asked to rank three systems in order of how quickly changes were implemented. The reasoning of the employees was that one of the systems had a dedicated team of developers for it, while the others were only worked on by people with split responsibilities. It would make sense if this system was the fastest to develop in. No significant changes between the other two systems were mentioned. This was reflected in the MTpP metrics; The system with dedicated developers had a MTpP of 0.7 days while the other two had MTpPs of 1.7 and 2.0 days.

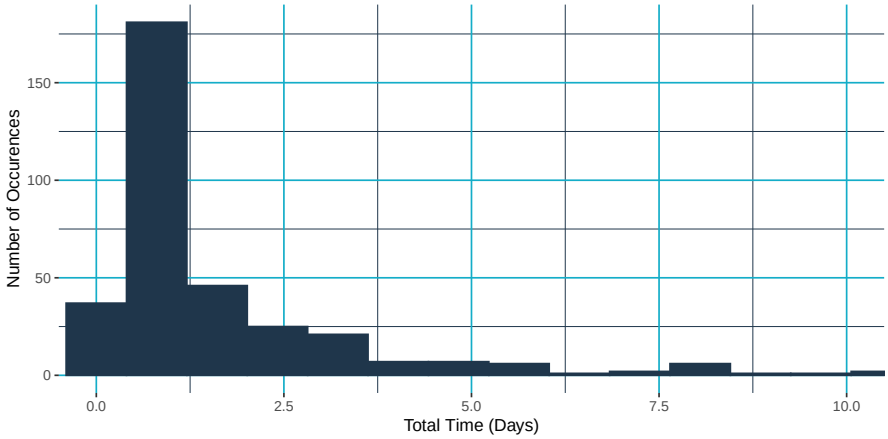


Figure 6.3: Histogram showing the number of occurrences of different values of MTpP.

Additionally, the employees were asked if they could think of a reason why the MTpP of specific system had significantly increased in a specific year. They were able to link this to the point when active development on the system had been completed, and the development that was taking place were minor bug fixes, which were on a lower priority compared to earlier development goals.

The employees were also asked some more general questions. Firstly, they were asked whether development was speeding up or slowing down. They explained that due to the mixed research and development responsibilities of their team, generally projects are developed and then handed over to a different team within the company. The systems included in this analysis are currently being handed over. This is also reflected in the data, with the number of pulls per year climbing to a maximum of 213 in 2019 and then decreasing to a very moderate 26 in 2021.

Finally, they were asked whether they felt that the architecture was generally improving or decreasing. To this they replied that generally they try to fix issues as they come up. Similarly they spend a lot of time within the team to reflect on what is and is not working. So generally they would say that architecture quality is rising. This is reflected in the data with the architecture score increasing from year to year with the exception of 2021. It should be noted that, because of the slower rate of development in 2021, there are far fewer datapoints in this time frame. Therefore, this particular result should probably be taken with a grain of salt. Disregarding this year, the data aligns very well with explanations of developers.

All in all, the experience of the employees in question aligned very well with the data collected in this project. Additionally, they were able to provide very reasonable explanations to all anomalous data within the dataset. This provides a high level of confidence that the MTpP metric is working as intended in providing a good indication of how quickly development is taking place. Additionally, the metric was easily understood by the employees in question, and they quickly presented possible alterations to the metric which might make it better suited for specific situations. From this it can be concluded that the metric is both an effective metric for SoD and can be easily understandable by people in the field.

6.4.2. Validation within the dataset

Besides the validation through workshops there are a number of relations which were included within the design criteria for the metric, have been established in previous research, or intuitively should be true. By checking whether these correlations match the ones that can be found in the data collected for this project. To gain some intuition of how the MTpP is distributed a histogram of the different values it takes can be found in figure 6.3. The different correlations which have been checked in this manner are listed below.

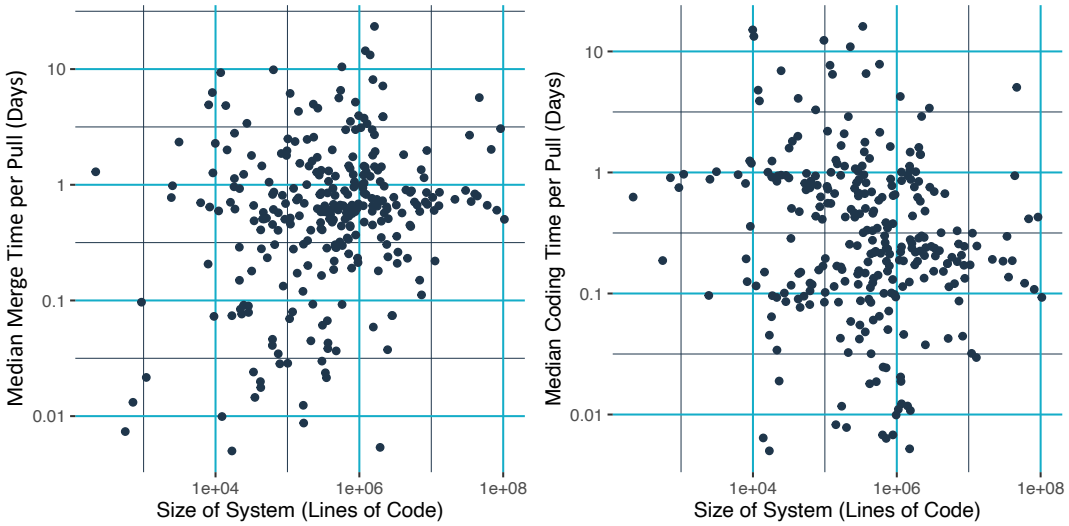


Figure 6.4: Scatterplot showing the MTpP split into merge time and coding time against its system size (in LOC) Note that both the x and y axes are on plotted exponentially.

6.4.2.1. Correlation with Number of Authors

It has been known for some time that paradoxically adding more authors to a project does not generally increase the productivity of the team and in some cases even decreases the productivity (Brooks, 1974; Dingsøyr & Moe, 2013). To investigate this, it is interesting to split MTpP into coding time and review time, as the expectation would be that larger teams allow for more efficient coding, but are less productive in the review aspect. Performing a Kendall rank correlation test shows that increasing the number of authors working on a system tends to decrease the coding time slightly, while it increases the merge time. The former has a p-value of 0.0015 with a τ of -0.119 while the latter has a p-value of 1.35×10^{-9} and a τ of 0.228. This results in a correlation for the full MTpP with a τ of 0.085 and a p-value of 0.0248. This is quite a strong indication that the metric is indeed working as intended.

6.4.2.2. Correlation with LOC

Another aspect that can be easily validated is that while the metric was chosen to not be based on the body of code, Almost everything in software engineering correlates at least somewhat with the LOC count of a system. One check that can be performed to show that MTpP might be failing in one of its design criteria is to check whether MTpP correlates very strongly, or not at all, with LOC count.

A visual representation of this correlation is shown in figure 6.4. Visual inspection seems to show that there is little to no correlation between the two metrics. This is confirmed by calculating the Kendall rank correlation coefficient for MTpP and AR. This evaluates to a τ of 0.048 with a p-value of 0.314, which is in line of expectations when no significant correlation exists.

6.4.2.3. Correlation with Number of Pulls

One additional method for falsifying the MTpP metric is to check how it correlates with the number of pulls per year. If the time per pull decreases strongly with the number of pulls this might indicate that work is simply being spread over a larger number of smaller branches instead of actually working faster. To check for this relation the two metrics are plotted in a scatterplot as shown in figure 6.5.

From this graph there seems to be no significant relation between the two metrics. Once again this can be quantified using a Kendall rank correlation. This results in a p-value of 0.0003 with a τ of -0.178. This indicates that there is a statistically significant relation between the number of pulls and the MTpP but it is quite weak. This is in line with expectations if the metric was working. After all, it is reasonable to expect that companies who are capable of finishing a pull quicker are able to perform more of them in a year. If the correlation was very extreme, it might indicate that when a project has twice as many

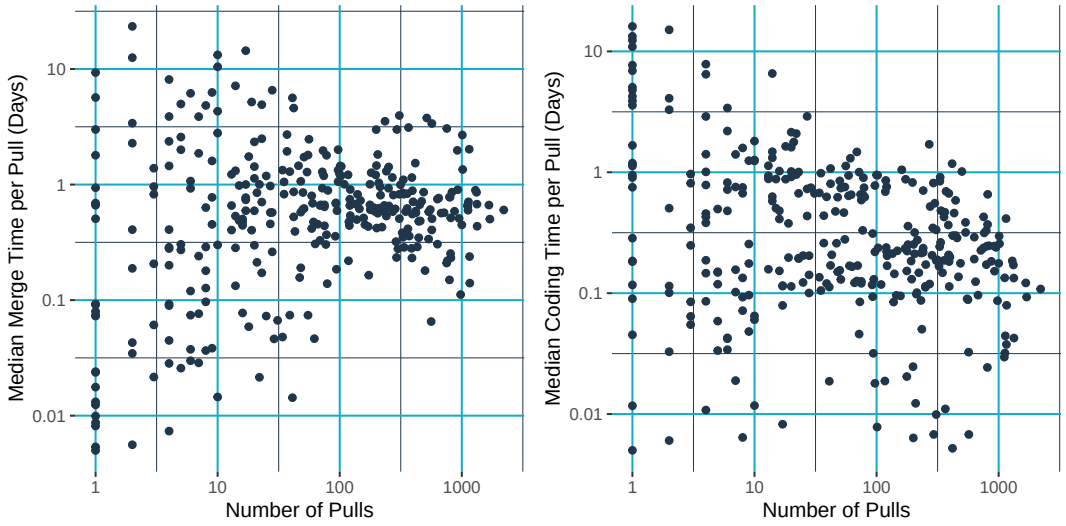


Figure 6.5: Scatterplot showing the relation between MTpP split into merge time and coding time and the number of pulls in a year. Note that both the x and y axis are plotted exponentially.

pulls with half the MTpP compared to a different project this might indicate that some projects simply dedicate a pull to a different amount of work. The fact that no strong correlation exists indicates that the amount of work for a single pull is at least somewhat consistent.

6.4.2.4. Difference between Open Source and Commercial Projects

Another aspect that might indicate that MTpP is not working as intended could be if there was a significant difference in coding time for commercial and open source projects. After all, the coding of a specific feature depends for a large part on the competency of the programmer, not so much the environment. Additionally, many open source projects, especially within the projects chosen for this project, are enterprise driven. Simultaneously, longer merge times indicate a larger communication overhead. This means that we can expect merge time to be longer for open source projects compared to commercial projects. Because open source projects tend to be used by large and diverse groups of people backward compatibility and conservation of features becomes more important. This could significantly increase merge times as many more usecases need to be tested. The findings confirm this; the median coding time for commercial and open source projects are 0.20 and 0.25 respectively, while the merge times are 0.09 and 0.74 days.

7

Research Results

The following chapter, outlining the findings of this project has been divided into two sections. Firstly, there is the numerical analysis which looks at the (lack of correlation) between MTpP and the AR in a number of different methods. This is followed by a section which discusses the more abstract implications of this analysis.

7.1. Data Analysis

A scatterplot of what the distribution of MTpP and AR looks like can be seen in figure 7.1. There are two points that immediately jump out when looking at this graph. Firstly, the AR score is defined to be a normal distribution between 0.5 and 5.5. Despite this the particular sample here scores much higher, without any data-point towards the low end of the spectrum. Why this might be the case is discussed further in section 8.5.3.2.

Additionally, this plot provides evidence that there is no significant correlation between MTpP and the ARs of a system. Once again a Kendall rank correlation can be used to quantify this. This provides a p-value of 0.802 for MTpP. This indicates that there is no statistically relevant correlation between the metrics. Similarly, the correlation between the coding time and AR is also not statistically relevant with a p-value of 0.334. Interestingly, there is a significant correlation between merge time and the AR, with a p-value of 0.015, but the τ is very small; 0.088.

Most other factors which might influence MTpP or the AR such as working method, culture, company size and so on, can be removed by considering only a single project at a time. A Kendall rank correlation test between MTpP and the AR was performed for each of the different systems in the dataset. This removes some data from the dataset as at least three snapshots are needed in order for the rank correlation to have any meaning. Almost exactly half of the remaining projects analysed in this manner has a positive correlation, while the other half has a negative correlation. This once again affirms that there is no strong correlation between ATD and SoD.

Another method of looking at the data which highlights extreme cases is by assigning risk profiles to pull lengths. Pulls with a total duration longer than 90% of the pulls in the dataset are considered high risk and are assigned a risk value of 4. Similarly, scores longer than 80%, 70% and 60% are assigned scores 3, 2 and 1 respectively. By aggregating these scores for each project and each year we can calculate the mean risk score for each project. The coding, merge and total time can then be correlated (again with a Kendall correlation test) with the AR, the results of which can be seen in table 7.1. Only the result for merge time is statistically significant which has a moderate correlation coefficient of -0.128, indicating that as architecture quality increases the chance of extremely long merge time becomes lower.

To gain a more insight into what might be happening at a more global level within this dataset, the AR of each project has been rounded to a 0.5 value. Then, each of these groups is plotted in a boxplot

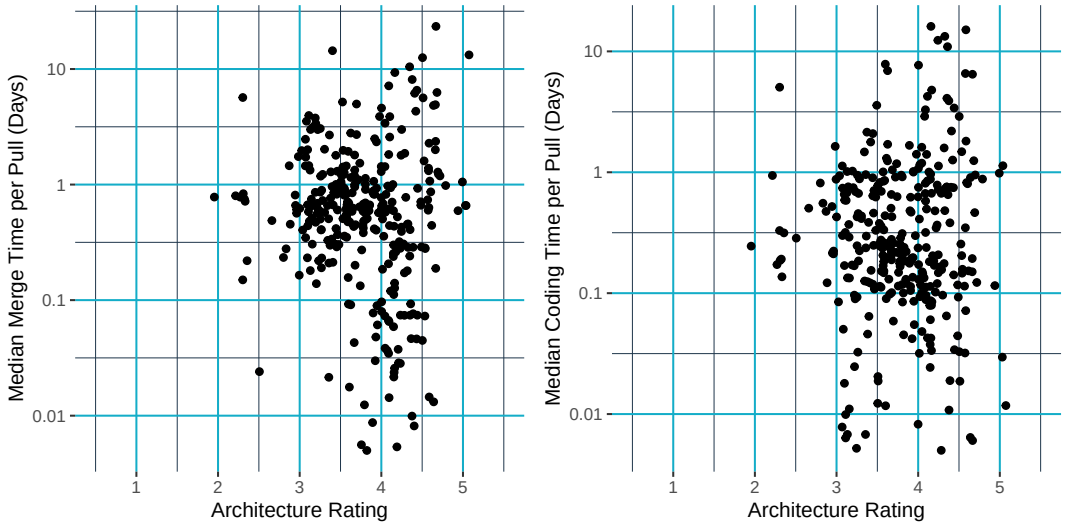


Figure 7.1: A scatterplot of the relation between MTpP split into merge time and coding time, and AR. Note that the y-axis is logarithmically scaled. Additionally, while AR is defined to be normal this is not the case within this sample.

Table 7.1: Kendall Correlation coefficients and corresponding p-value of the different aspects of MTpP against the AR.

	P-value	τ
Coding Time	0.0962	-0.061
Merge Time	0.0004	-0.128
Total Time	0.0578	-0.068

where the width of the box represents the number of datapoints in that box. The result of this can be found in figure 7.2. In this plot there once again does not appear to be a clear correlation. One thing that might be worth noting is that the coding time seems to climb slightly at very high architecture scores. Additionally, variance in merge time seems to be quite a bit larger than variance in coding time.

Finally, all the different sub-components of the AR are correlated with MTpP to see if perhaps something unexpected is happening in the aggregation of the different aspects into a single score, or whether there are particular sub-components which correlate more strongly with MTpP. The findings of this analysis do not significantly deviate from the earlier findings. Showing a range of correlations with a τ ranging between -0.15 and 0.15. From this it can be concluded that there is no strong correlation.

7.2. Interpretation

As discussed in section 6.4, there is strong evidence that both AR and MTpP are working as intended as metrics for ATD and SoD respectively. From the previous section it can therefore be concluded that there is no significant correlation between MTpP and ATD. Nevertheless, faster working is given as one of the important reasons of investing in improved architecture quality (Martini & Bosch, 2015). These results refute this. There are a number of possible explanations for this, such as other effects which are out of the scope of this project. These which will be discussed below.

In the previous section it was mentioned that merge time tended to increase slightly with increased ARs. This can perhaps be best explained by considering what implications a good architecture would have for the system as a whole. Maintaining a good architecture would require clear protocols on how to work on the code. Such protocols can ensure certain requirements are met, but can also slow down development as all features need to adhere to somewhat strict guidelines before they can be merged. Someone has to then check whether these guidelines are met, resulting in increased merge times, while not having much effect on coding time. It should be noted that, while this effect is statistically significant it is so small that it is questionable whether it worth considering.

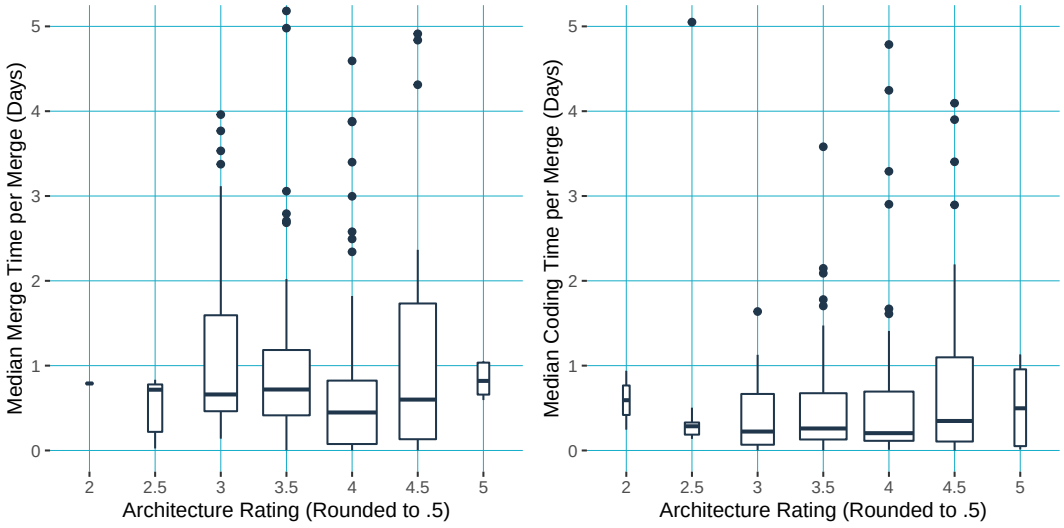


Figure 7.2: A boxplot of different categories of AR, against the MTpP split into merge time and coding time of the projects in said category. The width of the boxplot is scaled according to the number of observations in that category. Once again, the y-axis is plotted logarithmically.

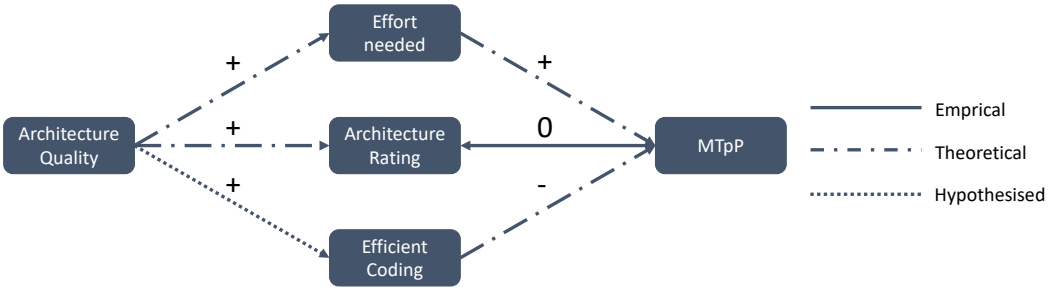


Figure 7.3: Relational Diagram of different metrics and concepts relating to ATD and SoD. A + next to one of the arrows indicates a positive correlation while a - represents a negative correlation and a 0 represents no significant correlation. The type of dash used to draw the arrow indicates how the relationship was established.

The fact that coding time increases slightly at higher ARs (see figure 7.2) could be a sign that at these higher scores investing in a better software architecture yields diminishing returns. At some point the benefits of better software architecture are outweighed by the additional work needed to maintain the architecture at such a high level. This might indicate that architecture quality has a larger effect on the review process and other aspects that would be included in the merge time than on actual coding time. In a similar vein as with the increase in merge time mentioned in the previous paragraph, the effect is very small, so one can wonder whether this is important enough to mention.

7.2.1. Relational Diagram

To show where this unintuitive results might come from, a relational diagram of the different metrics and concepts was constructed, which can be seen in figure 7.3. In the following sections, the theoretical and hypothesised correlations will be discussed. By definition when the effort needed increases this must also increase the coding and merge time. After all, tasks requiring little effort never take a lot of time. Architecture quality must also have a positive correlation with effort. After all, a high architecture quality means that the project adheres to a number of architectural rules. It must always be more difficult to adhere to many rules than to adhere to no rules. Because of this, maintaining good architecture quality must also require more effort.

Regarding the relation between the architecture quality and the AR, The metric is built upon standards that are widely accepted within the industry such as that ”copy pasting” code is a bad practice

(Li et al., 2006). A more extensive explanation of these metrics and why they are generally considered bad practices was given in appendix B. Because these metrics are directly based upon the source code and compared across many different systems and only rank correlation is used to establish a (lack of a) correlation there is little room for the metric to fail. Finally, increasing the efficiency of coding by definition decreases the time needed to do said coding. After all, efficiency is a measure of how much work is done using a specific amount of a resource (like time).

Having established these theoretical connections, it becomes visible in the relational diagram that, to explain the lack of a correlation between the AR and MTpP, there must be some other aspect that has a dampening effect. The most probable reason for this might be that working in an environment with clearly defined architecture is easier to work in, increasing the efficiency. This would also explain why many managers do feel that work proceeds faster when working in a well defined architecture despite pulls not completing faster. After all, more work is being performed (adhering to strong architecture standards) in the same amount of time.

7.2.2. Investment vs Maintaining

It is worth noting the difference between investing in a strong architecture versus maintaining one that is already in place. The analysis performed in this project concerns the effect of an architecture that is already in place, and shows that this has no strong correlation with MTpP exists. On the other hand, there is research which suggests that these investments can be quite significant, especially if carried out later in the projects lifecycle (Nord et al., 2012).

This might results in situations where it is not worthwhile to invest in improving software architecture directly, especially if certain security and reliability requirements have already been met. In these cases it might be worth investigating a boyscouts approach of ”leaving the area cleaner than you found it”. Encouraging developers to clean up the architecture small step by small step as they are working on other features. This could slowly increase the AR without incurring large investments to do so.

8

Conclusions

This chapter presents and discusses the conclusions of the project. First the main research question will be answered, after which a more in depth answer to each of the sub-questions will be given. This will be followed by a discussion of the business implications of these results, a number of possible threats to validity and finally the scientific implications and possible avenues for future research are discussed.

Architectural technical debt is already widely used and found to be useful in management of architecture quality. However, there is little evidence about how architectural technical debt actually influences the coding process. This makes it difficult to quantify whether investments in architecture quality are warranted and to justify the (possible) need for such investments. To help fill this gap, the following research question was stated: ”What is the effect of architectural technical debt on software development speed?” To answer this question a new metric was developed based on the goal question metric methodology to measure software development speed. This metric has potential to be used in many other contexts to measure the speed of development of a project. By combining this new metric with the existing architecture quality metric developed by SIG an empirical analysis was performed on a large number of open source and commercial software systems. Based on the results from chapter 7, the answer to this question is that no correlation between architectural technical debt and speed of development could be found. Therefore architectural technical debt has no directly measurable effect on speed of development.

This does not mean that there is no interaction between speed of development and architectural technical debt. Software architecture has been a hot topic for many years and many experts agree that it is essential to adhere to strong architectural principles. As was discussed in section 7.2 it is likely that coding efficiency is in fact increasing with improved software architecture, but this is counteracted by other effects like increased coding efficiency.

8.1. SoD Quantification (RQ 1)

The first sub-question posed in the introduction was: ”What are measures of speed of development that allow for comparison between different projects?” Based on the findings in section 2.2 it was judged that there is no appropriate metric for speed of development currently in use within the scientific community. A large number of metrics that were in use were generally based upon the software product. This means that these metrics are inherently linked to the architecture rating (which is part of the product). A number of investigated metrics were based on the process of coding, but these lacked in broad applicability, automatability or comparability across projects. Because of this the choice was made to develop a new metric to better suit the needs of this project.

For this project it was important that the metric to be used was based on the process of writing code rather than the actual product, as the latter can be strongly influenced by the architecture quality.

Additionally, to allow for the larger scale empirical analysis the metric should be automatable and easily applicable to a large number of systems. There was one metric in use which checks all of these boxes, namely issue resolution time. This metric does have the major downside that it includes the time during which an issue is identified but not actively worked on. Nevertheless the core concept of this metric was taken to form the basis of the new metric of speed of development.

Instead of relying on issues, a new metric was developed in this project which is built upon GIT histories. This was done according to the goal question metric methodology. Because GIT directly reflects the development history of a project this ensures that the metric more accurately reflects time where active development is taking place. A new metric was created based on this GIT history, which represents the median length of time branches exists for in a project. The metric is named Median Time per Pull. Additionally, the metric allows for the distinction between coding time and merge time, giving insights into how much time is spent on actual coding and how much time is spent waiting on the merge to be performed. One flaw in this metric is that some coding already takes place before the first commit. This time is not included in the Median Time per Pull.

To ensure that this new metric was working as intended, a number of validation steps were taken in section 6.4. Most importantly a workshop with one of the customers of SIG took place where employees were asked to predict results found by the tool as well as attempt to explain certain anomalous results within the data. They were able to correctly identify which systems had developed faster compared to others, and were able to assign specific events to certain deviations in the data. This generates significant confidence that the median time per pull metric is indeed working as intended.

In addition to this workshop, a number of numerical analyses were performed to validate the metric. The metric behaved as expected (based on previous research) when looking for correlations between median time per pull and the number of authors, the size of the system, and the number of pulls for a project in a year. This provides additional confidence that the metric is working as intended.

This project provides the scientific community with an easy to implement and high granularity tool for analysing speed of development in large numbers of projects. The tool has been validated using a number of different methods. Additionally, the only requirements for the systems of interest are that the commit history is available and that the commit history has not been forced into a linear form. To answer the research question posed in the introduction: No broadly applicable tool for comparing speed of development across different projects existed, but such a metric has been developed as part of this project.

While this metric has been specifically designed for this project the metric shows promise for much broader applications. The metric requires little of the project being analysed to work, allowing for very broad applications. Additionally, it can easily be applied to a large number of projects, so long as the GIT histories are available. Median time per pull could be used to analyse a large number of relationships in software development that are currently assumed to exist, but not actually empirically shown. For example, whether other aspects of code quality significantly affect speed of development.

8.2. ATD quantification

A systematic literature review was performed to gain a thorough understanding of the different methods for assessing architectural technical debt and its costs. This review found a number of different methods for assessing architectural technical debt. While a number of them were quantitative these often resolved around determining the monetary principal of the architectural technical debt. This is not easily usable for comparison across many different projects, as depending on the size of the project what constitutes large and small amounts of architectural technical debt can vary widely.

ac SIGs architecture quality metric is based in long established best practices and ranks systems on a one to five star scale based on where they fall in an established benchmark. This makes the metric especially useful for the comparative analysis in this project. To answer the second sub-question: There is no metric in use in the scientific community well tailored to these kinds of comparisons, but SIGs metric can be used rather effectively.

8.3. Correlation between SoD and ATD (RQ 2)

The third sub-question asks: ”What kind of relation exists between the amount of architectural technical debt in a project and its speed of development?” Based on the findings in chapter 7 the conclusion was drawn that no significant correlation between the two measures exists. Multiple different methods of analysing the data were used to verify this un-intuitive result, all of which resulted in the same conclusion.

Increased development velocity has been mentioned as one of the reasons to improve architectural technical debt by experts. It seems unlikely that the effect that these experts experience simply does not exist. It is more likely (as explored in section 7.1) that the effect is dampened by some other factor. A likely explanation for this is that while developer efficiency does increase with improved architectural technical debt so do the requirements on the developed code, dampening the effect and explaining why it is not measured here.

To answer the research question; There is no measurable correlation between architectural technical debt and speed of development. Nevertheless, there might very well be many other factors that are influenced by architectural technical debt, meaning that the importance of architectural technical debt management cannot be disregarded outright. Instead more research is needed to measure the other tacit effects of architectural technical debt on the software development process or the quality of the product.

8.4. Business Implications

The following paragraphs discusses the business implications of the lack of a correlation between architectural technical debt and speed of development for each of the aspects of a business presented in the balanced scorecard methodology. More generally however, for most companies the manner in which they handle their architectural technical debt likely will not change much. If architectural technical debt is repaid with the sole purpose of increasing speed of development however the findings from this project indicate that this is not effective. By combining the context given in chapter 4 with the findings from the previous chapter, conclusions on how businesses should incorporate the findings in more specific aspects of business can be formed.

The second significant contribution this project delivers is the creation of the new median time per pull metric. While the metric has been designed specifically for this project it conceptually closely matches the workflow of actual software development. Because of this it could prove useful in management of development teams as well.

8.4.1. Internal Business Processes

Median time per pull provides interesting insights into the software development process. Being able to investigate whether a lot of time is being spent on coding or merging, or whether development is speeding up or slowing down can prove extremely valuable in managing development teams. Median time per Pull could provide such insights, but as was mentioned before it is important not to rely on median time per pull as a KPI. Like most single performance metrics in software engineering median time per pull can quite easily be gamed. In this case it could be as easy as splitting up tasks into more distinct pulls. This would decrease the median time per pull and increase the number of pulls, which would indicate that development is speeding up and more work is being done. In actuality, however, no additional work would be done, and it might even cause a decrease in quality and the amount of work being done. Because of this reason the metric should likely only be used sporadically and with great care. If the median time per pull of a specific department is found to be particularly low or high this should be a reason for conversation to determine why this is happening and whether further action is needed.

One more specific area where median time per pull and AR combined could provide meaningful insights is when determining what areas of the software development process require additional attention. If one of the two metrics is found to be adequate but the other is not, there is clear focus on what needs to improve. If both are not considered good enough, a strategic choice needs to be made (as

they don't improve together) on which of the two aspects require more immediate attention.

This research indicates that improving software architecture to increase development velocity is not a valid strategy. On the other hand, improved software architecture also does not decrease speed of development. What this means is that the other benefits of improved software architecture don't have to come at the cost of reduced speed of development. It is important to make a distinction between maintaining a certain architecture level versus achieving said level from poorer architecture. This project shows that the former of these does not cost significant investment, while the latter can definitely cost significant investments. Because of this it might be best to adopt a ”boyscouts attitude” to software architecture where you always leave it ”cleaner than you found it”. This can allow for slow but steady increases in software architecture quality without incurring large investment costs.

8.4.2. Financial

Median time per pull could provide meaningful insights into the financial decision making process of companies. Cost estimation in software engineering is rather difficult and it is not uncommon for software projects to significantly overshoot their budgets. Being able to assess how long it takes to complete a pull also allows managers to put a price tag on that pull. If development teams are able to accurately estimate the number of pulls needed to implement a feature or fix a bug median time per pull would allow that team to predict the cost of that feature or bug based on how quickly the team has historically worked. Estimating the number of needed pulls should be much easier than directly estimating the cost or time needed to implement a feature. Especially since modern software development methods already use sprints to distribute work, where large development goals are split into multiple smaller sub-tasks.

The lack of a relationship between architectural technical debt and speed of development does not have a large effect on the income part of financial performance. After all, architecture and development speed are not noticed by ICT with service customers. CSD customers do notice these backend considerations as they have to work with the actual code. The findings of the previous section indicate that software architecture might be less important than previously thought. As a result it could be CSD customers would care less about software architecture. It is important, however, to not forget the other effects architectural technical debt could have on software (development).

The findings of this report have more significance regarding the costs of software development. It can be rather difficult to efficiently scale software projects and their development teams. Managers may have been tempted to improve their software architecture hoping that this would increase their development speed. The results from this project show that this strategy does not work. Instead, some cost reductions could be achieved by placing less strict requirements on software architecture. As mentioned before it should be noted, that this could have serious adverse effects on other aspects of software quality such as developer motivation and reliability.

8.4.3. Customer

If a software company (especially a customer software development company) is aware of their median time per pull they could use this in communication with their customers. It would be interesting to be able to promise customers an average implementation time for new features or time to fix a bug based on the median time per pull of the development team in question. Naturally, communication with customers should avoid technical terms like median time per pull and instead focus on the implications for customers. Nevertheless, the metric could provide a solid basis to build such marketing materials on.

As mentioned in the previous section there is no reason ICT with service customers would care directly about the amount of architectural technical debt in their product, nor about the speed of development of the team that developed it. Naturally, there are aspects influenced by architectural technical debt of speed of development such as security which are definitely important. But the customer in these cases does not directly care about architectural technical debt or speed of development. For CSD customers the findings of this report could have more of an impact. As the results show that speed of development does not improve with architectural technical debt it might be the case that the

architecture is only relevant insofar as the customer needs to continue working on the system after it has been handed over, or whether certain security and reliability requirements need to be met. Very good communication with the customer throughout the project about their requirements is needed to best ascertain whether a strong architecture is required or not.

8.4.4. Learning & Growth

The results from this project seem to indicate that there is little reason to invest in software architecture to ensure that a software package can remain easy to work in. This is not entirely the case, while the results show that architectural technical debt and speed of development are not correlated with one another there are other aspects of evolvability which are affected by the amount of architectural technical debt in a system. Mainly understandability and evolvability of software. The former of these allows companies to stay able to work with their software even if developers switch teams or employers. The latter allows the owner of software to easily adapt to new challenges and opportunities that arise and were not per se thought of when the software was first developed.

8.5. Threats to Validity

To properly assess the validity of this project four different aspects of validity which are often used in empirical software engineering research (Feldt & Magazinius, 2010) are considered. These aspects are: conclusion validity, internal validity, construct validity and external validity.

8.5.1. Conclusion Validity

Conclusion validity is mainly concerned with whether the conclusions reached are statistically valid. to give an indication of statistical relevance the p-value is given for every correlation analysed within the report. A result is only considered statistically relevant if the p-value is 0.05 or lower. This means that the chance that the particular result, or a less likely result would occur is 5% or less. As this threshold is considered standard practice within many different areas of research this should not cause any issues for the conclusion validity of the project.

8.5.2. Internal Threats to Validity

Internal threats to validity are possible causes of the observed behaviour which are not included in the research. Generally these are metrics or aspects which are not measured or beyond the control of the project, but could still influence the outcomes. A number of such aspects have already been discussed in section 1.3 This project has two main strategies for dealing with this. Firstly, it relies on the law of large numbers (Dekking et al., 2005) to reduce variance between individual measurements. This is why 120 software projects of various sizes and developers were used in the analysis.

Additionally, the analysis was performed on individual systems with more consistent teams and ways of working. This should reduce the variance of other metrics drastically, allowing for more clean analysis of the exclusive correlation between architectural technical debt and speed of development. The fact that these results don't disagree with those found in the entire dataset indicates that these other factors don't cause enormous amounts of interference.

8.5.3. Construct Validity

Construct validity concerns itself with whether the constructs used in the research actually represent the real-world point of interest. In the case of this project the main constructs used are AR and median time per pull. It should be noted that a number of different validation steps have been taken to ensure that both median time per pull and AR are working as intended. Nevertheless, there are a few threats to validity worth considering.

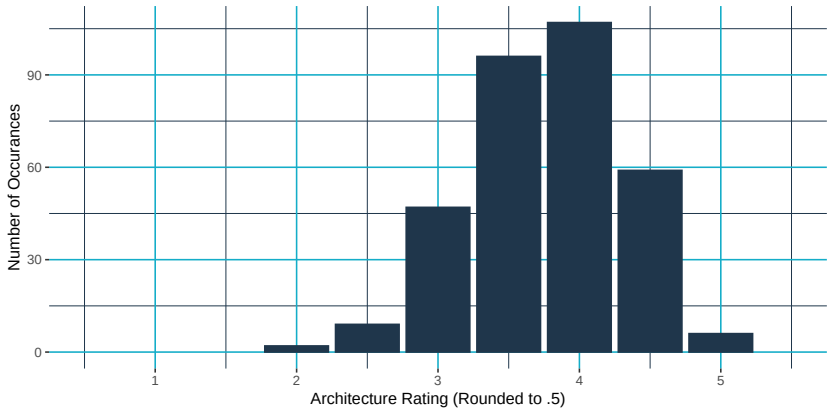


Figure 8.1: Histogram showing the Number of occurrences of specific ARs.

8.5.3.1. Feature Size

During this project, a pull is taken as a constant unit of work. This does not necessarily have to be the case. Nevertheless, there are a number of reasons why in most cases the amount of work should be roughly comparable. Firstly, the core benefit of using branches is to make sure that developers don't interfere with one another's work while programming. As a result branches are almost exclusively used by single, or sometimes pairs of programmers. Because this is the case there is a soft upper bound on the amount of work that can be performed within a single branch. After all, there is only so much a human brain can consider at a time. Additionally, with the dominance of sprints in software development, features also have to be developed in a relatively short amount of time, further capping the amount of work that can be done in a single branch. This was further confirmed by the median median time per pull over all pulls, which is very close to one day. Additionally, as was discussed in section 6.4.2.3 there is no strong correlation between median time per pull and the number of pulls, indicating that a pull does represent a somewhat consistent unit of productivity.

8.5.3.2. Skewed Architecture Quality Data

Normally, the scores provided by SIG should be normally distributed with a mean at three stars, 70% of cases between two and four stars and 90% between one and five stars. When plotting a histogram of the different ARs it was found these results don't adhere two this pattern. This can be seen clearly in figure 8.1, the highest peak is much further to the right than it should be. In fact, there are zero cases of one star, 2.6% is two stars, 26.1% three stars, 60.6% has four stars and 10.7% have five stars.

There are two explanations for this. Firstly, the architecture score is normalized across its sub-components not over the score as a whole. Because it is very unlikely that a system scores very badly in all different metrics, there is a ”regression to the mean” (Barnett et al., 2005) effect which causes outliers to be less common than they should otherwise be.

Additionally, the AR is currently benchmarked against a relatively small (±60) number of systems. As a result it is not unlikely that there is a deviation from what can be considered common behaviour in either the benchmark, or the dataset used for this analysis.

Nevertheless, the benchmark is only used to format the data into a easily digestible format. The relative score of different systems does not rely on the benchmark at all. This is also why the Kendall rank correlation is very useful in this particular usecase, as it compares the relative position of datapoints rather than their absolute score. Specifically because of this, the skewed nature of the data does not represent a threat to the findings of this project.

8.5.3.3. Merge Time

It should be noted that merge time as defined in section 5.3 is not a perfect representation of review time. After all, if a branch is reviewed, and some changes need to be made, these commits are considered

coding commits, despite the fact that the review process has already begun. Nevertheless, it should provide a good indicator for how long it takes other people within the organisation to look at and verify code written by others.

8.5.3.4. Coding before the first commit

As was already mentioned before in section 5.3, the work which takes place before the first commit is not included in median time per pull. In some cases the majority of work could take place in this first commit, and so a lot of work would be out of view. During this project it was investigated, whether including pulls where only a single coding commit existed (with a coding time of zero) significantly altered the results, and this was not the case. Because of this, there is no reason to believe that the exclusion of these pulls forms a significant problem.

8.5.4. External Threats to Validity

External threats to validity are those that question whether the context of a project is representative of reality. This project makes use of a large number of projects for its analysis. This large dataset makes it unlikely that this sub-set is statistically different from what is common in reality.

Many of these project are open source. One can wonder whether these systems are representative of the software development that takes place in a commercial context. As was explained in section 6 the majority of projects were chosen with constant and dedicated development teams to minimise this effect. Nevertheless there are differences between the open source and commercial dataset, as discussed in section 6.4.2.4. When the correlation coefficient between AR and median time per pull is calculated for the two subsets the results don't disagree with one another. For the commercial projects the p-value is 0.0095 and the τ is 0.176 while for open source projects the p-value is 0.0126 and the τ is 0.107. Which indicates that for this project there is no threat to the results because open source projects were used.

8.6. Scientific Contributions & Future Research

In the following section the contributions this project makes to the scientific community are discussed. Additionally, this project leaves a number of avenues for additional research. These will be briefly discussed as well.

The major scientific contributions of this project are twofold. Firstly, a new metric allowing for the convenient and large scale analysis of development speed in software projects. This metric did not exist beforehand. Secondly, the first large scale empirical research looking into the relation between architectural technical debt and speed of development was performed in this project.

The results of this second contribution show that the impact of architectural technical debt on the software development process are less strong than previously thought. Despite this there is a lot of previous literature as well as many expert opinions talking about the importance of architecture quality, showing that there is a lot of interest in these subjects. This project shows that it is important to empirically validate these effects as they might not be as strong as previously thought.

Three main benefits of architectural technical debt repayment have been mentioned in this report, namely: improved security, improved reliability and improved developer motivation. To the knowledge of this author there exists no empirical evidence to support the claim that these correlations actually exist. It would be worthwhile to perform research similar to this project to validate whether these actually exist. After all, this project provides evidence refuting the reigning consensus that improved software architecture leads to improved speed of development. It is not unthinkable that similarly the other benefits of software architecture are not as strong as previously thought. Creating a coherent framework of the different benefits that do and do not come with improved architecture would be an extremely valuable addition to the scientific knowledge regarding software architecture. This project provides a first step by exploring the relationship between architectural technical debt and speed of development but much more research is needed establish which other relationships exist as well as how those different relationships interact. Only by establishing such a framework can it truly become possible to establish

the value of improved software architecture.

Additionally, speed is often the name of the game in software development. Nevertheless, there are little methods available to managers to actually speed up development. It could prove worthwhile to look at software projects in this project that develop exceedingly quickly, and attempt to find what (if anything) makes these projects special.

There are many other factors which could influence median time per pull besides the AR. For example the maintainability of the code. It could prove worthwhile to see how these factors impact median time per pull in an attempt to compensate for these factors, allowing for a cleaner analysis of the relation between speed of development and architectural technical debt.

Finally, after starting this project a new paper has been made public as a preprint (Tornhill & Borg, 2022). In this project the researchers use data from the Jira issue management system in combination with GIT commits with the aim of getting the best of both worlds with regards to median time per pull and issue resolution time. This allows Tornhill and Borg to better approximate the point where actual work begins, even if this work does not take the form of coding, or is not immediately committed. It is worth investigating whether this same method can also be applied to the research in this project and whether this has significant effects on the results.

Bibliography

Abran, A. (2019). Software development velocity with cosmic function points. *The Common Software Measurement International Consortium*.

Adobe. (2022). *About adobe*. Retrieved June 16, 2022, from https://www.adobe.com/about-adobe.html

Aggarwal, K., Mijwil, M. M., Al-Mistarehi, A.-H., Alomari, S., Gök, M., Alaabdin, A. M. Z., Abdulrhman, S. H., et al. (2022). Has the future started? the current growth of artificial intelligence, machine learning, and deep learning. *Iraqi Journal for Computer Science and Mathematics*, *3*(1), 115–123.

Albrecht, A. J. (1979). Measuring application development productivity. *Proc. Joint Share, Guide, and IBM Application Development Symposium, 1979*.

Albrecht, A. J., & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE transactions on software engineering*, (6), 639–648.

Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 44–51.

Andreessen, M. (2011). Why software is eating the world. *Wall Street Journal*, *20*(2011), C2.

Audi AG. (2022). *Company*. Retrieved June 1, 2022, from https://www.audi.com/en/company.html

Baik, J., Boehm, B., & Steece, B. M. (2002). Disaggregating and calibrating the case tool variable in cocomo ii. *IEEE Transactions on Software Engineering*, *28*(11), 1009–1022.

Barnett, A. G., Van Der Pols, J. C., & Dobson, A. J. (2005). Regression to the mean: What it is and how to deal with it. *International journal of epidemiology*, *34*(1), 215–220.

Basili, V. R., & Weiss, D. M. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions on software engineering*, (6), 728–738.

Besker, T., Martini, A., & Bosch, J. (2017). Impact of architectural technical debt on daily software development work—a survey of software practitioners. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 278–287.

Bieker, T. et al. (2003). Sustainability management with the balanced scorecard. *Proceedings of 5th international summer academy on technology studies*, 17–34.

Bijlsma, D., Ferreira, M. A., Luijten, B., & Visser, J. (2012). Faster issue resolution with higher technical quality of software. *Software quality journal*, *20*(2), 265–285.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., & Selby, R. (1995). Cost models for future software life cycle processes: Cocomo 2.0. *Annals of software engineering*, *1*(1), 57–94.

Brooks, F. P. (1974). The mythical man-month. *Datamation*, *20*(12), 44–52.

Caldiera, V. R. B. G., & Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of software engineering*, 528–532.

CAST. (2022). *Function point counting delivers reliable metrics for improving application analysis*. Retrieved February 15, 2022, from https://www.castsoftware.com/glossary/function-point-counting

Coelho, E., & Basu, A. (2012). Effort estimation in agile software development using story points. *International Journal of Applied Information Systems (IJAIS)*, *3*(7).

Cunningham, W. (1992). The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30.

De Toledo, S. S., Martini, A., & Sjøberg, D. I. (2021). Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, *177*, 110968.

Dekking, F. M., Kraaikamp, C., Lopuhaä, H. P., & Meester, L. E. (2005). *A modern introduction to probability and statistics: Understanding why and how* (Vol. 488). Springer.

de Toledo, S. S., Martini, A., Sjøberg, D. I., Przybyszewska, A., & Frandsen, J. S. (2021). Reducing incidents in microservices by repaying architectural technical debt. *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 196–205.

Diaz-Pace, J. A., Tommasel, A., Pigazzini, I., & Fontana, F. A. (2020). Sen4smells: A tool for ranking sensitive smells for an architecture debt index. *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*, 1–7.

Dingsøyr, T., & Moe, N. B. (2013). Research challenges in large-scale agile software development. *ACM SIGSOFT Software Engineering Notes*, *38*(5), 38–39.

Dorfman, R. (1979). A formula for the gini coefficient. *The review of economics and statistics*, 146–149.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., & Laplante, P. (2016). Cyclomatic complexity. *IEEE software*, *33*(6), 27–29.

Eliasson, U., Martini, A., Kaufmann, R., & Odeh, S. (2015). Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: A case study. *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 33–40.

Fawareh, H. (2020). Software quality model for maintenance software purposes. *Int. J. Eng. Res. Technol*, *13*(1), 158–162.

Feldt, R., & Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey. *Seke*, 374–379.

Fenton, N. E., & Neil, M. (1999). Software metrics: Successes, failures and new directions. *Journal of Systems and Software*, *47*(2-3), 149–157.

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The space of developer productivity: There's more to it than you think. *Queue*, *19*(1), 20–48.

Fox News. (2018). *Google ceo sundar pichai testifies on capitol hill [video]*. Retrieved June 16, 2022, from https://www.youtube.com/watch?v=prdxra7B5H4&t=11202s

GitHub. (2022). *Github rest api*. Retrieved April 7, 2022, from https://docs.github.com/en/rest

Graylin, J., Hale, J. E., Smith, R. K., David, H., Kraft, N. A., Charles, W., et al. (2009). Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, *2*(03), 137.

Hill, P. et al. (2010). *Practical software project estimation*. Tata McGraw-Hill Education.

Hohmann, L. (2003). *Beyond software architecture: Creating and sustaining winning solutions*. Addison-Wesley Professional.

Honglei, T., Wei, S., & Yanan, Z. (2009). The research on software metrics and software complexity metrics. *2009 International Forum on Computer Science-Technology and Applications*, *1*, 131–136.

Intellect Software Development Company US. (2022). *About intellectsoft software development company us*. Retrieved July 8, 2022, from https://www.intellectsoft.net/about

Janka, M., Heinicke, X., & Guenther, T. W. (2020). Beyond the "good" and "evil" of stability values in organizational culture for managerial innovation: The crucial role of management controls. *Review of Managerial Science*, *14*(6), 1363–1404.

Kaplan, R. S. (2009). Conceptual foundations of the balanced scorecard. *Handbooks of management accounting research*, *3*, 1253–1269.

Kaplan, R. S., & Norton, D. P. (1992). The balance scorecard-measures that drive performance. *Harvard business review*, *70*(1), 71–79.

Kaplan, R. S., & Norton, D. P. (2007). Balanced scorecard. *Das summa summarum des management* (pp. 137–148). Springer.

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., & Shapochka, A. (2015). A case study in locating the architectural roots of technical debt. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *2*, 179–188.

Kemerer, C. F. (1987). An empirical validation of software cost estimation models. *Communications of the ACM*, *30*(5), 416–429.

Khatibi, B. V., & Dorosti, M. (2016). An improved cocomo based model to estimate the effort of software projects.

Ko, A. J. (2019). Why we should not measure productivity. *Rethinking productivity in software engineering* (pp. 21–26). Springer.

Kruchten, P., Nord, R. L., Ozkaya, I., & Falessi, D. (2013). Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, *38*(5), 51–54.

Lamma, E., Mello, P., & Riguzzi, F. (2004). A system for measuring function points from an er–dfd specification. *The Computer Journal*, *47*(3), 358–372.

Lenarduzzi, V., Lomio, F., Saarimäki, N., & Taibi, D. (2020). Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software*, 110710.

Li, Z., Liang, P., & Avgeriou, P. (2015). Architectural technical debt identification based on architecture decisions and change scenarios. *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 65–74.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., & Ampatzoglou, A. (2014). An empirical investigation of modularity metrics for indicating architectural technical debt. *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, 119–128.

Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, *32*(3), 176–192.

Lucas Jr, H. C., & Goh, J. M. (2009). Disruptive technology: How kodak missed the digital photography revolution. *The Journal of Strategic Information Systems*, *18*(1), 46–55.

Mäntylä, M. V., Khomh, F., Adams, B., Engström, E., & Petersen, K. (2013). On rapid releases and software testing. *2013 IEEE International Conference on Software Maintenance*, 20–29.

Mao, Y., Lai, Y., Luo, Y., Liu, S., Du, Y., Zhou, J., Ma, J., Bonaiuto, F., & Bonaiuto, M. (2020). Apple or huawei: Understanding flow, brand image, brand identity, brand personality and purchase intention of smartphone. *Sustainability*, *12*(8), 3391.

Martini, A., & Bosch, J. (2015). Towards prioritizing architecture technical debt: Information needs of architects and product owners. *2015 41St euromicro conference on software engineering and advanced applications*, 422–429.

Martini, A., & Bosch, J. (2016). An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 31–40.

Martini, A., Bosch, J., & Chaudron, M. (2015). Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, *67*, 237–253.

Martini, A., Sikander, E., & Madlani, N. (2018). A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, *93*, 264–279.

Mavroeidis, V., Vishi, K., Zych, M. D., & Jøsang, A. (2018). The impact of quantum computing on present cryptography. *arXiv preprint arXiv:1804.00200*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308–320.

Microsoft. (2022). *About microsoft*. Retrieved June 16, 2022, from https://www.microsoft.com/en-us/about

Mo, R., Snipes, W., Cai, Y., Ramaswamy, S., Kazman, R., & Naedele, M. (2018). Experiences applying automated architecture analysis tool suites. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 779–789.

Moreira, M. E. (2013). Working with story points, velocity, and burndowns. *Being agile* (pp. 187–194). Springer.

Nayebi, M., Cai, Y., Kazman, R., Ruhe, G., Feng, Q., Carlson, C., & Chew, F. (2019). A longitudinal study of identifying and paying down architecture debt. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 171–180.

Nord, R. L., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012). In search of a metric for managing architectural technical debt. *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 91–100.

OpenXcell. (2022). *About openxcell*. Retrieved July 8, 2022, from https://www.openxcell.com/company/about-us/

Oracle. (2022). *About oracle | company information oracle*. Retrieved June 16, 2022, from https://www.oracle.com/corporate/

Oxagile. (2022). *About oxagile*. Retrieved July 8, 2022, from https://www.oxagile.com/company/

Patel, A., Panchal, D., & Shah, M. (2015). Towards improving automated evaluation of java program. *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*, 489–496.

Perens, B. et al. (1999). The open source definition. *Open sources: voices from the open source revolution*, *1*, 171–188.

Pérez, B., Castellanos, C., Correal, D., Rios, N., Freire, S., Spínola, R., Seaman, C., & Izurieta, C. (2021). Technical debt payment and prevention through the lenses of software architects. *Information and Software Technology*, *140*, 106692.

Prechelt, L. (2019). The mythical 10x programmer. *Rethinking productivity in software engineering* (pp. 3–11). Springer.

R Foundation. (2022). *The r project for statistical computing*. Retrieved July 7, 2022, from https://www.r-project.org

Shah, J., & Kama, N. (2018). Extending function point analysis effort estimation method for software development phase. *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, 77–81.

Sheetz, S. D., Henderson, D., & Wallace, L. (2009). Understanding developer and manager perceptions of function points and source lines of code. *Journal of Systems and Software*, *82*(9), 1540–1549.

Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, *3*(2), 30–36.

Soubra, H., Abran, A., Stern, S., & Ramdan-Cherif, A. (2011). Design of a functional size measurement procedure for real-time embedded software requirements expressed using the simulink model. *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, 76–85.

Spadini, D., Aniche, M., & Bacchelli, A. (2018). Pydriller: Python framework for mining software repositories. *The 26th acm joint european software engineering conference and symposium on the foundations of software engineering (esec/fse)*. https://doi.org/10.1145/3236024.3264598

Tornhill, A., & Borg, M. (2022). Code red: The business impact of code quality–a quantitative study of 39 proprietary production codebases. *arXiv preprint arXiv:2203.04374*.

Van Grembergen, W., & De Haes, S. (2005). Measuring and improving it governance through the balanced scorecard. *Information systems control Journal*, *2*(1), 35–42.

Van Solingen, R., Basili, V., Caldiera, G., & Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of software engineering*.

Verdecchia, R., Kruchten, P., & Lago, P. (2020). Architectural technical debt: A grounded theory. *European Conference on Software Architecture*, 202–219.

Wagner, S., & Ruhe, M. (2008). A structured review of productivity factors in software development.

Xiao, L., Cai, Y., Kazman, R., Mo, R., & Feng, Q. (2021). Detecting the locations and predicting the costs of compound architectural debts. *IEEE Transactions on Software Engineering*.

Zelman, W. N., Pink, G. H., & Matthias, C. B. (2003). Use of the balanced scorecard in health care. *Journal of health care finance*, *29*(4), 1–16.

A

Considered Metrics for Literature Review

A.1. ATD Metrics

The following analysis of existing literature regarding ATD measurements is included as supporting information for the literature review in section 2.1. The method through which these methods were found is explained in the same section. Three different categories of methods are identified. Ones concerning themselves with the sole identification of ATD, methods for assessing how ATD will grow over time and broader methods for determining the cost and benefit of ATD.

A.1.1. ATD Identification

Before being able to do any cost-benefit analysis it is first prudent to be able to identify ATD. Numerous methods have been suggested in literature, all with different benefits and drawbacks. Broadly speaking detection methods fall in one of three categories which will be discussed in the following sections.

Identification based on human interpretation is perhaps the most common manner of identifying and quantifying ATD. There exists a large body of knowledge based on interviews with experts who experience ATD (and its effects) first hand. Kazman et al.(2015) shows that ATD is indeed something that is experienced to be a significant problem in software development based on the opinions of experts. In research by Besker et al. (2017) evidence is even found that ATD might be one of the most troublesome forms of TD, as it grinds progress on a project to a halt, and discourages employees to work on the problem at all.

Additionally, research has been done to identify the common reasons for introducing ATD. This allows us to create a list of reasons for the creation of architectural debt. Such a list consists of (Verdecchia et al., 2020):

- Producing a minimum viable product
- Workarounds that stick
- Re-inventing the wheel
- Source Code ATD
- Architectural Lock Down
- New Context, Old Architecture

Additionally, architecture in general, and ATD specifically are very interesting in a micro services context. This form of software engineering is characterised by extremely high levels of modularisation

within the software, making proper architecting even more important. This form of software is growing rapidly, especially in cloud services, and brings a whole host of ATD issues with it. (De Toledo et al., 2021)

Li et al.(2015) and Eliasson et al.(2015) both created frameworks allowing for employees to identify and quantify ATD and its associated interest. While this framework is experienced by its users to be easy to use and useful it does rely on a lot of interpretation of the users, making it more difficult to apply quickly and in any situation.

Identification based on change logs works by considering the changes that occur over time in software. Clusters of files where large changes are made often can indicate that there is an architectural fallacy within the cluster of files. Mo et al. (2018) use this principle to detect and quantify different ATD items within a large company with a program called the DV8 suite. To work however, the system needs quite extensive information about the subject software. Namely:

- The dependency between the different files within the system
- The change history of the system
- The bug history of the system

Based on this data it is possible to use Mo et al.'s DV8 suite to determine which folders are clustered together. Then, based on the change history of the system, as well as the bug tracker the system can identify which changes require code changes in many different files within the cluster. This then allows DV8 to flag areas where such changes are common as likely ATD candidates. Based on empirical data and estimations of how much additional code needs to be rewritten for each change DV8 is then also able to give an estimation of both the principal involved in the ATD item, as well as its interest.

In a similar vein Diaz-Pace et al.(2020) created a system which is able to detect architecture code smells. Here smells are common bad practices found within programming. By applying this method over a longer period of time the system (by the name of Sen4Smells) is able to track different architectural code smells over time and determine their interest accordingly.

Identification based on source code would provide a more practical manner of detecting ATD. The source code will always be available (if the subject company is willing to collaborate) while detailed historic data about the evolution of a project might be more troublesome to acquire.

To show that this is indeed possible Li et al. (2014) showed that there is in fact a strong (negative) correlation between the average number of modified components per commit, which is a measure for ATD items, and two indexes of software modularity which can be measured directly from source code. namely:

1. Index of Package Changing Impact, which represents how independently different software packages are used.
2. Index of Package Goal Focus, which represents how much overlap there is between services offered by different components in the system.

A.1.2. Growth of ATD

As was mentioned briefly before, an additional field of interest besides the principal and interest associated with an ATD item is the way in which it develops over time. While Xiao et al. made mention of this within their regression model, it has also been the primary subject of a number of studies.

One possible line of reasoning is presented by Martini et al. (2015). As ATD grows so do the required maintenance activities needed to keep the software running. At some point these activities will become so time consuming that it is no longer practical, or even possible to keep working with the product. Martini et al. calls this point in time the ”crisis point”. With total periodic refactoring of the

software the crisis point could be postponed indefinitely, however even with large efforts to keep ATD to a minimum it is likely that ATD will grow over time due to hidden ATD items. Therefore, the crisis point will eventually be reached. The profit gained by refactoring ATD can then be represented as the additional usable lifetime of the product.

Regression based modelling can also provide interesting insights into the growth and development of ATD. Such a model has been created which is able to predict in what manner the interest of a certain ATD item will grow over time by looking at files with were error prone and tended to form errors in sets (Xiao et al., 2021). An interesting find is that in the vast majority of investigated cases ($\pm$ 75%) interest will grow linearly, while in $\pm 20\%$ of cases interest would grow exponentially. The remaining cases fell either into slow (logarithmic) or fluctuating (polynomial) growth patterns.

Finally, a noteworthy addition to the body of knowledge is made by Lenarduzzi et al.(2020) who compare the growth of ATD between monolithic systems and microservices. their main findings are that while transitioning to such a system ATD increases more quickly, but after ATD grows at rates as low as 1/10th the original rate.

A.1.3. Cost and Benefit of ATD Repayment

The main question all of the previously mentioned sources aim to work towards is when to repay which ATD. To this end cost benefit analysis tools are of incredible value, as they can provide insight as to exactly when it is viable for a company to refactor their software and by extension repay their ATD.

Common Management Strategies on their own do not provide full cost benefit analyses they do provide a good starting point for developing one, as employees can have many years of experience handling ATD.

Based on interviews with people fulfilling a number of different roles in the software development process Verdecchia et al. (2020) find three main categories of management strategies:

- Active Management Strategies consist of ways to ensure that the amount of ATD is kept at a minimum all the time, or even shrinks over time.
- Reactive Management Strategies are ways to handle ATD when it becomes a problem. This strategy does often result in the need for large refactoring activities at some point however.
- Passive Management Strategies often amount to neglecting the problem, because it is simply not considered worth it to solve.

Similar results were found in a recent survey based paper (Pérez et al., 2021). Here it is worth noting that the vast majority of suggested activities to manage ATD would fall in the reactive management category mentioned above.

Cost Benefit Analysis Tools represent one of the first steps towards a true cost benefit analysis. The first attempt at such a tool was made by Nord et al. (2012). In their research they compare the cost of a system where attention is paid to architecture from the get go vs the same system where the priority was to deliver the software as soon as possible. They found that the total cost of the first system was approximately 65% of the second system. It is worth noting that the cost of the first system is much more front loaded compared to the second system.

A slightly more realistic case study has been performed by Nayebi et al. (2019). Here, instead of making a decision at the start of a project a decision is made to refactor an existing project, allowing the authors to compare the product before and after the analysis. They found that after refactoring (in their particular case) the average time to resolve issues dropped by 72% and the average number of lines of code that needed to be changed to resolve the issue dropped from 102 to 34. It should be noted however, that it is somewhat questionable whether the systems before and after refactoring were directly comparable.

In general however, ATD decisions are not made at at a single point in time but as the project evolves and about many different ATD items. Quantifying these decisions was first tackled by Martini and Bosch (2016). Their empirical method, by the name of AnaConDebt, takes into consideration a number of factors to provide indicators which aid in making a decision on *if* and *when* to refactor ATD. For example, some ATD items are considered contagious, meaning that new components added will inherit the same ATD problems as the contagious item. This means that the principal and interest of such an item grow very rapidly, encouraging the quick repayment of the item in question. A similar model was presented two years later by Martini et al. (2018).

Holistic Approaches In section A.1.3 a number of approaches are discussed to manage and make decisions with regard to ATD. Most often however, these approaches exclusively consider maintenance cost when talking about the benefit of repaying ATD. There are however more tacit benefits that come with ATD repayment. In this sections papers discussing such aspects will be presented.

First initiatives to a broader approach to ATD management were undertaken by Martini and Bosch (2015). In a survey based study they found especially the impact of ATD on lead time, maintenance cost and risk are considered to be very important when making decisions about whether or not to repay ATD. These findings are confirmed in the case study by de Toledo et al. (2021) who found that by reducing ATD the total number of issues decreased, but more essentially to the company in question, the number of critical issues decreased as well. Hereby, reducing ATD increased reliability and reduced customer complaints.

A.2. SoD Metrics

In a similar vein as for ATD the following metrics were investigated in support of section 2.2 and as possible metrics for use in this project.

A.2.1. LOC Count

By far the simplest metric for measuring progress on a software project is to measure the number of LOC in a project. This is extremely easy to measure, and as a result this method has been around for a very long time, dating back as far as the 1960s (Fenton & Neil, 1999).

There are a number of issues with using LOC over time as a direct measurement for productivity. Firstly, the amount of code that needs to be written to achieve a specific goal varies greatly depending on the language the code is written in. Making it difficult to compare different technologies with one another. Additionally, it is often a sign of high code quality if the same goal can be achieved using a lower LOC count. For example, the number of LOC needed to fix issues decreased drastically after ATD repayment (Nayebi et al., 2019). Because of these reasons the use of LOC counts as a measure of development speed is rather limited in the context of this project.

Nevertheless, LOC counts are extremely easy to carry out, only requiring minimal analysis of the source code. It is likely for this reason that, despite its serious drawbacks, this method is still used quite often to measure the productivity of individual developers and development teams (Wagner & Ruhe, 2008). Nevertheless, some issues can arise when tracking LOC counts across different files when they are moved or renamed. If not handled properly this can cause to large LOC churn where none actually exists.

A.2.2. Function Points

FP were a response to LOC counts attempting to capture more detail about the complexity of written software while remaining usable across different technologies. FPs attempt to capture the functionality a system provides to its user. They were initially developed by Albrecht (1979). The method generally considers five different types of 'functions': External input types, external output types, logical internal file types, external interface file types and external inquiry types. The method then essentially functions by counting the number of functions at three different possible levels of complexity. Finally, the total is

adjusted by at most 35% to account for different circumstances, based on 14 processing complexity characteristics (Kemerer, 1987). Since its initial introduction numerous adjustments and improvements have been suggested to make the model more usable and accurate (Abran, 2019; Shah & Kama, 2018).

One of the more obvious benefits of FP is the fact that it can be applied across different technologies. This allows for better comparisons across different systems. FPs do suffer the downside that they are less intuitive than LOC. Everybody with a slight idea of how code is generally written is able to imagine what a line of code looks like. What a single function point represents is much more abstract. Perhaps it is because of this that when experts calculate FP for the same system their assessment sometimes differs as much as 30% (Sheetz et al., 2009). This could be avoided by automating the process, something that is discussed in the following paragraph.

There are numerous efforts to automate FP estimations, both based on the requirements of the software (Soubra et al., 2011) or design documentation (Lamma et al., 2004). Additionally, commercial solutions exist to count FP automatically based on source code (CAST, 2022). Finally, empirical research on how to convert LOC into FP and vice versa depending on the language of the code has existed for a long time (Albrecht & Gaffney, 1983). All of these points make it quite simple to measure FP at any point during the development process.

A.2.3. Cocomo 2

The constructive cost model 2 (Cocomo 2) (and its predecessor, Cocomo 81) is a regression model which attempts to predict the cost / effort of a particular program based on the estimated LOC needed to complete the project (Boehm et al., 1995). as well as a number of other requirement metrics such as the required security and reliability. The model is based on data from roughly 160 projects which ran in the US air force.

There are a number of issues with the Cocomo model. Firstly, the fact that it is ”only” based on 160 projects from a rather specific context makes it somewhat doubtful whether the model can easily be applied outside of this the context of a large military organisation. This point is reinforced by a number of papers that describe including additional projects to increase the accuracy of the model (Baik et al., 2002; Khatibi & Dorosti, 2016). Additionally, the model still requires an expert to give their opinion on a large number of different parameters, allowing for very different outcomes of similar, or even the same system.

This final points also makes it difficult to implement the model consistently on a large scale. Nevertheless, the model is very popular and automated systems exist to help easily and quickly applying the Cocomo model to software projects.

A.2.4. Story Points

In many agile working environments a number of points are attributed to a certain task to gain some insight into how large a specific task is. Usually, agile workflow programs like Atlassian’s Jira call these points Story Points and allow for easy tracking of story points across releases and programmers.

Story points have been used to estimate development effort (Coelho & Basu, 2012), however there are some problems with this approach. Story points are relative in nature, meaning two story points represent twice the effort of a single story points. There is however, no definition of the size of a single story point. This means the meaning of a single story points can vary greatly from company to company or even between teams. Additionally, the exact nature of a story points can also vary depending on implementation. They can represent development effort, complexity or other aspects (Hill et al., 2010). Nevertheless, story points are often used to calculate development velocity; the speed with which story points are completed (Moreira, 2013). Still, especially when comparing projects across different companies it is doubtful whether story points can be a consistent measure of productivity.

Due to their ubiquitous use and automated tracking in numerous agile working programs story points represent an easy and common way to measure development progress. Therefore is would be very easy to use this measure across different software development projects.

A.2.5. Issue Resolution Time

One could also look at the time an issue remains open as a measure of SoD. Issues are generally created to represent a task that needs to be completed. The speed with which they are completed could represent a measure of development speed. This method has been used before in a similar context as this project (Bijlsma et al., 2012).

One of the main benefits of using issue resolution time as a metric is that it remains very close to the actual point of interest. Namely, how quickly can an issue be raised, and subsequently resolved. Therefore it is a rather attractive metric to use. There are drawbacks however, most notably that issues are generally created, but not immediately worked on. It is not uncommon for certain types of issues to remain open for well over a year before work is even started.

Additionally, while this metric is easy to calculate with the help of an issue tracker, usage of which is very common, this kind of information is not commonly made public. Because of this, this metric is very well suited for usage within a single project, but does not lend itself well for large scale empirical analysis as is the intention of this project.

A.2.6. Lead Time / Time to Market

A different approach to measuring development speed could be to measure how quickly new releases follow one another. After all, reducing the time to market is one of the main reasons to invest in development teams. Measuring this directly by measuring the time between new releases might give the most direct insights into development speed.

By remaining close to the goal of development teams, namely to produce new features, the metric becomes a more direct measure of the area of interest. Nevertheless, one of the main strategies used to reduce the time between releases is to keep the scope of said releases down (Mäntylä et al., 2013). This means that the effort represented by a single release can vary extremely between different projects, making it difficult to compare between teams.

Measuring the time between releases should be very easy if the project makes use of some sort of version control software and version tags (like most projects do) one can simply compare the difference in time of release between different releases.

A.2.7. McCabe's Cyclomatic Complexity Metric

CC is another method attempting to estimate the complexity of a program rather than it's mere size (McCabe, 1976). It is slightly different from the other metrics mentioned because it does not attempt to find the size of a piece of code, but rather its complexity. Because most work in programming is in finding a suitable solution, not in typing lines of code, this might still provide a reasonable measure of how much work a specific function is to write. The metric finds its roots in graph theory. In essence the method works by creating a graph of a function where statements are represented by nodes and their relationships are represented by edges. By counting the number of linearly independent paths through a function one gets the cyclomatic metric.

The main argument that measuring CC is a worthwhile method for determining complexity is that intuitively it represents the complexity of a program rather well. Nevertheless, there has been serious critique on the metric for some time. It has been argued that it is difficult to assess what exactly CC measures, as well as that there are cases where the metric behaves unintuitively (Shepperd, 1988). More recently strong empirical evidence has been found that CC and LOC are strongly linearly related to one another. Therefore, it is argued, the metric is no more valid in measuring program complexity (or size) than LOC already was (Graylin et al., 2009).

When CC was first developed it had to be calculated by hand by developers. This was a long winded task, and so CC was simplified to ease this process. Since then however, it has become quite easy to automate the calculation of CC (Patel et al., 2015). As such it is quite easy to measure CC which is likely why it is still used widely in the industry to this day (Ebert et al., 2016).

B

Overview of SIGs Architecture Rating

The SIG AR is based on ten individual metrics. These metrics are outlined in the following sections.

B.1. Code Breakdown

Code breakdown represent the level of modularisation within the codebase. Generally speaking a larger degree of modularisation is desirable. Small components make it easier to comprehend how a module works before working on it. Additionally, it is easier to divide a large number of small components over one or more teams than to do so with a small number of large components. It is measured using the Gini coefficient, which is a measure of inequality (Dorfman, 1979). This means that systems with roughly equally sized modules will score well, while those with large variance in modules will score poorly.

B.2. Component Cohesion

Component cohesion is , as the name implies, a measure of how cohesive a component is. Generally speaking, a component should have rather specific functionality. If this is not the case, the component can be used by many different external components, causing the component to become entangled with many other functionalities. As a result, changes in the component do not only effect its immediate surroundings, but also other functionality within the system, making maintenance very difficult and increasing the risk of mistakes. This is quantified by calculating the ratio between the internal and external dependencies of the component. This means, that if a component mostly has dependencies within the component it is considered to have a high cohesion, while a component that relies a lot on other components for its functionality is considered to have low cohesion.

B.3. Component Coupling

Component coupling is similar to component cohesion, in that both are representations of how the system depends upon its different components. Here however, it is represented simply as a count of the number of external dependencies of the component, both incoming and outgoing. Such a dependency means that a component cannot be changed without also considering the components that depend on it, or it depends on. Because of this components with a small number of external dependencies are considered to be better.

B.4. Code Reuse

Code reuse is generally considered to be a bad practice in software engineering (Li et al., 2006). If the same code is used in multiple locations this means that whenever a change is made it needs to be

made in other places as well. This means that it will take longer to make changes and if a programmer is unaware of the fact that the same code should be changed elsewhere as well mistakes could enter the system. The code reuse of a component is measured by calculating the percentage of code in the component that is also present in other components.

B.5. Communication Centralization

When communication between components needs to take place, it is generally preferable if this happens through a low number of interfaces. This makes it easier to work within a component, as only changes that are being made in files that communicate with other components need to consider those other components. changes to all other files can be made while only considering the component itself. Communication centralisation captures this preference. The metric is defined as the percentage of files that communicates directly with other components.

B.6. Bounded Evolution

Bounded evolution measure to what degree components evolve with another. If when a change is made to one component, a change is also always made to another component they are considered to be co-evolving. Such a relationship implies that the two components rely on one another in some way. Once again this makes it more difficult to work in a system, as all co-evolving components need to be considered to make changes to any one of these components. This is measured by calculating the percentage of changes made in a component, that happen at the same time as changes made in other components. Components that tend to change on their own are scored higher than those that don't

B.7. Data Coupling

Once again, this metric is very similar to another metric, but now relating to data management. In this case it is very similar to component coupling. Data coupling measures the degree to which a single data store is used by multiple components. If a store is used by many different components it becomes difficult to make changes to the store without large changes to all the components reliant on it. This metric is quantified as the number of distinct incoming dependencies from other components that rely on the store in question. A small number of dependencies is considered to be better.

B.8. Technology Prevalence

Technology prevalence expresses that it is easier to work with a system which is built using common and modern technologies. Relying on technologies that are falling out of fashion makes it difficult to find up-to-date libraries, tools and employees with the right knowledge to work with the technology in question. Additionally, new technologies can provide opportunities dew to innovation which older technologies cannot. The metric is defined as the percentage of code that is written in a programming language that is considered common and / or modern, where a higher percentage provides a higher score.

B.9. Component Freshness

Knowledge about components on which no active development is taking place deteriorates over time. To represent this the component freshness metric is used in the architecture model. Components which are changed very often however, can also be an indication of poor architecture, showing that the component in question structurally requires a lot of work. to quantify this component freshness is ranked against a bell curve, where both very small, and very large percentual changes of the code in a component are considered to be signs of bad architecture.

B.10. Knowledge Distribution

The final metric included in the architecture model is knowledge distribution, representing how knowledge of components is distributed across the development team. Here a similar concept applies as for component freshness. Have a very small number of developers actively work on a component is risky, as all knowledge about the component could easily be lost. On the other hand, a very large number of programmers working on a component could hurt efficiency, as nobody is allowed to develop specialized knowledge about specific components. to quantify this, the number of programmers that actively contributes to the component is ranked against a bell curve, where both extreme ends of the spectrum are considered to be a sign of poor knowledge distribution.

C

Code

C.1. PyDriller - (Python)

```
# getCommits gets all commits associated with a certain project

def getCommits(url, project):
   print("Retrieving individual commits")
   c = []
   h = ["Hash", "Date", "Direct_Parent","Second_Parent","Deletions","Insertions"]
   n = 0
   for commit in
       Repository("".join(["https://github.com/",url])).traverse_commits():
      if len(commit.parents) > 1:
         c.append([commit.hash,
               commit.committer_date,
               commit.parents[0],
               commit.parents[1],
               commit.deletions,
               commit.insertions])
      elif len(commit.parents) == 1:
         c.append([commit.hash,
               commit.committer_date,
               commit.parents[0],
               None,
               commit.deletions,
               commit.insertions])
      else:
         c.append([commit.hash,
               commit.committer_date,
               None,
               None,
               commit.deletions,
               commit.insertions])

      n += 1
      if n % 100 == 0:
         print(str(n) + " commits retrieved")
   print("Process Complete, ", str(n), " commits retrieved. ")

   with open(str(project) + ".csv", "w") as f:
      writer = csv.writer(f)
      writer.writerow(h)
```

```
        writer.writerows(c)


# The following functions get pull numbers and associated hash



def getPullData(url, project):
    result = []
    pagenumber = 0
    print("Retrieving pull data")
    while True:
        apiresponse = re.get("https://api.github.com/repos/" + url + '/pulls?page='
            + str(pagenumber)+"&per_page=100&state=all",
                        auth=(user, auth))
        print(str("Page:" + str(pagenumber)))
        pagecontents = json.loads(apiresponse.content)
        result += pagecontents
        if len(pagecontents) < 100 | pagenumber > 200:
            break
        pagenumber += 1

    with open(str(project + "_pulls.json"), "w") as jsonFile:
        json.dump(result, jsonFile)


def listPullNumber(project):
    print("Listing merged pull requests")
    commits = []
    n = 0
    with open(str(project + "_pulls.json")) as json_file:
        data = json.load(json_file)
        while n < len(data) and data[n] != "message":
            if data[n]["merged_at"] is not None:
                commits.append([data[n]["number"],data[n]["merge_commit_sha"],
                    data[n]["merged_at"], data[n]["created_at"]])
            n += 1
    headers = ["Number", "Merge_Sha","Merged_At", "Created_At"]
    with open(str(project) + "_Pull_Merge.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerow(headers)
        writer.writerows(commits)
    return commits

def getAllPullData(url, project):
    getPullData(url, project)
    listPullNumber(project)

def getMultipleProjects(listofurls, listofprojects):
    if len(listofurls) != len(listofprojects):
        print("Lists are not of same length")
    n = 0
    while n < len(listofurls):
        print(listofprojects[n])
        getCommits(listofurls[n],listofprojects[n])
        getAllPullData(listofurls[n],listofprojects[n])
        n += 1


getMultipleProjects(wikiurls, wikiprojects)
```

C.2. Data Analysis - (R)

```
---
title: "Pull Analysis"
output: html_document
date: '2022-06-29'
---
```{r}

#install.packages("rjson")

library(tidyverse)
library(ggplot2)
library(foreach)
library(doParallel)
library(ranger)
library(kableExtra)
library(tictoc)
library(furrr)
library(magrittr)
library(profvis)
library(data.table)
library(progressr)
library(hexbin)
library(rjson)
library(gridExtra)
```
The following bloc generates some info on projects that is useful during the
    entire proces. The metadata dataframe is created simply to contain some basic
    information about the different projects. The createMetaData function then
    takes this dataframe and expands it to consider all the different snapshots of
    the project over different years. It also contructs full github urls and the
    age of the project. The optional "type" variable makes it possible to export
    the file in either as unix time (Used within the pull analysis), or as a
    yeardate format (Used for the bash script used to run multiple architecture
    analyses).
```{r}
repo <- "opensource/"
sourceurl <- "https://github.com/"

metadata <- rbind(
  c("Zuul", "Opensource", 2013, 2021, "netflix/zuul"),
  c("Bitcoin", "Opensource", 2010, 2021, "bitcoin/bitcoin"),
  c("Atom", "Opensource", 2012, 2021, "atom/atom"),
  c("Echarts", "Opensource", 2014, 2021, "apache/echarts"),
  c("FastJSON", "Opensource", 2012, 2021, "alibaba/fastjson"),
  c("Graal", "Opensource", 2012, 2021, "oracle/graal"),
  c("Gradle", "Opensource", 2009, 2021, "gradle/gradle"),
  c("GSON", "Opensource", 2010, 2021, "google/gson"),
  c("Jitsi_Meet", "Opensource", 2014, 2021, "jitsi/jitsi-meet"),
  c("JRuby", "Opensource", 2002, 2021, "jruby/jruby"),
  c("LibGDX", "Opensource", 2011, 2021, "libgdx/libgdx"),
  c("Puppet", "Opensource", 2006, 2021, "puppetlabs/puppet"),
  c("Rasa", "Opensource", 2017, 2021, "rasahq/rasa"),
  c("Roslyn", "Opensource", 2015, 2021, "dotnet/roslyn"),
  c("Quarkus", "Opensource", 2019, 2021, "quarkusio/quarkus"),
  c("Roslyn_Analyzers", "Opensource", 2016, 2021, "dotnet/roslyn-analyzers"),
  c("SigridCI", "Opensource", 2021, 2021, "software-improvement-group/sigridci"),
  c("SuperTux", "Opensource", 2004, 2021, "supertux/supertux"),
  c("Swift", "Opensource", 2011, 2021, "apple/swift"),
  c("Tensorflow", "Opensource", 2016, 2021, "tensorflow/tensorflow"),
```
```

```
  c("VertX", "Opensource", 2014, 2021, "eclipse-vertx/vert.x"),
  c("Visual_Code", "Opensource", 2016, 2021, "microsoft/vscode"),
  c("Webpack", "Opensource", 2013, 2021, "webpack/webpack"),
  c("FFmpeg", "Opensource", 2000, 2021, "ffmpeg/ffmpeg"),
  c("Axios", "Opensource", 2014, 2021, "axios/axios"),
  c("Deno", "Opensource", 2018, 2021, "denoland/deno"),
  c("Three", "Opensource", 2010, 2021, "mrdoob/three.js"),
  c("Typescript", "Opensource", 2014, 2021, "microsoft/typescript")
  )%>%
  data.frame()%>%
  rename(Project = X1, LOC = X2, Source = X3, Min = X4, Max = X5, url = X6)

createMetaData <- function(metadata, type = "unix"){
  N <- 1

  architectureinput <- data.frame(
    Project = character(),
    Url = character(),
    Git_year = double(),
    Age = double(),
    GitStartdate = character(),
    GitEnddate = character(),
    ArStartdate = character(),
    ArEnddate = character(),
    LOC = integer()

    )
  pb <- txtProgressBar(min = 0, max = length(metadata$Project), style = 3)
  for (n in 1:length(metadata$Project)){
    min <- as.numeric(metadata$Min[n])
    max <- as.numeric(metadata$Max[n])
    year <- as.numeric(metadata$Min[n])

    fulldf <- read.csv(paste0(repo ,metadata$Project[n],".csv"))

    while (year <= max){

       yeardf <- filter(fulldf,
            as.numeric(as.POSIXct(gsub(".{3}$", "00", Date), format = "%Y-%m-%d
                %T%z")) > as.numeric(as.POSIXct(paste0(year - 1, "-01-01"), format
                = "%Y-%m-%d")) &
            as.numeric(as.POSIXct(gsub(".{3}$", "00", Date), format = "%Y-%m-%d
                %T%z")) < as.numeric(as.POSIXct(paste0(year - 1, "-12-31"), format
                = "%Y-%m-%d")))

       LOC_Change <- sum(yeardf$Insertions) - sum(yeardf$Deletions)


      if (type == "unix"){
        architectureinput[N, ] <- c(metadata$Project[n],
                              paste0(sourceurl,metadata$url[n],".git"),
                              year,
                              year - min ,
                              as.numeric(as.POSIXct(paste0(year, "0101"), format =
                                  "%Y%m%d")),
                              as.numeric(as.POSIXct(paste0(year, "1231"), format =
                                  "%Y%m%d")),
                              as.numeric(as.POSIXct(paste0(year - 1, "0101"), format
                                  = "%Y%m%d")),
                              as.numeric(as.POSIXct(paste0(year - 1, "1231"), format
                                  = "%Y%m%d")),
```

```
                            LOC_Change)
    } else{
      architectureinput[N, ] <- c(metadata$Project[n],
                            paste0(sourceurl,metadata$url[n],".git"),
                            year,
                            year - min ,
                            paste0(year, "0101"),
                            paste0(year, "1231"),
                            paste0(year - 1, "0101"),
                            paste0(year - 1, "1231"),
                            LOC_Change)
    }
    year <- year + 1
    N <- N + 1
   }
   setTxtProgressBar(pb, n)
  }
  return(architectureinput)
}

architectureinput <- createMetaData(filter(metadata, Source == "BT"), type =
    "unix") %>%
  group_by(Project)%>%
  mutate(Size = cumsum(LOC))
```
```
The following block of code simply writes a csv file for use in conjunction with
    the SIG Architecture analysis tool
```{r}
write.table(select(
  architectureinput,
  c(Project,
    Url,
    Git_year,
    GitStartdate,
    GitEnddate)
),
file = "inputData.txt",
sep = " "
)
```

All main functions used for the actual analysis are defined below.
```{r}
#Count Parents counts how often the parent of the commit taken as input can be
    found as a parent across the dataset. It takes a hash and a list of commits as
    input and returns an integer As is the case with many other functions in this
    bloc the provided commits dataset needs to be indexed according to the hash in
    order to work properly..
countParents <- function(hash, commits){
  a <- as.integer(length(filter(commits, Direct_Parent ==
      commits[.(hash)]$Direct_Parent)$Hash))
  b <- as.integer(length(filter(commits, Second_Parent ==
      commits[.(hash)]$Direct_Parent)$Hash))
  return(a+b)
}

#findPull follows a string of parents until it finds a commit which parents show
    up more than once. This way the functino takes a starting hash and a list of
    hashes as input and returns a vector of hashes representing a pull. Note that
    the provided dataframe must be indexed by hash in order to make the function
    complete in a reasonable timeframe.
```
```

```
findPull <- function(hash, commits){
  inicommit <- commits[.(hash)]
  suppressWarnings(
    if(is.na(inicommit$Second_Parent)){
      return(NA)
    }
  )
  commit <- commits[.(inicommit$Second_Parent)]
  suppressWarnings(
    if(is.na(commit$Direct_Parent)){
      return(NA)
    }
  )
  pull <- c(inicommit$Hash, commit$Hash)
  while (countParents(commit$Hash, commits) < 2){
    commit <- commits[.(commit$Direct_Parent)]
    suppressWarnings(
      if(is.na(commit$Direct_Parent)){
        return(pull)
      }
    )
    pull <- c(pull, commit$Hash)
  }
  return(pull)
}

#aggregatePull takes a pull as input and outputs a vector of useful data about
    this vector, namely: the number of commits in the pull, the time of the
    initial commit, the time of the final commit, the difference in time between
    these two the number of LOC added and deleted over the pull, as well as the
    sum of these two.
aggregatePull <- function(pull, commits){
  len <- length(pull)
  insertions <- 0
  deletions <- 0
  if (len < 3){
    return(NA)
  }
  for (n in 1:length(pull)){
    commit <- commits[.(pull[n])]
    if (n == 1){
      merge_commit <- commit$Date
    }

    if (n == 2){
      last_commit <- commit$Date
    }

    if (n == length(pull)){
      initial_insertions <- commit$Insertions
    }

    if (n == length(pull)){
      initial_deletions <- commit$Deletions
    }

    if (n == length(pull)){
      first_commit <- commit$Date
    }
```

```
    #Excluding Merge Pull from stats (as this is simply the total content of the
        pull).
    if (n != 1){
      insertions <- insertions + commit$Insertions
      deletions <- deletions + commit$Deletions
    }
  }
  return(c(len, merge_commit, first_commit, last_commit, insertions, deletions,
      initial_insertions, initial_deletions))
}

#aggregateAllPulls simply performs the aggregatePull function over an entire
    dataframe instead of a single pull. It takes a list of pulls and a list of
    commits as input, as well as a name for the particular project. The function
    does not return anything, but instead saves a csv with the name of the project.
aggregateAllPulls <- function(pulls, commits, project, year){
  data <- vector("list", 8)
  k <- 0
  m <- length(pulls$Hash)
  pb <- txtProgressBar(min = 0, max = m + 1, style = 3)

  for (n in pulls$Hash){
    pull <- findPull(n, commits)
    row <- aggregatePull(pull, commits)
    data <- rbind(data, as.list(row))
    k <- k + 1
    setTxtProgressBar(pb, k)
  }

  write.csv(data, file = paste0(repo, "Data_On_Pulls_", project,"_", year, ".csv"))

}

#aggregateProject does a number of things. it can be provided a project for which
    the required files are placed in the repo folder and will analyse this
    project. More specifically, it will read in the data and filter it to only
    include pulls and commits between the start and stop date. Additionally, it
    will index the commits provided according to their hash in order to speed up
    the process. Finally it applies the aggregateAllPulls function.
aggregateProject <- function(project, start, stop, year){

     writeLines(paste("\n", project, year))

     commits <- read.csv(paste0(repo, project, ".csv"))

     commits <- commits %>%
       mutate(Date = as.numeric(as.POSIXct(gsub(".{3}$", "00", Date), format =
           "%Y-%m-%d %T%z")),
             Hash = as.character(Hash),
             Direct_Parent = as.character(Direct_Parent),
             Second_Parent = as.character(Second_Parent))%>%
       filter(Date > start & Date < stop)

     commits <- setkey(as.data.table(commits), Hash)

     pulls <- commits %>%
       filter(Second_Parent != "")

     print(paste0("Analysing ", length(commits$Hash), " Commits over ",
         length(pulls$Hash), " Pulls"))
```

```
    aggregateAllPulls(pulls, commits, project, year)
}

#aggregateMultipleProjects() takes care of running the aggregateProject()
    function over a large number of projects split into different snapshots.
aggregateMultipleProjects <- function(input){
 for (n in 1:length(input$Project)){
  aggregateProject(input$Project[n],
                input$GitStartdate[n],
                input$GitEnddate[n],
                input$Git_year[n])
 }
}

#averageData loads a .csv file as provided by the project input and aggregates a
    number of statistics about this collection of pulls.
averageDataMin <- function(project, year){
 if (file.info(paste0(repo, "Data_On_Pulls_",project,"_", year, ".csv"))$size ==
     3){
  return(NA)
 }
 df <-read.csv(paste0(repo, "Data_On_Pulls_",project,"_", year, ".csv"))[-1,-1]

 df <- filter(df,
      #Pulls with a single commit don't contain any information (as they are only
          the merge commit) and so are filtered out, meanwhile very large pulls
          are no longer representative for a unit of work, so they are filtered
          out as well.
      as.numeric(V1) > 2 & as.numeric(V1) < 100,
      #Filter stating that the time difference cannot be negative, or smaller
          than one hour, nor can it be larger than one month
      (as.numeric(V2)-as.numeric(V3)) > 3600 & (as.numeric(V2)-as.numeric(V3)) <
          (31 * 24 * 60 * 60),
      )

 median_commits <- median(as.numeric(df$V1), na.rm = TRUE)
 median_time <- median(as.numeric(df$V4) - as.numeric(df$V3), na.rm = TRUE) /
     (3600 * 24)
 median_merge_time <- median(as.numeric(df$V2) - as.numeric(df$V3), na.rm = TRUE)
     / (3600 * 24)
 median_insertions <- median(as.numeric(df$V5), na.rm = TRUE)
 median_deletions <- median(as.numeric(df$V6), na.rm = TRUE)
 median_initial_insertions <- median(as.numeric(df$V7), na.rm = TRUE)
 median_initial_deletions <- median(as.numeric(df$V8), na.rm = TRUE)
 number_of_pulls <- as.numeric(length(df$V1))

 return(c(project, year, median_commits, median_time, median_merge_time,
     median_insertions, median_deletions, median_initial_insertions,
     median_initial_insertions, number_of_pulls))
}



#ImportMultipleProjects simply creates a dataframe with statistics of all the
    projects provided in the list of projects.
importMultipleProjects <- function(input){
 df <- data.frame(
 Project = character(),
 Year = double(),
 Number_Of_Commits = double(),
```

```
  Median_Time = double(),
  Median_Merge_Time = double(),
  Median_Insertions = double(),
  Median_Deletions = double(),
  Median_Initial_Insertions = double(),
  Median_Initial_Deletions = double(),
  Number_Of_Pulls = double()
  )
  for (n in 1:length(input$Project)){
    #print(paste(input$Project[n], input$Git_year[n]))
    df[n,] <- averageDataMin(input$Project[n], input$Git_year[n])
  }
  return(df)
}


#combineAllData() simply combines all proeject files found in the architecture
    input into a single csv file in order to allow analysis on all projects
    simultaniously Additionally, it adds a column for the project the particular
    pull comes from
combineAllData <- function(architectureinput){
  alldata <- data.frame()
  for (n in 1:length(architectureinput$Project)){
     alldata <- rbind(
       alldata, mutate(
         read.csv(
           paste0(
             repo, "Data_On_Pulls_", architectureinput$Project[n], "_"
                 ,architectureinput$Git_year[n], ".csv"
             )
           )
         ,Project = architectureinput$Project[n]
         )
       )
  }
  write.csv(alldata, paste0(repo, "Data_On_Pulls_All.csv"))
}

```

The following functions sole purpose is to import architecture ratings.

```{r}
#extractArchitectureRating simply creates some robustness with regards to missing
    variables, by checking if the variable exists or returns a NULL and if this is
    the case passing NA as that particular variable.
extractArchitectureRating <- function(output, variable, n = 1){
  var <- output[["systemElements"]][[n]][["measurementValues"]][[variable]]
  if(is.null(var)){
    return(NA)
  }
  else{
    return(var)
  }
}


createArchitectureRatings <- function(){

  architectureratings <- data.frame(
    ID = integer(),
    Architecture_Rating = double(),
```

```
  Authors = double(),
  Bounded_Evolution_Rating = double(),
  Churn = double(),
  Churn_Per_Author = double(),
  Code_Age = double(),
  Code_Breakdown_Rating = double(),
  Code_Reuse_Rating = double(),
  Commits = double(),
  Communication_Centralization_Rating = double(),
  Communication_Rating = double(),
  Complexity = double(),
  Component_Cohesion_Rating = double(),
  Component_Coupling_Rating = double(),
  Component_Freshness_Rating = double(),
  Data_Aacess_Rating = double(),
  Data_Coupling_Rating = double(),
  Duplication = double(),
  Evolutionn_Rating = double(),
  Files = double(),
  Inter_Component_Duplication = double(),
  Knowledge_Distribution_Rating = double(),
  Knowledge_Rating = double(),
  Lines = double(),
  Lines_Of_Code = double(),
  Local_Duplication = double(),
  Non_Duplicated_Code = double(),
  Structure_Rating = double(),
  Team_Fractal = double(),
  Technology_Prevalence_Rating = double(),
  Technology_Stack_Rating = double(),
  Yearly_Churn_Percentage = double()
)
pb <- txtProgressBar(min = 0, max = length(architectureinput$Project), style = 3)
for(n in 1:length(architectureinput$Project)){
  if (file.exists(paste0(repo, "results/sig-output",n,".json"))){
  output <- fromJSON(file = paste0(repo, "results/sig-output",n,".json"))
  row <- c(n,
         extractArchitectureRating(output ,"ARCHITECTURE_RATING"),
         extractArchitectureRating(output ,"AUTHORS"),
         extractArchitectureRating(output ,"BOUNDED_EVOLUTION_RATING"),
         extractArchitectureRating(output ,"CHURN"),
         extractArchitectureRating(output ,"CHURN_PER_AUTHOR"),
         extractArchitectureRating(output ,"CODE_AGE"),
         extractArchitectureRating(output ,"CODE_BREAKDOWN_RATING"),
         extractArchitectureRating(output ,"CODE_REUSE_RATING"),
         extractArchitectureRating(output ,"COMMITS"),
         extractArchitectureRating(output ,"COMMUNICATION_CENTRALIZATION_RATING"),
         extractArchitectureRating(output ,"COMMUNICATION_RATING"),
         extractArchitectureRating(output ,"COMPLEXITY"),
         extractArchitectureRating(output ,"COMPONENT_COHESION_RATING"),
         extractArchitectureRating(output ,"COMPONENT_COUPLING_RATING"),
         extractArchitectureRating(output ,"COMPONENT_FRESHNESS_RATING"),
         extractArchitectureRating(output ,"DATA_ACCESS_RATING"),
         extractArchitectureRating(output ,"DATA_COUPLING_RATING"),
         extractArchitectureRating(output ,"DUPLICATION"),
         extractArchitectureRating(output ,"EVOLUTION_RATING"),
         extractArchitectureRating(output ,"FILES"),
         extractArchitectureRating(output ,"INTER_COMPONENT_DUPLICATION"),
         extractArchitectureRating(output ,"KNOWLEDGE_DISTRIBUTION_RATING"),
         extractArchitectureRating(output ,"KNOWLEDGE_RATING"),
         extractArchitectureRating(output ,"LINES"),
```

```
          extractArchitectureRating(output ,"LINES_OF_CODE"),
          extractArchitectureRating(output ,"LOCAL_DUPLICATION"),
          extractArchitectureRating(output ,"NON_DUPLICATED_CODE"),
          extractArchitectureRating(output ,"STRUCTURE_RATING"),
          extractArchitectureRating(output ,"TEAM_FRACTAL"),
          extractArchitectureRating(output ,"TECHNOLOGY_PREVALENCE_RATING"),
          extractArchitectureRating(output ,"TECHNOLOGY_STACK_RATING"),
          extractArchitectureRating(output ,"YEARLY_CHURN_PERCENTAGE"))
   architectureratings[n,] <- row
   setTxtProgressBar(pb, n)
   }
  }
  return(architectureratings)
}
```

```
The following functions can be run (in order) to perform a full analysis.
```{r}
aggregateMultipleProjects(architectureinput)
```
```{r}
architectureratings <- createArchitectureRatings()
```
```{r}
pullratings <- importMultipleProjects(architectureinput)
```
```{r}
Age <- architectureinput$Age
Size <- architectureinput$Size
round <- .5

ratings <- cbind(architectureratings, pullratings, Age, Size)
```
```