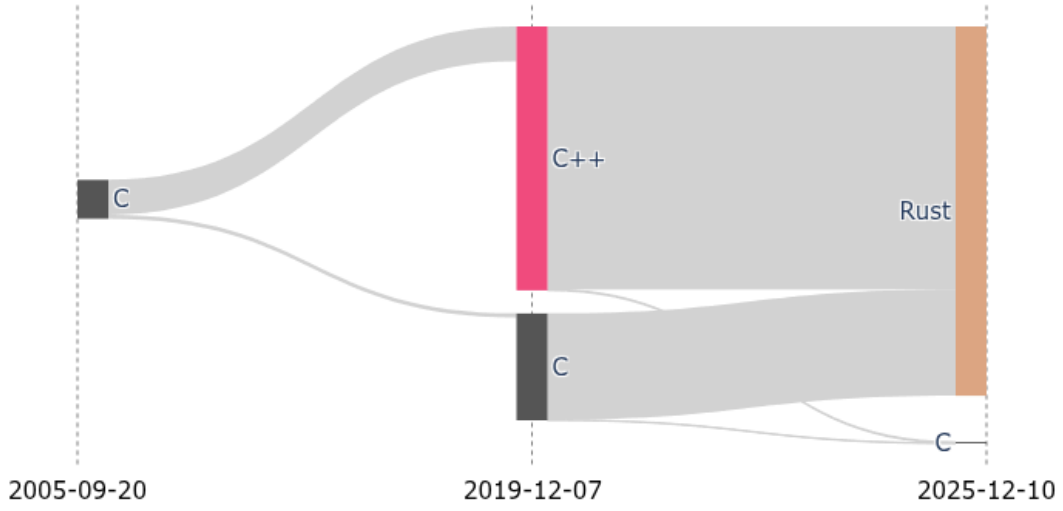
On Software Projects that Migrate to Rust

Version of 8 July, 2026



Jort van Driel

On Software Projects that Migrate to Rust

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jort van Driel
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Identifier movement in fish-shell (Own)

On Software Projects that Migrate to Rust

Author: Jort van Driel
Student ID: 6334377

Abstract

Memory safety vulnerabilities present a critical challenge, driving both commercial and open-source projects to adopt the Rust programming language. Despite this trend, no previous research has examined the approaches taken by open-source projects migrating to Rust and their impact on software quality.

This thesis takes a mixed-methods approach to analyse the migration process. It introduces a novel, language-agnostic technique that helped identify 285 migrations by tracking Abstract Syntax Tree (AST) identifiers over time and across different repositories. Interviews with 38 of these projects revealed that they were migrating not only from C/C++ for memory safety, but also from higher-level languages such as Go, Python and JavaScript, in order to improve performance and maintainability.

Although project authors overwhelmingly report that their migration was successful, a quantitative analysis of the same projects across 22 metrics presents a more nuanced perspective. It shows testing practices improved post-migration, while some structural metrics showed a decline, with increased code duplication and significant reliance on `unsafe` code blocks. This highlights a possible discrepancy between developer perception and measurable code quality, suggesting that successful migration may be helped with more upfront architectural planning.

Thesis Committee:

Chair:	dr. prof. Diomidis Spinellis
Committee Member:	dr. Jesper Cockx
University Supervisor:	dr. prof. Diomidis Spinellis

Preface

From a young age, I was drawn to the world of computers and software. I clearly remember how a course on the basics of HTML, JavaScript, and so on (unexpectedly taught by my Dutch teacher) sparked my fascination with programming. While many see creating our everyday digital services as something complex yet dull, I don't think I'm the first to argue that there is a certain beauty in the ability to express myself creatively through, well, *code*. Although I wouldn't describe software development as art per se (it is rarely written for the process rather than the results), the set of optimisation problems inherent in each project do require creativity and ingenuity to solve. Using these six years of university to specialise and learn the necessary skills, I think, more importantly, it has shown me that the solutions we build are often shaped as much by industry and societal trends as they are by our own creativity.

Our increasing reliance on AI clearly demonstrates this. While it is a logical solution for solving these optimisation problems more efficiently and for shipping more quickly, it moves our focus away from the craftsmanship of the code itself and more to the results it creates. As this thesis shows, there is a great deal of interest in migrating to Rust, both commercially and in open-source projects. Rust introduces many new concepts that encourage projects to migrate, from system languages to high-level languages. To me, this shows that, although AI brings a number of benefits, we mustn't allow ourselves to become locked into the languages in which AI works well, and instead strive to generate and embrace new ideas that improve how we develop and create. In this regard, I hope that this research will help clarify the steps involved in moving to a modern programming language such as Rust, *so that we may continue to embrace new ideas*.

Having said all that, I owe a large amount of gratitude to my girlfriend, Veerle, for her support during the writing of this thesis and during the years prior. I would also like to thank my parents and friends for shaping who I am today and showing me that I can rely on them in the future, just as they can rely on me. Lastly, I would like to express my sincere gratitude to my supervisor, Diomidis, for his sharp guidance and his willingness to provide ideas and feedback that have helped transform this thesis into something I can be truly proud of.

Jort van Driel
Delft, the Netherlands
2026-07-08

Contents

Preface vii

Contents ix

1 Introduction 1
1.1 Motivation . . . 1
1.2 Research questions . . . 2
1.3 Research contributions . . . 3
1.4 Thesis outline . . . 4
1.5 A note on the use of AI . . . 4

2 Background and related work 5
2.1 Memory safety and Rust . . . 5
2.2 Software language migration . . . 6
2.3 Software quality . . . 7
2.4 Evaluating open-source software quality . . . 9
2.5 Adoption of Rust and its ecosystem . . . 9
2.6 Migration detection and analysis . . . 10
2.7 Empirical studies on language migration . . . 11
2.8 Research on Rust migration and translation . . . 12

3 Methodology 13
3.1 RQ1: Finding migrations to Rust . . . 13
3.2 RQ2: Interviewing project authors . . . 20
3.3 RQ3: Measuring quality changes . . . 22

4 Projects that migrated to Rust 26
4.1 Collected projects . . . 26
4.2 In-repository measurements . . . 27
4.3 Cross-repository measurements . . . 29
4.4 Final selection and verified set . . . 30
4.5 Collected characteristics . . . 32
4.6 Improved bounds using the migration and verified sets . . . 33

5 Interview results **34**
5.1 Context 34
5.2 Motivating reasons 36
5.3 Strategies applied 39
5.4 Perceived results 40
5.5 Difficulties and reflection 41

6 Measurement results **44**
6.1 Derived metrics 44
6.2 Collected measurements 50

7 Threats to validity **58**
7.1 Conclusion validity 58
7.2 Internal validity 59
7.3 Construct validity 60
7.4 External validity 60

8 Conclusions **62**
8.1 Implications 63
8.2 Limitations 64
8.3 Future work 64

Bibliography **65**

A Verified project set **71**

Chapter 1

Introduction

This chapter introduces the context and motivation for studying open-source software migrations to the Rust programming language. It sets the primary research goal and the specific research questions that steer this thesis, outlines the key methodological and empirical contributions, and concludes with an overview of the thesis structure.

1.1 Motivation

Memory safety issues remain a leading cause of vulnerabilities in large-scale software systems. In 2019, Microsoft reported that around 70 percent of their vulnerabilities between 2006 and 2018 were linked to memory safety issues [1]. Similarly, Google found that 76 percent of Android vulnerabilities in 2019 shared this root cause [2]. However, over the past six years, this proportion in Android has dropped to under 20 percent. A key factor in this decline has been the gradual adoption of the memory-safe programming language Rust.

Motivated by these security benefits, an increasing number of software projects are migrating from legacy languages to Rust. Notable examples include the Rust implementation of GNU coreutils[1] and Mozilla's ongoing 'oxidation' of Firefox. However, major technology companies tend to keep the specific migration processes they employ opaque. Consequently, examining publicly available open-source projects is a useful way to observe and analyse these migration strategies.

Despite this industry trend, there is a gap in empirical software engineering research concerning the migration of projects to Rust. While recent academic work has highlighted automated techniques for translating C code to Rust [3], no prior empirical studies have examined the overarching, architectural strategies that developers use when migrating software projects. Furthermore, it remains unclear whether these language migrations successfully improve software quality beyond merely preventing memory safety violations.

This thesis aims to address this gap by investigating the migration strategies employed by open-source projects and their subsequent impact on software quality. To

[1] uutils/coreutils: https://github.com/uutils/coreutils

this end, the study employs a mixed-methods approach. Firstly, a quantitative analysis tracks selected quality attributes before, during and after migration, employing a novel, language-agnostic technique that involves tracking Abstract Syntax Tree (AST) identifiers. Secondly, qualitative interviews with project developers provide an empirical view of their motivations, strategies employed, and experiences. Thirdly, the projects are measured quantitatively to determine the extent of the changes experienced compared to the interview results.

The research also aims to help organisations and developers make informed decisions about the potential benefits of adopting Rust in their specific circumstances. To this end, it provides insights into strategies that improve software quality during language transitions, based on a large number of case studies of the migration process.

1.2 Research questions

In order to guide this research, we first define our main research objective and establish the scope of our study. Following the empirical research guidelines provided by Wohlin et al. [4], we designed our research questions to address this main objective sequentially, combining exploratory, qualitative and quantitative methods.

Main research goal
How does migration to Rust relate to changes in the quality of popular open-source software projects?

To ensure that our goal is both feasible and rigorous, we have established three boundaries to define the scope of our research. Firstly, due to the industry trend discussed in our motivation, we restrict our analysis to migrations that target the Rust programming language. Secondly, our focus is solely on open-source software (OSS). This ensures that we have public metadata with which we can detect migrations and evaluate software quality reproducibly. Lastly, we filter our dataset to include only *popular* projects (defined in our methodology as >300 GitHub stars). Kalliamvakou et al. [5] caution that the majority of repositories on GitHub are inactive or small 'toy' projects, and so filtering by popularity ensures that we are more likely to study engineered software.

We break down our main research goal into the following three sequential research questions:

RQ1
Which popular open-source projects have (partially) migrated from a different programming language to Rust, and what are their main characteristics?

In order to answer RQ1, we first need to identify a dataset of projects that have been fully or largely migrated to Rust. To detect migrations, we conduct an exploratory, quantitative archival analysis by tracking identifiers across time and codebases. We then perform a classification analysis to determine these projects' primary characteristics. The results of

this exploratory phase determine the pool of participants for our subsequent case studies in RQ2 and RQ3.

RQ2
Which goals motivated these projects to migrate, how did they go about it, and did they experience achieving these goals?

In RQ2, we explore the human and architectural aspects of the migration process. We take an exploratory, qualitative approach, using semi-structured interviews with the project authors. By applying thematic analysis to these interviews, we extract the primary motivations of the developers, the strategies they employed, and their subjective perceptions of whether the migration was successful.

RQ3
What are the measured changes during and after migration for the mentioned goals?

Finally, RQ3 contrasts the developers' perceived experiences (RQ2) with the measured results. We perform a descriptive, quantitative analysis of the projects' codebases before, during, and after migration. To ensure our measurements are meaningful, we track metrics that are directly related to the goals stated by participants in RQ2. Using statistical analysis, we compare the software's measured quality against the perceived and theoretical benefits of migrating to Rust.

1.3 Research contributions

This thesis makes three contributions to the field of empirical software engineering, with a specific focus on software language migration research and Rust adoption:

- We present a systematic, language-agnostic approach to detecting software migrations, which involves tracking identifiers over time and across codebases. Although this study is focused on identifying migrations to Rust, the underlying methodology can be applied more broadly to other programming languages. Alongside this technique, we are publishing datasets of repositories and verified migrations that have been collected from GitHub.
- This work introduces several new Rust libraries for analysing `git` repositories. We developed `truck-facto-rs`[2] to calculate a repository's Truck Factor, and `szz-rs`[3] to efficiently find fix-inducing commits. Additionally, we extended Mozilla's Rust-Code-Analysis[4] to support the broader set of programming languages analysed in this study.
- We present the key insights gained from our 38 interviews with project authors and our quantitative analysis across 22 metrics. By demonstrating where these projects were improved or encountered difficulties, this research provides practical guidance

[2]truck-facto-rs: https://github.com/JortvD/truck-facto-rs
[3]szz-rs: https://github.com/JortvD/szz-rs
[4]Fork of Rust-Code-Analysis: https://github.com/JortvD/rust-code-analysis/

for developers considering migrating to Rust, while also highlighting areas for future research.

A replication package, along with all the code produced for this thesis, are available on GitHub[5].

1.4 Thesis outline

This thesis is structured according to the empirical reporting guidelines recommended by Wohlin et al. [4]. The document is organised as follows: Chapter 2 establishes the theoretical background and reviews related literature. Chapter 3 then details the overall research methodology, including the procedures for collecting and analysing data. The results are presented across three chapters, with one chapter addressing each research question. Chapter 4 presents the automated detection and manual verification of the migrated projects (RQ1); Chapter 5 explores the qualitative insights from the developer interviews (RQ2); and Chapter 6 presents the quantitative measurements of software quality for the migrated projects (RQ3). Please note that, although metric selection is typically placed in the methodology chapter, we present it in a later chapter because it is derived a posteriori from the interview findings. Finally, Chapter 7 evaluates the potential threats to validity, and Chapter 8 concludes the thesis by summarising the findings, discussing practical implications, and outlining directions for future work.

1.5 A note on the use of AI

GitHub Copilot was used to assist with the coding for this thesis, primarily for its next-line prediction feature, which increased efficiency. Although we initially intended to use Gemini for literature research, we ultimately found that snowballing known papers and using search tools such as Semantic Scholar was a more effective approach. The interview transcripts were transcribed verbatim using Teams, then processed into clean verbatim versions using a local model to remove filler words and transcription artefacts. The thesis itself was spell-checked using DeepL Write[6] and refined manually based on feedback from Gemini Pro 3.1, which was asked to provide feedback as if it were a peer.

[5]Replication package: https://github.com/JortvD/rust-migration-thesis

[6]DeepL Write: https://www.deepl.com/en/write

Chapter 2

Background and related work

This chapter provides the theoretical background and reviews related literature in order to contextualise the research. It covers the concepts of memory safety and the evolution of the Rust programming language, as well as the definitions and techniques surrounding software language migration and established models of software quality. It also discusses previous empirical studies on software quality, migration detection and Rust adoption.

2.1 Memory safety and Rust

Since their inception in the early 1970s, C and its successor, C++, have been the *de facto* standards for systems programming and other performance-critical applications. Despite ongoing efforts to modernise these languages, they fundamentally rely on manual memory management, which places the entire burden of memory safety on the developer. A program is generally considered memory-safe if it is free from vulnerabilities relating to unauthorised or incorrect memory access. More formally, Dhurjati et al. [6] define "a software entity (a module, thread, or complete program) to be *memory-safe* if (a) it never references a memory location outside the address space allocated by or for that entity, and (b) it never executes instructions outside the code area created by the compiler and linker within that address space." The consequences of manual memory management can be severe. In 2019, both Microsoft's Security Response Center [1] and Google's Chrome Safety Team [7] reported that memory safety violations accounted for around 70 percent of all security vulnerabilities in their products.

To enforce memory safety, most modern programming languages remove the need for developers to manage memory themselves. The most common technique is garbage collection (GC), which is a runtime process that automatically frees memory that the program can no longer access. However, this safety feature incurs performance overheads. Tracking references and periodically pausing execution to free memory consumes computational resources and can cause unpredictable latency spikes. This unpredictability conflicts directly with the requirements of low-level systems programming, which relies on deterministic performance and minimal resource consumption.

Introduced in 2010, Rust was designed to bridge this exact gap by providing guaranteed memory safety without the need for a garbage collector. Like C++, Rust adheres to the zero-cost abstraction principle, which states that developers should not pay a performance penalty for unused features, and that abstractions should compile into code that is as efficient as hand-written low-level code. Rather than relying on a runtime garbage collector, Rust enforces strict rules regarding variable ownership and access (borrowing), which are verified entirely at compile time.

Over time, the scope of this compile-time verification has been expanded to prevent other classes of complex bugs. For instance, the compiler actively prevents iterator invalidation, which occurs when an underlying collection is modified while being iterated over, as well as data races, which happen when multiple threads attempt to simultaneously read and write to the same memory without synchronisation. Consequently, not only does Rust match the safety of traditional memory-safe languages such as Java and Go, it often provides stronger guarantees for concurrent programming too, without sacrificing the performance of a low-level language.

2.2 Software language migration

The concept of software migration is as old as software itself, yet the precise definitions and terminology can vary significantly depending on the context. For example, ISO/IEC/IEEE [8] defines *transition* as "activities involved in moving a new or changed service, system, or component to or from an environment." Brodie and Stonebraker [9] offer a broader perspective, describing *migration* as "replacing problematic hardware and software … with newer, more modern hardware and software." They also distinguish between two main strategies: migrating an entire system at once ("Cold Turkey") or taking an incremental, phased approach ("Chicken Little").

In order to categorise how software evolves over time, Weiderman et al. [10] define three distinct phases: maintenance, modernisation, and replacement. Unlike maintenance, which involves non-structural, incremental updates such as bug fixes, modernisation and replacement involve significant structural changes. Modernisation alters the system's structure while preserving the rest of the environment, whereas replacement aims to completely replace the existing system.

The concepts of environmental shifts and system replacement apply directly to the process of migrating a codebase from one programming language to another. However, the terminology used in language migration is highly specific. Terekhov and Verhoef [11] use the term language *conversion* to describe the process of moving between languages without altering the underlying functionality, architecture or design decisions. Visser [12] takes a different approach, analysing the abstraction levels between the source and target languages. He defines *program transformation* as the process of converting one program into another, and refers to cross-language changes as *translations*. Visser categorises these translations into three types: *synthesis* (moving to a lower abstraction level, such as compilation),

reverse engineering (moving to a higher abstraction level), and *migration* (moving between languages at the same level of abstraction).

> In the context of this thesis, we define a *migration* as the process of moving a program from one programming language to another at the same level of abstraction (i.e., the source code manipulated by developers). This definition encompasses translations involving a paradigm shift, such as moving from an object-oriented language to a functional one.

The tools and techniques available for language migration have evolved to handle larger and larger software projects. Historically, most automated techniques have focused strictly on conversion. While these tools can often handle basic paradigm shifts, such as mapping C pointers to Java references [13], they rarely perform deep architectural re-engineering. Early automated tools relied on direct, rule-based mappings known as transliterations. More recently, however, tools have begun integrating large language models (LLMs) [14], which offer greater flexibility in mapping language semantics and can facilitate automated re-engineering.

Regardless of the technique used, language migrations often require the new program to be semantically equivalent with the legacy version. This means that the migrated software must preserve the original's exact business logic and side effects, even if the underlying implementation differs. To verify this equivalence, developers use various validation methods. One common approach is to use of characterisation tests [15], which capture the original system's observed behaviour to ensure that the new version replicates it entirely.

2.3 Software quality

As with software migration, the definition of "software quality" is inherently context-dependent and varies based on the perspective applied. Garvin [16] outlines five views on quality: the transcendental view (quality is universally recognisable, yet undefinable); the user view (quality is fitness for the user's purpose); the manufacturing view (quality is strict adherence to specifications); the product view (quality is tied to the inherent characteristics of the software); and the value-based view (quality is dictated by cost and acceptable pricing).

The field of software engineering originally adapted its quality taxonomies from those used in traditional physical engineering. For example, the early framework introduced by McCall et al. [17] closely mirrored aerospace engineering standards. This framework divided quality into three categories: *Product Operation* (correctness, reliability, efficiency, integrity, and usability), *Product Revision* (maintainability, flexibility, and testability), and *Product Transition* (portability, reusability, and interoperability).

The core elements of this taxonomy remain foundational today, heavily influencing the ISO/IEC 25010:2023 industry-standard product quality model [18], shown in Figure 1. Within this modern standard, high-level categories are termed quality *characteristics*, which

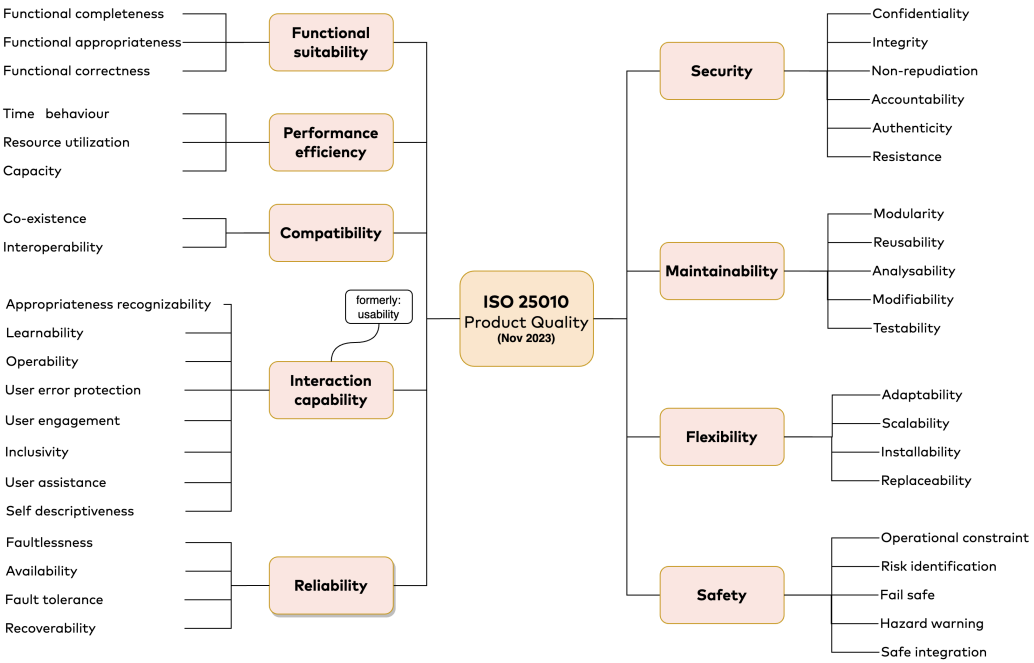


Figure 1: The ISO/IEC 25010:2023 product quality model, with characteristics and subcharacteristics. Image by Arc42.org [19].

are further divided into quality *subcharacteristics*. In academic literature, these terms are frequently used interchangeably with quality *attributes* and *subattributes*.

Throughout this thesis, we evaluate software quality strictly from the perspective of the product view. The ISO/IEC 25010:2023 product quality model is used as the default classification system for all quality characteristics discussed below.

Although a taxonomy provides the necessary vocabulary for discussing quality, it does not prescribe how to measure it. Structured approaches are therefore required to systematically select appropriate metrics. One widely adopted method is the Goal-Question-Metric (GQM) framework, which was developed by Basili and Weiss [20] in 1984. The GQM framework operates across three levels: the *conceptual* level (goals), the *operational* level (questions), and the *quantitative* level (metrics). A measurement program begins by defining specific goals, which are broken down by purpose, quality issue, object, and viewpoint. These goals are then translated into quantifiable questions that characterise the software. The questions dictate the specific data metrics required to answer them.

Once the relevant metrics have been defined, the evaluation of software quality is usually divided into static and dynamic analysis. Static analysis evaluates the software without executing it, often by examining Abstract Syntax Tree (AST) of the source code, which is a formal tree representation of the code's structure. As Binkley [21] notes, while this structural approach is highly scalable and can cover all possible execution paths, it is an over-approximation by nature and may yield false positives. Conversely, dynamic analysis evaluates the software's behaviour during active execution, such as through test

suite coverage. Pezzè and Young [22] characterise the fundamental trade-off between the two approaches as balancing safe (or sound) analysis with precise analysis. Static analysis is generally safe and conservative, guaranteeing that all potential paths are considered. Dynamic analysis, on the other hand, is highly precise, and captures actual runtime behaviour, but is limited to the paths that are explicitly executed. As they compensate for each other's limitations, they are often used together in hybrid analysis to provide a more comprehensive view of software quality.

2.4 Evaluating open-source software quality

Since we use a large number of open-source projects as case studies, it is important to set a precedent for using this type of data. Mining software repositories to evaluate quality is a well-established method in empirical software engineering. This section reviews key literature that leverages open-source data to assess various types of software quality, providing the methodological basis for our approach.

The use of open-source projects for quality analysis has been validated by previous literature. For example, an early comparative study by Mockus et al. [23] demonstrated that open-source development can match or exceed closed-source projects in terms of speed, productivity, and code quality when given the right organisational structure. Leveraging the transparency of these projects, subsequent researchers have mined large amounts of publicly available metadata. Li et al. [24], for example, used a dataset comprising over 29,000 open-source bug reports to systematically identify patterns and characteristics of software defects.

The rise of platforms such as GitHub has significantly expanded the scale of empirical quality analysis. Relevant to the goals of this thesis, Ray et al. [25] used a large dataset of GitHub projects to study the direct impact of specific programming language features on software quality. Similar large-scale approaches have also been used to evaluate the correlation between development practices, such as code review, and the incidence of post-release defects [26].

However, using these large datasets requires methodological rigour. A reproduction study of the work of Ray et al. by Berger et al. [27], highlighted the risks of data noise and sampling bias inherent in GitHub. The authors emphasised that researchers must exercise extreme caution when drawing statistical conclusions from such repositories. This caveat is relevant to our research and informs the data curation and manual verification steps integrated into our study's design.

2.5 Adoption of Rust and its ecosystem

As this thesis examines the migration of open-source projects to Rust, it is important to contextualise our findings with the existing literature on Rust adoption. In this section we review previous studies that have explored the challenges, barriers and benefits that developers have experienced when integrating Rust into their workflows and ecosystems.

Previous research has consistently shown a duality in Rust adoption: there exists a steep initial learning curve, but with significant long-term benefits. Using a mixed-methods approach of interviews and surveys, Fulton et al. [28] investigated the experiences of the Rust development community. They found that, although developers often struggle to learn the language and perceive a lack of mature infrastructure, adopting Rust ultimately has positive impacts, notably increased confidence in code correctness and a more reliable development process. Subsequent research by Zhu et al. [29] corroborates these findings, providing further detail on the specific conceptual challenges that developers face when initially learning Rust's strict memory management paradigm.

Zeng and Crichton [30] explored these barriers to entry further by analysing developer discourse from forums such as Reddit and Hacker News. Based on this data, they formulated several hypotheses as to why Rust can be difficult to adopt in practice. They found that the documentation for Rust's tooling is often less clear than for the language features, that it is difficult to translate common design patterns from other programming languages into Rust, and that developers are reluctant to migrate incrementally due to the perceived high cost of integrating Rust into existing toolchains.

These integration challenges become particularly apparent when examining large-scale adoption efforts. Li et al. [31] analysed the Rust-for-Linux (RFL) project to determine whether its integration had met the project's foundational goals. After reviewing bug reports and developer discussions, they concluded that, although RFL is a positive step towards a more secure kernel, it does not eliminate all vulnerabilities and introduces measurable performance overheads. Furthermore, although the introduction of Rust successfully attracted new contributors to Linux, these developers had not yet fully integrated into the core development team.

2.6 Migration detection and analysis

Accurate detection of software migrations is a prerequisite for their analysis. Previous literature has explored various detection techniques for different types of migration, ranging from library to architectural changes. Reviewing these methods provides context for the automated detection strategy developed in this thesis.

In the context of library migrations, researchers often analyse project metadata. For example, He et al. [32] identified Java library migrations by tracking dependency modifications within `pom.xml` configuration files. However, they found that using metadata alone could be misleading, so they verified their dataset by corroborating it with commit messages and performing a manual inspection. They then used this verified dataset to analyse the frequency, execution, and underlying motivations of these library migrations.

Alrubaye et al. [33] took a different approach to validation. They started similarly by tracking dependency changes in the `pom.xml` files, but then validated the migrations by analysing the source code itself. Specifically, they used Abstract Syntax Trees (ASTs) to map method invocations from the old library to the new one, thereby ensuring that the migration was reflected in the code's structure.

Some studies analyse complex migrations, such as transitioning from monolithic architectures to microservices, by relying on developer input or focused case studies instead of automated mining. For example, Tuusjärvi et al. [34], gathered self-reported migration data by distributing a structured questionnaire to software professionals. While surveys provide high certainty that a migration occurred, they are difficult to scale across thousands of repositories. Alternatively, Lenarduzzi et al. [35] analysed architectural migration through a detailed case study of a single project. Notably, they used SonarQube to measure trends in technical debt before and after the transition. This evaluation methodology directly informs the quantitative metrics applied later in this thesis.

When searching specifically for automated methods to detect programming language migrations, literature is scarce. The closest established field is cross-language clone detection, which identifies code fragments in different languages that implement the same logic. Early techniques, such as that proposed by Cheng et al. [36], use a novel text-matching approach to identify overlapping identifiers and code structures between C# and Java. More recent advances, such as the work of Perez and Chiba [37], applied machine learning, training a long short-term memory (LSTM) network to detect semantic clones across Python and Java.

Although these tools were originally designed to detect duplicate code, they can be effectively adapted for migration detection, since a migrated project is essentially a cross-language clone of its former self. Peters et al. [38] successfully applied a form of Cheng's text-matching strategy to identify Android applications that had migrated from Java to Kotlin. This work is key to proving the effectiveness of our methodology, as it shows that tracking overlapping text and identifiers across different languages is a viable way of detecting language migrations.

2.7 Empirical studies on language migration

While much of the literature focuses on developing automated translation tools, and some on detecting migration, only a smaller body of research considers the practical effects of switching to a new programming language.

For example, the study by Peters et al. [38] identified 62 Java-to-Kotlin migrations and analysed the runtime efficiency of ten sampled projects before and after migration. The researchers used this sample to conduct a statistical analysis to determine whether migrating to Kotlin affects the runtime efficiency of Android apps.

Other studies have also focused on the perspectives of developers, examining why they migrate and the challenges they face. Martinez and Bruno [39] surveyed developers who had migrated Android apps from Java to Kotlin. They identified these projects by searching for commits that replaced a `.java` file with an identical-named `.kt` file. The survey asked about motivations and helped to inform research into improving the experience of migrating from Java to Kotlin.

Some research has been conducted into how these migrations were executed. Focusing on incremental migrations, Mateus et al. [40] analysed Java-to-Kotlin transitions across

1,457 open-source projects in order to identify the incremental migration strategies employed. They then used this data to train a machine-learning model to recommend the optimal sequence of files to migrate.

2.8 Research on Rust migration and translation

Although there are empirical studies on general language migrations, research specifically targeting Rust has mostly focused on automated translation tools, particularly those converting C code to Rust. As outlined in a recent overview by Hong et al. [3] in Communications of the ACM, a substantial body of recent work has been dedicated to bridging the gap between these two system programming languages.

Early foundational models, such as C2Rust[7], work by transliterating C code directly into Rust. However, since C relies heavily on raw pointers and manual memory management, C2Rust primarily produces `unsafe` Rust blocks. Consequently, the tool merely performs a direct translation, leaving the developer responsible for manually refactoring the generated code to satisfy Rust's borrow checker and ensure true memory safety.

To reduce this manual effort, subsequent research has focused on generating safe Rust automatically. For instance, Emre et al. [41] devised static analysis techniques to evaluate C code and map certain patterns onto safe Rust abstractions, thereby reducing the need for human intervention. Recently, much of the research focus has shifted towards using large language models (LLMs) for these translations [14], accompanied by specialised benchmarks to evaluate their accuracy and safety [42].

Research into the translation of higher-level languages into Rust is scarce. Although some open-source translation tools exist, such as py2many[8] for Python, these have not yet been formally studied or evaluated in academic literature. Most importantly, aside from papers presenting automated C-to-Rust conversion tools, there is currently no empirical research analysing the real-world strategies that developers actually use to migrate active software projects from *any* programming language to Rust. This represents a critical gap in the literature, which this thesis aims to address.

[7] C2Rust: https://github.com/immunant/c2rust

[8] py2many: https://github.com/py2many/py2many

Chapter 3

Methodology

This chapter details the mixed-methods research design used to answer the research questions. It outlines the automated techniques developed to identify and verify software migrations on GitHub (RQ1), the setup and execution of semi-structured interviews with project authors to understand their motivations and strategies (RQ2), and the derivation of quantitative software quality metrics based on the Goal-Question-Metric (GQM) framework (RQ3).

3.1 RQ1: Finding migrations to Rust

To answer our first research question, we first establish a reliable and reproducible pipeline for identifying open-source projects that have migrated to Rust. This section explains how we detect and verify these migrations. Firstly, we evaluate different detection strategies and explain our chosen approach. Next, we describe how we collected our initial dataset of GitHub repositories. Finally, we explain the rules we use to find migrations by tracking code identifiers, followed by the manual verification steps required to confirm our final dataset.

3.1.1 Weighing approaches

We evaluated several potential methods for identifying migrations before selecting a detection strategy, weighing their theoretical soundness against practical considerations.

The first approach considered involved analysing the inverse correlation between code additions and deletions across two languages. Specifically, this involved tracking commits with line additions in Rust alongside deletions in the original language. However, this method is highly susceptible to noise and timing discrepancies. This is because migrations are rarely executed within a single commit or a tightly bound timeframe; for example, deletions of legacy code can lag months behind the introduction of Rust code. Furthermore, line-count correlations can be heavily skewed by unrelated repository changes, such as a shrinking front end coinciding with an expanding Rust back end. Finally, code density varies significantly between programming paradigms. For instance, 1,000 lines of Java may equate to 1,000 lines of Rust, whereas 1,000 lines of Go may translate to just 500 lines, rendering direct numerical comparisons unreliable.

The second option was to monitor changes in build systems or CI/CD pipelines. This was based on the assumption that adopting Rust would require the use of Rust-specific tools. The main disadvantage of this approach is that infrastructure is rarely coupled to a single language. Many projects use language-agnostic build systems such as CMake, Nix and Bazel, meaning the underlying infrastructure may not change at all when Rust is introduced. Additionally, this approach would be ineffective for partial migrations where legacy build pipelines are maintained alongside new ones or for repositories lacking a default build system.

A third alternative, which is similar, involves tracking the replacement of dependencies. The hypothesis is that removing a legacy dependency while adding a Rust equivalent indicates migration. However, this assumes near-one-to-one parity between libraries in different ecosystems, which is rarely the case. Furthermore, this method would exclude standalone applications or services that do not act as dependencies themselves. Implementing this approach would also require the creation of parsers for numerous package managers in order to support a wide range of languages.

A fourth approach used Natural Language Processing (NLP) to analyse commit messages, pull requests and documentation for explicit references to migration. Although this seems straightforward, natural language is inherently ambiguous. For example, a commit message stating "migrated to Rust" could imply a full system rewrite, or it might simply mean that the developers upgraded their Rust compiler to a newer version. This method also relies heavily on consistent and descriptive documentation practices among developers, which can vary across projects. Lastly, implementing this approach manually for a large number of projects is very labour-intensive.

Having ruled out analysing the metadata and infrastructure, and decided not to rely purely on the documentation, we concluded that analysing the source code itself offers the most reliable migration signal. By parsing the project's code files' Abstract Syntax Tree (AST), we can inspect both the structural elements and the specific names used in the code. As structural elements depend on the syntax of a specific programming language and its idioms, we expect them to undergo significant change during a migration. However, we expected the names of variables, functions and classes to be more likely to remain the same, as these represent the core features and API contracts of the software.

Therefore, if names were present in an initial version of a language, were subsequently removed, and then appeared exclusively in Rust, this would provide strong evidence of migration. We refer to this replacement as *movement*. However, relying purely on movement has a notable downside in that it will miss projects where the migration is based on a different repository (i.e. a cross-repository migration) or where the old code is never actually removed. To include these cases, we must also consider identifier *overlap*, i.e. names that exist simultaneously in the legacy language and in Rust. While identifying overlap solves these issues, it is more susceptible to noise. For instance, if two different languages within a repository call the same external API, they will share overlapping identifiers even though no migration has taken place.

Despite the potential for noise, we opted for the approach of tracking both identifier movement *and* overlap because it provides the most robust fingerprint of migration. While build systems, dependencies and documentation are heavily influenced by the specific workflow chosen by a development team, the core vocabulary of the software's domain is expected to remain constant between versions or cross-repository migrations. Whether representing an API endpoint, a database model or a highly specific function name, domain concepts endure.

Ultimately, our approach is largely language-agnostic because it is based on tracking these domain identifiers, requiring only an AST parser for the given language. Furthermore, unlike infrastructure- or dependency-based methods, inspecting the code directly provides the precise granularity needed to detect both partial and full migrations.

3.1.2 Collecting the repositories on Github

In order to identify projects that have migrated to Rust, we first compiled a comprehensive dataset of popular GitHub repositories. This data was collected using the GitHub Search API (`/search/repositories`). As the API restricts searches to a maximum of 1,000 repositories per query, we implemented a partitioned search strategy based on repository star count.

We started with a baseline of 300 stars and iteratively queried the API, setting the minimum star threshold to the highest star count in the previous batch. Crucially, we traversed the star counts in ascending order to mitigate the risk of missing repositories whose star count increased during the data collection process. Searching in descending order could cause a repository gaining stars to jump out of an upcoming search window and into an already-processed one, effectively omitting it from the dataset. Although an ascending traversal requires post-collection duplication, it ensures a more complete and reliable dataset.

With a lower bound of 300 stars, we can find popular GitHub repositories while also benefiting from the convenience of sitting at the limit of GitHub's API constraints. Lower numbers of stars give query results of over 1,000 repositories per specific star count. At that point, a secondary partitioning criterion would be required (such as creation dates), which would inefficiently increase the number of API calls required. As an alternative to the GitHub API, we considered extracting repositories by parsing star events directly from the GitHub Archive. However, processing the database of over 10 billion records [43] was deemed unnecessarily resource-intensive compared to our API approach.

From this initial dataset, we extracted a target subset of projects containing Rust code. However, as the standard Search API only returns a repository's primary language, projects in which Rust was introduced alongside a larger legacy codebase would be missed. Therefore, we queried the `/repos/{owner}/{repo}/languages` endpoint for every repository to obtain a language breakdown. This endpoint uses GitHub's Linguist tool, which calculates language distribution based on compiled binary size rather than lines of code using different strategies. It excludes documentation, generated code, and vendored files from its calculations.

To prevent the inclusion of trivial projects, such as minor build scripts or experimental files, we have set a minimum threshold of 1% Rust content. This ensures that the presence of Rust is substantial enough to represent a potential migration. Repositories that meet this criterion constitute what we will refer to as the **Rust set** from now on.

3.1.3 Finding migrations that occurred within a repository

The central hypothesis for detecting an **in-repository** repository migration is that domain-specific names initially written in a legacy language A will eventually transition exclusively to a new language B. We define this phenomenon as **movement**: *when a normalised identifier is present in language A and not in language B at time X, and is exclusively present in language B at time Y, where Y is later than X and languages A and B are different*.

This approach is based on the idea that, although syntax and architecture may change during migration, the core vocabulary–such as domain concepts, API contracts, and function names–remains largely consistent in order to preserve functionality. To account for different naming conventions between languages (e.g. camelCase versus snake_case), identifier matching is normalised. For instance, the Java function `doSomeAction` is considered equivalent to `do_some_action` in Rust.

Based on this concept, we hypothesise that migration is likely to have occurred if the maximum number of identifiers moved between two languages exceeds a defined threshold. We formally define a function, $\mathcal{M}(\theta)$, which represents the number of moved identifiers for a specific configuration, $\theta = (t_1, l_1, t_2, l_2)$, where l_1 and l_2 represent the languages present at times t_1 and t_2, respectively. If $\mathcal{L}(t)$ denotes the set of languages present in a repository at time t, then $\mathcal{M}$ is evaluated under the constraints that $l_1 \in \mathcal{L}(t_1)$, and $l_2 \in \mathcal{L}(t_2)$, with $t_1 < t_2$.

However, as we have previously explained, relying purely on movement is insufficient for detecting all migrations. For example, partial migrations or projects that maintain the legacy version alongside the newer Rust implementation will not show movement. To include these instances, we use the weaker concept of **overlap**, defined as follows: *when a normalised identifier is present in language A at time X and simultaneously present in language B at time Y*. As with movement, we define a function, $\mathcal{O}(\theta)$, that calculates the number of overlapping identifiers for a configuration, θ, with the same constraints applied to $\mathcal{O}$.

To analyse the repository's history efficiently, we sample 100 commits distributed evenly across the entire commit history of the repository's default branch (e.g. `main` or `master`) instead of including all of its commits in our analysis. This sample size was chosen to strike a balance between computational feasibility and the size of the time gaps between the commits that were sampled. As large-scale migrations typically unfold over months or years, we assumed that a sample of 100 commits provides sufficient granularity to detect structural shifts.

For each sampled commit, we extract identifiers by parsing the AST of all the source code files. We perform a Depth-First Search (DFS) across the syntax tree using the `tree-sitter` parsing library [44], which we have configured to support the 23 most popular

programming languages on GitHub. During this process, we extract the text values of any syntax node with a type containing 'identifier' or 'name'. To capture structural context, we also include the record syntax types of the surrounding parent and grandparent nodes in our dataset.

To implement the previously mentioned normalisation strategy, the extracted text values are converted to lowercase and have underscores removed. We also enforce a minimum length threshold of four characters because shorter identifiers, such as single-letter loop variables like `i` or `x`, are highly prone to random collisions and unlikely to represent the domain-specific concepts required for migration.

Due to the vast size of this dataset and the fact that data collection is carried out in parallel across multiple threads, keeping the full AST data of 100 historical commits in memory simultaneously would result in a resource bottleneck. To ensure that this process is both scalable and memory-efficient, we have implemented several optimisations.

Firstly, repository handling is optimised by cloning bare repositories and checking out specific commits using the command `git -C archive --format=tar`, which our tests showed to be significantly more efficient than manipulating full Git working directories. Secondly, rather than storing extracted identifiers in memory, the data is streamed directly to a compressed archive on disk for each time point and categorised by programming language. Once the extraction phase for a repository is complete, the files are loaded incrementally. As identifiers are loaded, they are immediately hashed as 64-bit integers and ingested into a set in order to reduce resource consumption and improve runtime efficiency. Since we only use 64-bit hashes, there is a possibility of collisions, with a probability of around 0.03 percent when 100 million identifiers are hashed. Here, we decided that increasing the hash size was not worth the additional resource cost because, with such a low collision probability, a single collision would only increase the $\mathcal{M}$ result by one. Alongside AST parsing, we use the tokei[9] tool to track the distribution of lines of code (LOC) per programming language at every sampled commit.

3.1.4 Finding migrations based on other repositories

As we previously explained, restricting our analysis to internal migrations means overlooking instances where a migration is developed in a separate repository. This scenario, which we refer to as a **cross-repository** migration, often arises when a migration is created by different authors or when maintainers prefer to keep the new Rust implementation organisationally distinct from the legacy codebase.

To detect cross-repository migrations, we apply the same foundational hypothesis regarding domain vocabulary, but restrict our analysis to the weaker property of identifier *overlap*. The stricter *movement* is inapplicable in this context because we only examine the current state of repositories; we do not know whether old identifiers have been removed from the original repository. Collecting identifiers for all non-Rust repositories at multiple points in time would not have been computationally feasible within the scope of this project. Furthermore, since cross-repository migrations are usually independent and longitudinal

[9]tokei: https://github.com/xampprocky/tokei

tracking is less relevant, we can optimise our data collection by using a shallow clone (`--depth 1`), which only downloads and checks out the most recent commit of each repository.

However, cross-repository comparison still poses a significant computational challenge. Our dataset contains a large number of repositories, each of which could potentially be the origin a cross-repository migration to a Rust repository. Calculating the exact identifier overlap between every Rust repository and all potential non-Rust origin repositories would result in a combinatorial explosion. Conducting string comparisons for 100,000 repositories, each containing 100,000 identifiers, for example, would require approximately 10^{20} comparisons, which is computationally infeasible.

To resolve this computational issue, we apply the MinHash algorithm, which was introduced by Broder [45]. MinHash reduces the size of the sets by generating a fixed-size signature of the hashes of a set, while approximately preserving the Jaccard similarity coefficient (J) between two sets A and B, as defined in Equation 1. This enables us to approximate the similarity between two large codebases by comparing their hash signatures alone.

$$J(A, B) = \frac{|A \cup B|}{|A \cap B|} \tag{1}$$

To achieve this more efficiently, we use the SuperMinHash algorithm [46], which delivers faster runtime execution and lower variance than the original MinHash algorithm. For each project and programming language pair, we generate a signature of 1024 64-bit hashes. For example, with these parameters and for comparisons with a true Jaccard similarity of up to 15%, the upper bound of our estimate's standard deviation is 0.8% for unions of up to 1,000 identifiers.

Once again, given the scale in terms of the number and possible size of repositories, the extraction and comparison pipeline was optimised for resource and time efficiency. Identifiers are parsed using the AST approach from the previous section, but only for the most recent commit in each repository. We again only collect identifiers that are longer than four characters. Once all the data has been collected, we apply our hashing step, immediately ingressing each loaded identifier into the MinHash signature. These signature arrays are then stored in a custom binary format, enabling efficient deserialisation during the comparison phase.

Due to the binary format, each project is now around 64 KB in size, meaning the entire dataset can be loaded into memory at once. We can then find the closest matches for the languages in other repositories, excluding Rust, for each repository containing at least 1% Rust. Each match provides a Jaccard similarity score based on the previously described MinHash property.

Additionally, in our later analysis we require a minimum of 10 KB in Rust to prevent trivial programs from being included although Rust makes up over 1% of the entire project.

3.1.5 Dataset selection

Having established the metrics for tracking identifier movement, internal overlap and cross-repository similarity, the next step is to define the specific heuristic used to classify a

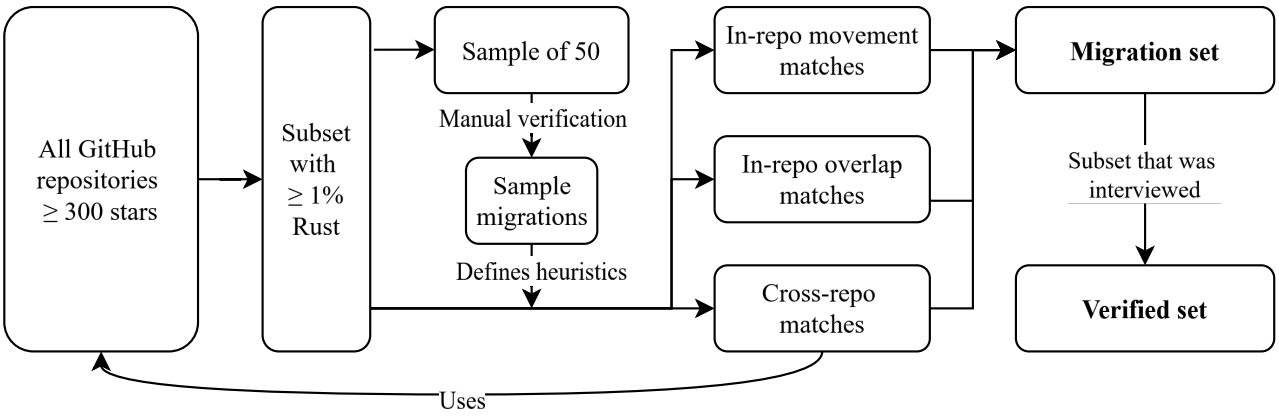


Figure 2: An overview of the steps we take for detecting migrations.

repository as having undergone a migration. As our detection techniques capture different migration strategies, three distinct inclusion criteria have been established. A repository is flagged as a potential migration if it exceeds the defined boundary value for either the maximum internal movement ($\mathcal{M}$), the maximum internal overlap ($\mathcal{O}$), or the cross-repository Jaccard similarity (J).

As tracking identifiers is a novel approach to detecting language migration, we have included a manual verification step to ensure the validity of our dataset and filter out false positives. While significant identifier *movement* is hypothesised to be a highly reliable indicator of migration, identifier *overlap* is inherently noisier. There are simply more cases in which identifier overlap shows up as a false positive for migration, such as in a single repository housing both a Rust backend API and a frontend consumer, or in a project maintaining cross-language bindings.

To mitigate this issue, any candidate flagged due to internal overlap or cross-repository similarity is corroborated by explicit textual evidence from the repository. We systematically search the repository's metadata (including commit messages, pull requests and documentation) for declarations of migration using the keyword pattern `(migration | rewrite | written | replacement | implementation) AND rust`.

To determine the boundary values for these heuristics, we conducted an empirical calibration phase. We randomly sampled 50 repositories from our *Rust set* and subjected them to a manual verification process, requiring documented proof of migration in order to classify them as true positives. From this subset of projects that were successfully verified as true migrations, we extracted the minimum observed values for movement, overlap, and Jaccard similarity. Using these minimum values as conservative, empirically grounded lower bounds reduces the risk of arbitrarily excluding subtle migrations when applying these thresholds to the full *Rust set*.

These bounds define the final inclusion logic as follows: a project is selected if it shows sufficient identifier movement, sufficient internal overlap coupled with documented evidence, or sufficient cross-repository similarity coupled with documented evidence. The

repositories that satisfy these criteria form our final dataset, hereafter referred to as the **migration set**. We present an overview of the previous steps in Figure 2.

3.1.6 Characteristics collection

To better understand the context of the migrated projects, we collect several key characteristics of each repository. Firstly, we identify the *original language* as the non-Rust language from the detected migration pair. To protect the anonymity of the projects when using these groups for qualitative interviews, we structurally group similar programming languages and aggregate the least frequently occurring languages into a consolidated 'Other' category.

Additionally, since the type of software can affect how a migration is executed, we manually assign each repository to an *application domain* using the taxonomy developed by Borges et al. [47].

Finally, we used the GitHub API to collect four metrics at the time of data collection: the number of *stars* and *forks* to measure project popularity and developer engagement; the *size* of the codebase in bytes; and the percentage of the codebase written in Rust. Since comparing raw numbers across different project scales can be difficult, we categorise these metrics. We apply K-means clustering to divide each of the four characteristics into three distinct tiers, which we refer to as small, medium, and large throughout the analysis.

3.2 RQ2: Interviewing project authors

Although detecting repositories provides broad, quantitative insights into language migrations, it does not fully capture the human decision-making processes behind them. To answer our second research question, we conducted a qualitative case study with project authors using semi-structured interviews. This approach helped us understand the motivations, strategies, and perceived outcomes of migrating to Rust, details that would be difficult to extract from the source code alone.

We targeted the core developers and maintainers of the projects in our *migration set*. We collected their email addresses from the repository metadata, prioritising the following sources in order: the repository owner's GitHub profile, the `README` or root manifest, and, finally, contact details found in other relevant manifests within the repository. To respect the developers' time and avoid sending spam, we sent a single email invitation with no follow-up reminders. Before scheduling an interview, participants were required to submit an informed consent form detailing how their data would be used and stored, and their right to retract their responses or withdraw.

3.2.1 Setup of the interviews

We conducted semi-structured interviews, using a mix of mostly open and some closed questions. As Wohlin et al. [4] notes, this format is effective in empirical software engineering research because it ensures that we achieve our research goals while allowing participants to elaborate on their specific contexts and challenges.

Table 1: The set of goals during the interviews, combined by phase.

Background	Responsibilities of the interviewee during the migration.
	Experience of the team with Rust.
	Scope of the migration.
Motivation	Events leading up to the migration.
	The reasons for choosing Rust.
	How they wanted their software to change.
Strategies	Which strategies they applied.
Outcomes	*If they (mostly) completed their initial migration goals*,
	Which changes they experienced, specifically for the expectations mentioned earlier.
Reflection	The difficulties they experienced during migration.
	What they would do differently if they were to do the migration again.
	The advice they would give to projects considering migrating to Rust.

Each interview had a target time limit of 30 minutes, and the interviews were conducted remotely using Microsoft Teams. Due to the limited time allotted for each interview, it was divided into five phases, as outlined in Table 1. Firstly, we established the participant's background and the scope of the migration. Then, we discussed their motivations for choosing Rust. Next, we discussed the strategies they used and evaluated if the outcomes met their initial expectations. Lastly, we asked the participants to reflect on the process, including any challenges they encountered, and to provide advice to future adopters. We systematically progressed through these topics, one by one.

We followed the best practices set by Strandberg et al. [48] by automatically transcribing the interviews using Microsoft Teams. This transcription is cleaned up manually and pseudonymised, which involves removing references that identify the project, the participant, or persons named by the participant. To verify our data, we used *member checking*: we sent the transcripts to the interviewees, giving them two weeks to review, clarify, or retract any statements. Afterwards, the transcripts are dereferenced, meaning we did not store which participant gave which transcript.

We applied an iterative thematic analysis approach to extract insights from the transcripts. Using QualCoder [49], we began *open coding* to tag quotes relevant to our interview objectives, such as specific technical challenges or performance outcomes. We grouped these initial codes into broader themes through multiple rounds of review, following an inductive thematic analysis approach. This structural approach enabled us to synthesise the diverse experiences of individual developers into generalisable findings about the Rust migration process.

3.2.2 Using the interview results

This qualitative phase also strengthens our quantitative pipeline because we can be certain that a migration to Rust occurred for projects for which we successfully interviewed the authors. We refer to this subset as the **verified set**.

3.3 RQ3: Measuring quality changes

Although the qualitative interviews conducted for RQ2 offer valuable insights into how developers perceive outcomes, self-reported data is inherently subjective and prone to bias. To address this limitation, we used a mixed-methods triangulation strategy to answer our third research question. We systematically collected metrics from our *verified set* of repositories to objectively measure changes in software quality before, during, and after migration to Rust. Rather than selecting arbitrary quality indicators, we evaluated metrics that are directly derived from the primary motivations and goals highlighted by the authors during the interviews. This ensures that our quantitative analysis remains relevant to the real-world drivers of Rust adoption.

3.3.1 Deriving metrics from interview motivations

We group the most frequently cited motivations for migrating to Rust into specific software quality goals. Then, we operationalise these goals using the Goal-Question-Metric (GQM) framework, which was established by Basili and Weiss [20] and further detailed by van Solingen and Berghout [50]. Within this framework, we define a broad objective, formulate specific questions to assess it, and select literature-backed metrics to provide quantitative answers. For instance, if interviewees frequently cite "reducing code duplication" as a core motivation (Goal), we would ask how much duplication is currently present in the code (Question), and select *% of Duplication* as our measurable indicator (Metric). Once potential metrics have been identified through the GQM framework, they are filtered against certain criteria to ensure both feasibility and scientific validity.

Firstly, we only use static analysis metrics and explicitly exclude dynamic analysis or project-specific performance benchmarks. Although dynamic analysis provides valuable runtime insights, executing historical versions of software is notoriously unreliable due to the so-called "buildability problem" [51]. As old dependencies become deprecated and build tools evolve, it becomes increasingly difficult to compile older commits reliably for all projects. While some projects may still compile, this would introduce bias into these metrics with regard to languages that are less susceptible to this problem. Furthermore, it was not feasible within the scope of this project to implement project-specific metrics for all projects in the *verified set*. This is especially the case because these benchmarks would have to work over the lifetime of the project to create comparable statistics, which would likely lead to multiple benchmark implementations per project. Additionally, this would strengthen the connection between a project's quantitative and qualitative results, which conflicts with our ethical constraint of anonymity.

Secondly, the selected metrics must be conceptually valid across different programming paradigms. This immediately excludes language-specific heuristics, such as the Java-centric Readability metric [52], or strictly Object-Oriented metrics like Weighted Methods Per Class (WMC) and Lack of Cohesion of Methods (LCOM) [53]. However, comparing even language-agnostic values across a migration can introduce a significant threat to construct validity for some metrics. For these metrics, we mitigate this issue by evaluating relative longitudinal trends and observing whether a metric's trajectory improves or degrades over time. This adapts the methodology successfully employed by Lenarduzzi et al. [35], who used continuous metric tracking to evaluate technical debt before, during, and after architectural migrations.

Lastly, we exclude goals that are coded for a single project only because their results would create a direct reference to the motivating project, thereby deanonymising it. Furthermore, these goals are the least relevant under our approach, as they were only mentioned once.

3.3.2 Tooling and metric collection

We use a combination of established static analysis tools and our own scripts to collect the derived metrics for our verified set.

Firstly, we used SonarQube[10] Server Enterprise Edition, version 2026.1, which is an industry-standard platform supporting a wide range of programming languages. Although SonarQube's "Technical Debt" metric relies on proprietary remediation heuristics, it has previously been accepted as a reliable proxy for code maintainability [35]. However, as the Enterprise edition can only analyse 1 million lines of code at a time, this tool is not suitable for every repository.

Secondly, we employed Mozilla's open-source tool, `rust-code-analysis` (RCA) [54], which supports a wide range of languages and metrics. As RCA is easily augmented, we extended it to support all the languages in our verified set, as well as adding support for our own custom metrics.

Furthermore, we collected available project metadata using the GitHub API. Due to the limitation of 1000 results per query, we again applied a partitioning schema to collect all the issues, pull requests and other metadata. This time, we applied only time-based partitioning.

Lastly, we implemented our own libraries for metrics that could not be extracted using off-the-shelf software. The implementation details of these tools, and the metrics they collect, will be discussed later in the thesis.

3.3.3 Sampling and phase segmentation

To capture the longitudinal trends of these metrics, we took samples from the repositories over time. Again, analysing every single commit in each project's history would be computationally infeasible, so we sampled the repositories at 250 commits distributed evenly across each project's history. This does not apply to metrics that analyse commits

[10]SonarQube: https://www.sonarsource.com/products/sonarqube/

or those that do not analyse code. The datasets generated by our tools are initially stored in their raw formats and are later aggregated into time series.

These longitudinal results are segmented into distinct phases to perform a comparative analysis: before migration, during migration and after migration. For metrics capable of distinguishing between Rust and non-Rust code at a single data point, this is also indicated for the latter two phases.

These boundaries are defined by timestamps, which are determined manually for each verified project. The boundary between "before" and "during" is marked by the exact commit at which the Rust code is first introduced for the migration. The boundary between "during" and "after" is defined by the completion of the migration. If an author confirms in their interview that the migration is finished, we select the timestamp of the first release after the mentioned or found migration date. Not all projects adhere to semantic versioning. Therefore, if there is no clear release indicating the migration's completion, we map the boundary to a different repository milestone, such as a pull request to the default branch. Crucially, if the interview reveals that a migration is ongoing, the project is analysed only across the "before" and "during" phases.

The above mapping only applies to in-repository migrations. For cross-repository migrations, the timeline is mapped across both projects. We indicate the entire original repository as "before migration" and the new Rust repository as "after migration".

3.3.4 Statistical analysis

Once the time series data has been collected and aggregated, and the phases have been identified, we conduct a statistical analysis to determine whether the introduction of Rust has caused a significant change in software quality. To this end, we apply one of three methods to establish whether there was a significant change.

The "recent" method takes the last 15 data points in a phase and applies the two-sided Mann-Whitney U test. We conclude that the change is significantly increasing or decreasing when $p < 0.05$. We apply this test because our metrics do not necessarily follow a normal distribution. The "all" method performs the same test, but on all the data points in a phase.

Lastly, the "trend" method performs a linear regression on each phase and compares the two resulting slopes using a two-sample Z-test (Equation 2). Here, we use the standard errors of our linear regression (SE) and the slopes β, to then calculate our
p-value $= 2 \cdot (1 - \varphi(Z))$, where φ is the standard normal cumulative distribution function. We again apply the criterion that a trend change is significant when $p < 0.05$.

$$Z = \frac{|\beta_a - \beta_b|}{\sqrt{\mathrm{SE}_b^2 + \mathrm{SE}_a^2}} \tag{2}$$

Our results show how many projects saw an increase and how many saw a decrease for each metric. We did not provide any aggregate statistics on the entire project set, such as whether there was an overall statistical increase or decrease, because our set does not necessarily represent all migrations. As Berger et al. [27] demonstrate, we must be cautious when conducting such analyses.

We further analysed the results of these metrics per goal, defining an improvement as a result that matches its goal. For example, the goal of reducing code duplication is achieved when there is a measured decrease in code duplication. Additionally, we conducted an analysis based on the characteristics collected earlier in the process.

Chapter 4

Projects that migrated to Rust

This chapter presents the findings of our automated detection of software migrations to Rust, thus answering the first research question. From an initial baseline of over 173 thousand repositories, our analysis identified 5,786 containing Rust code and verified these to create a set of 285 migration projects. We then discuss the characteristics of these projects and conduct further analysis to improve our detection method.

4.1 Collected projects

The repository collection was carried out on 5 December 2025, providing a baseline dataset comprising 173,806 GitHub projects that had received more than 300 stars. Of this initial set, 6,492 projects (3.7%) were found to contain at least some Rust code. Applying our previously established threshold, 5,786 (3.3%) contained at least 1% Rust. The distribution of these repositories relative to their star counts is shown in Figure 3.

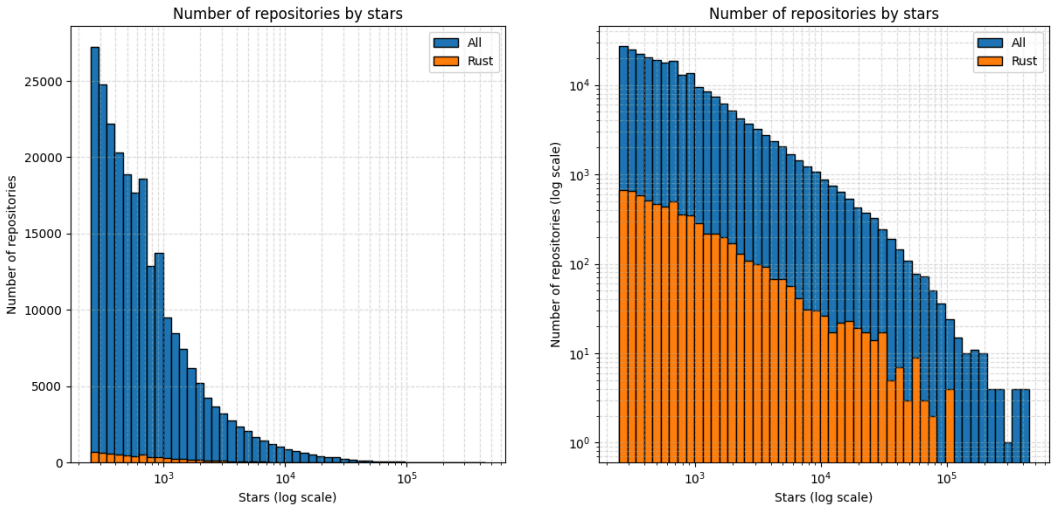


Figure 3: Histograms showing the distribution of repositories by number of stars, with the histogram for repositories containing at least 1% Rust overlaid on top of all the projects.

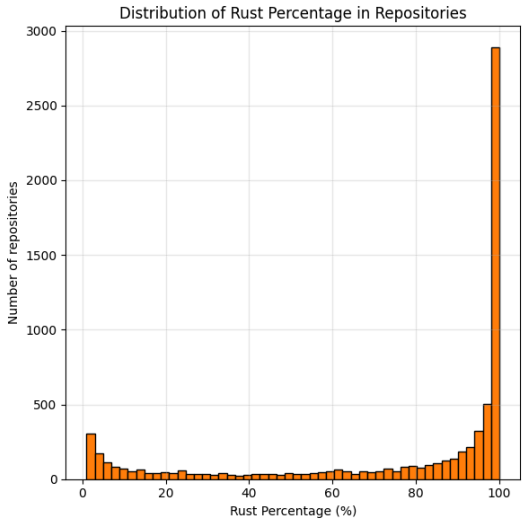


Figure 4: Histograms showing the percentage of Rust, with each bin representing 2%.

In Figure 3, the data shows a bimodal distribution, peaking at 98-100% Rust and with a lower peak at under 5% Rust. This suggests that, in addition to projects that are almost entirely written in Rust, there are also a significant number of projects that contain only a small amount of Rust, for instance in the form of isolated components such as Foreign Function Interface (FFI) bindings. Aggregating this data, we found that 4,423 projects (69.7%) are majority Rust (consisting of 50% or more Rust code) and 1,083 projects (16.7%) are 100% written in Rust.

4.2 In-repository measurements

In order to detect internal repository migrations, we analysed the longitudinal movement and overlap of identifiers across the 5,786 projects that contained at least 1% Rust code.

For each project, we identified the maximum movement pair (t_1, l_1, t_2, l_2). This tuple represents the largest number of identifiers that were present in an origin language (l_1) at time t_1 and subsequently transitioned exclusively to the target language (l_2) at time t_2. Under our methodology, the target language is defined as Rust, and where $t_1 < t_2$.

The logarithmic distribution of these maximum movement pairs is shown in Figure 5. The stacked bars are ordered from most to least common origin language (l_1). The distribution reveals a clear pattern: for projects with lower volumes of moved identifiers, high-level language such as JavaScript, TypeScript, and Python constitute the vast majority. However, for projects with higher volumes of moved identifiers, C has the largest share of the distribution. This suggests that migrations from C often involve large-scale rewrites to Rust, whereas migrations from these high-level languages involve migrating smaller, more specific components to Rust.

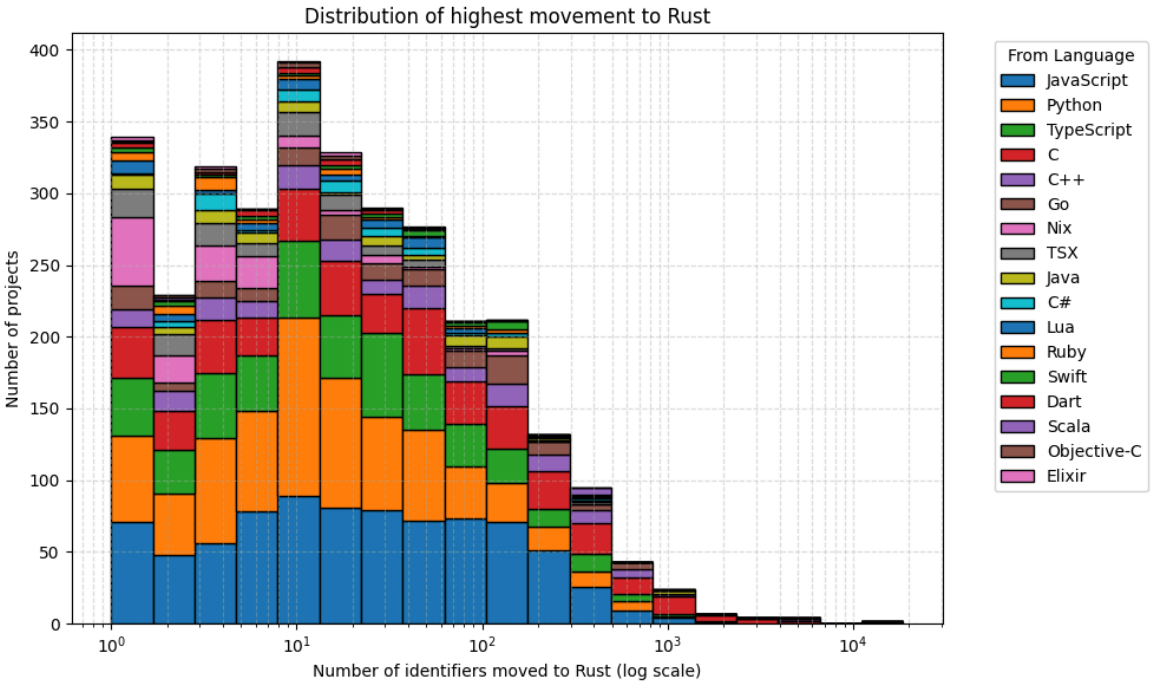


Figure 5: Stacked bar histogram showing the number of projects categorised by their maximum number of moved identifiers and the origin programming language.

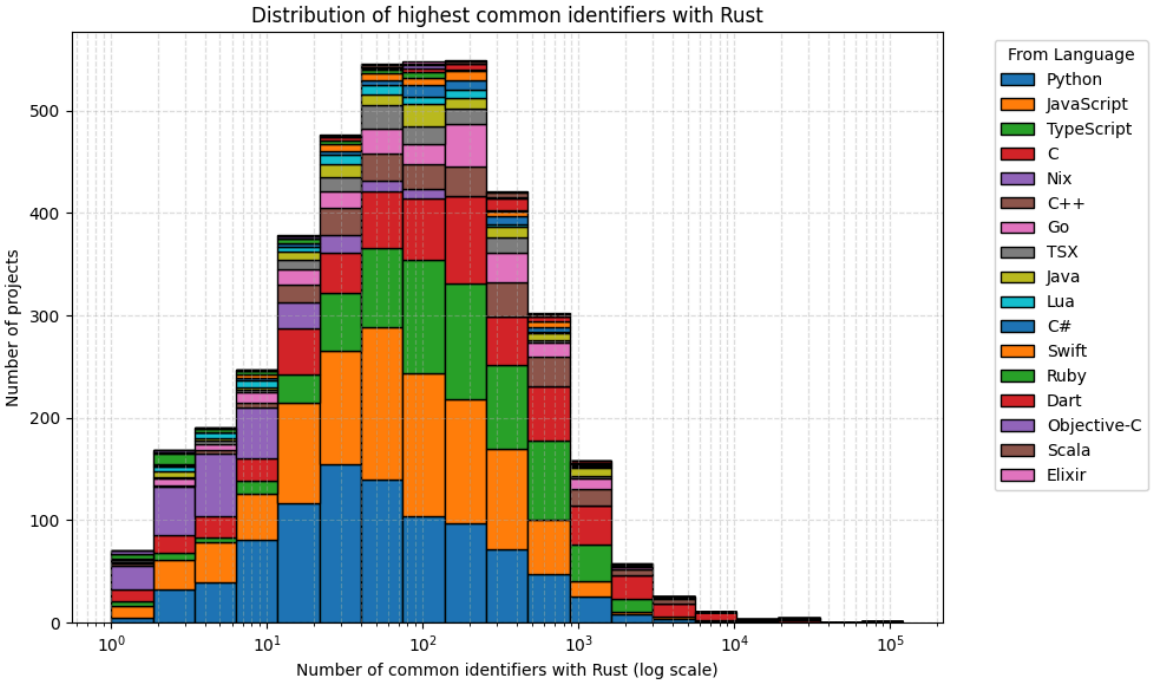


Figure 6: Stacked bar histogram showing the number of projects categorised by their maximum number of overlapping identifiers and the origin programming language.

Using the same constraints, we also calculated the maximum overlap pair for each project, as shown in Figure 6. Compared to the stronger movement property, the overlap distribution shows a notably higher mean.

The overlap language distribution closely mirrors the movement language distribution. While Python, JavaScript and TypeScript are the dominant origin languages for projects with fewer than 1,000 overlapping identifiers, while C is most frequent in repositories with higher amounts of overlap. This reinforces our observation that C-to-Rust migrations are the most extensive within our dataset.

4.3 Cross-repository measurements

To identify cross-repository migrations, we compared the extracted Rust identifiers from each repository in our *Rust set* with the non-Rust identifiers of other repositories using our MinHash proxies. For each Rust repository, we then get a single pair that had the highest Jaccard similarity. The distribution of these top-matching pairs is shown in Figure 7, with the histogram stacked according to the programming languages of the matched other repository.

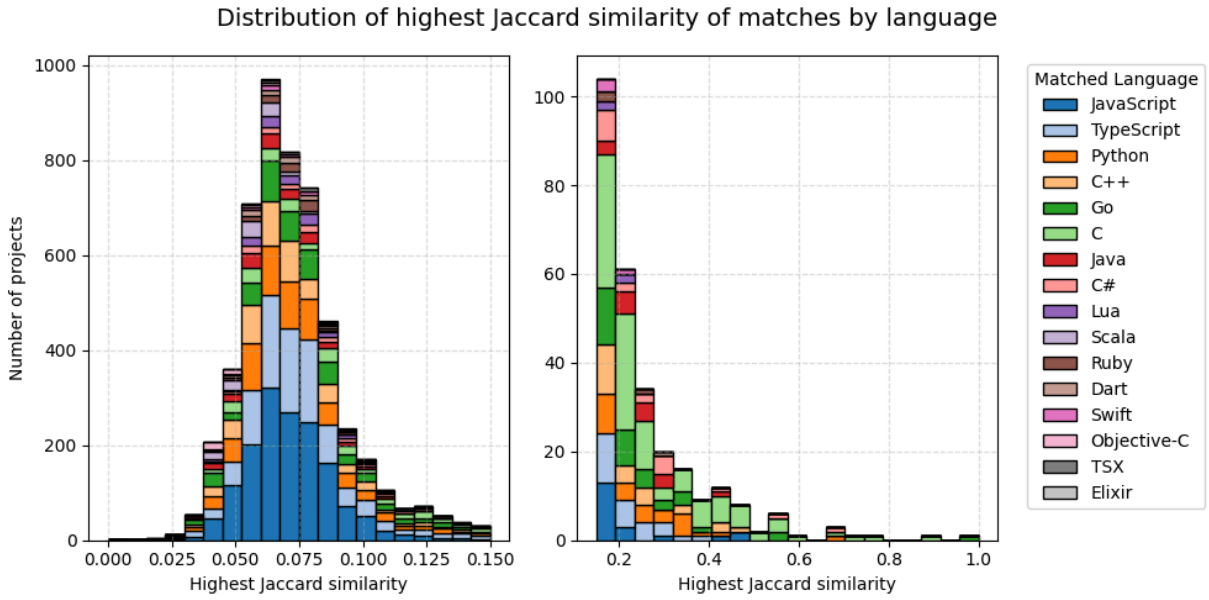


Figure 7: Stacked bar histogram showing the number of projects categorised by the highest Jaccard similarity of a matching other repository and the matched programming language. The overall distribution shows that matches with JavaScript, TypeScript, and Python are the most frequent in absolute volume. However, if we look at the upper tail of the distribution, which are pairs with higher Jaccard similarity, we find that matches with C, C++ and notably Go become more dominant. This suggests that, although there is a high volume of higher-level languages that overlap slightly with Rust projects, the largest amount of overlap originates from the system-level languages C and C++, as well as Go. These distributions are shown more clearly Figure 8. This figure shows that the distributions for Go, C and C#

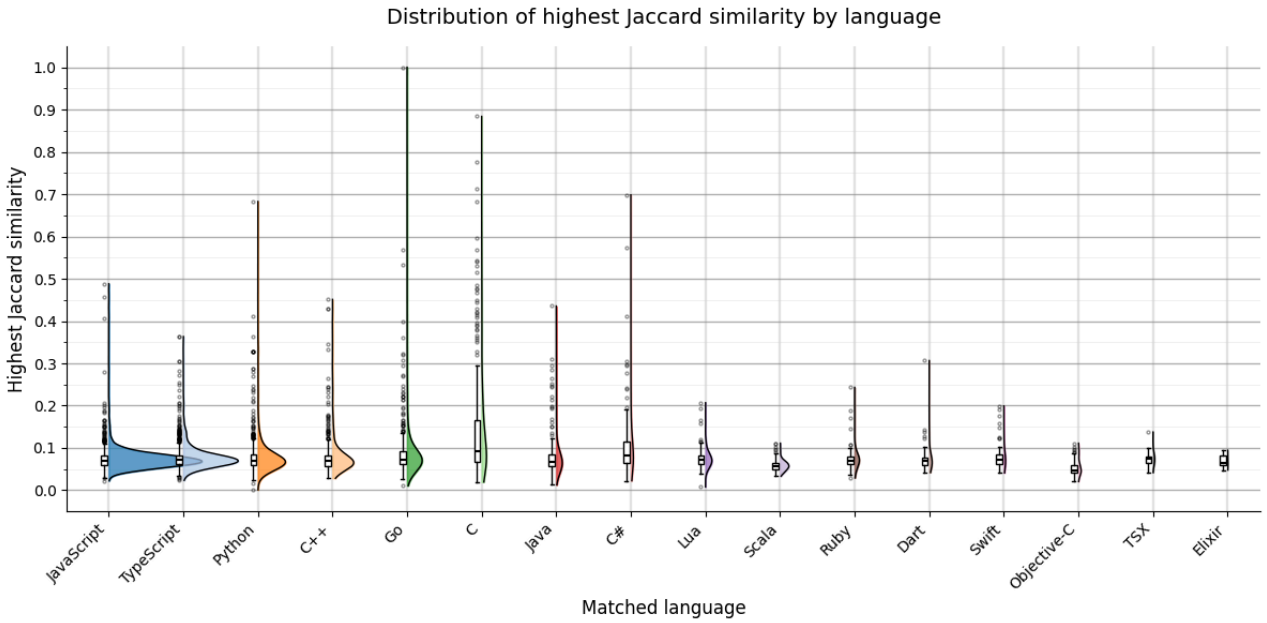


Figure 8: Violin charts showing the distribution of projects according to the highest Jaccard similarity per matched programming language, including a box plot. Please note that the curve is scaled by the relative number of projects.

have a higher mean, while those for JavaScript, TypeScript and Python have a lower mean and smaller variance.

A manual inspection of the outliers, which we say are pairs with a Jaccard similarity higher than 50%, shows two causes for these similarities. They occur either because both repository implement or wrap the exact same external API, or because the Rust repository is a direct, close migration of the original project.

4.4 Final selection and verified set

In order to establish empirically grounded boundaries for migration detection, we applied our manual verification method to a random sample of 50 projects from our *Rust set*. Within this sample, we successfully identified and verified four in-repository migrations. Figure 9 shows a Sankey diagram illustrating identifier movement within the `numaflow` repository. The visualisation shows that, while most Go identifiers remained static across versions, a subset moved exclusively to Rust, indicating a clear partial migration.

As explained in the methodology, to avoid excluding subtle or partial migration from our dataset, we used the minimum values observed in these four verified projects as a conservative baseline. This established lower bounds of 71 moved identifiers and 124 overlapping identifiers. When these thresholds were applied to the broader dataset, projects that were highly unlikely to contain migrations were excluded, successfully narrowing the initial search to 1,113 feasible in-repository candidates.

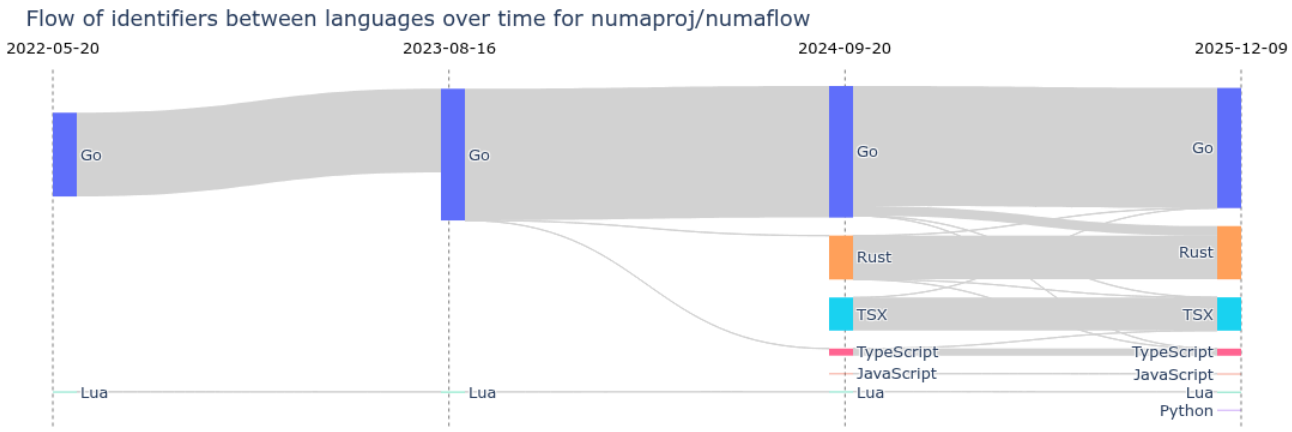


Figure 9: Sankey graph showing the movement of identifiers across project versions in `numaflow`. Note that this visualises only the subset of identifiers that overlap between versions, rather than the entire codebase.

Using the same methodology for cross-repository migrations, we identified three verified migrations within the initial sample of 50 projects. The lowest Jaccard similarity among these three was 8.7 percent. Applying this minimum overlap bound to the entire *Rust set* yielded 1,218 cross-repository migration candidates.

Combining the in-repository and cross-repository approaches yielded a total search space of 2,131 candidate projects, accounting for those identified by both methods. Following manual verification of these candidates, a final *migration set* of 285 projects was identified. Of these, 235 were classified as in-repository migrations (giving an initial detection precision of 21.1%), and 109 as cross-repository migrations (giving a precision of 8.9%).

4.4.1 Interview outreach

To compile our *verified set*, we contacted the authors of the 285 identified migrations via email. We found valid email addresses for 259 projects (91%), and 60 developers responded, giving us a response rate of 23%.

Of those who replied, only three developers (1% of those contacted) stated that no actual migration had taken place. Despite two of these projects showing significant movement of identifiers to Rust, and the third explicitly claiming in its documentation to be "based on" an older project, we trusted the authors' claims and excluded them from further analysis. Seven other developers declined the interview, mostly due to lack of time or because the migration had happened too long ago to remember clearly.

The remaining 50 developers agreed to participate. Ultimately, we successfully conducted 38 interviews: 34 via Microsoft Teams and four via email after the authors indicated that video calling was not possible for them. The other 12 developers either

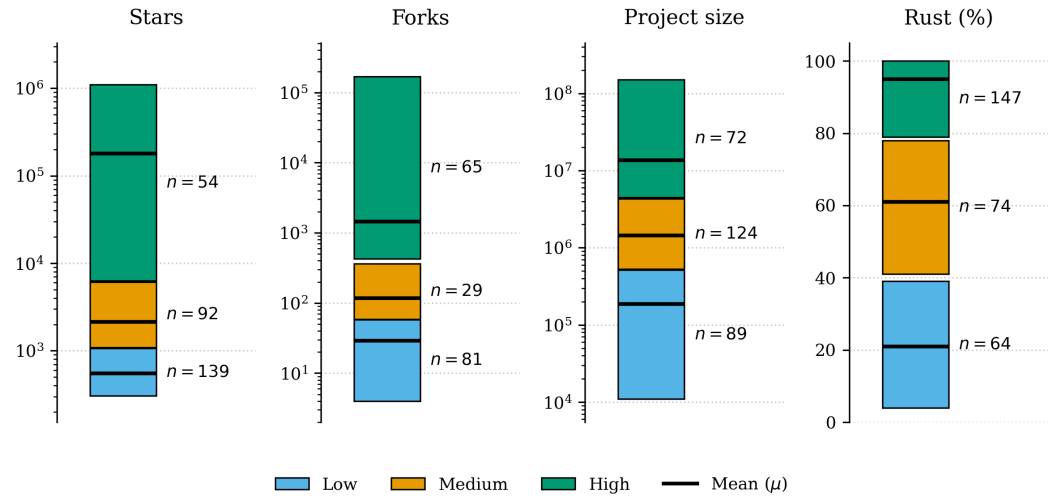


Figure 10: The K-means clustering boundaries for the four continuous project characteristics. Note that the scales for stars, forks, and repository size are logarithmic, while the Rust percentage scale is linear. The black line shows the mean for each group.

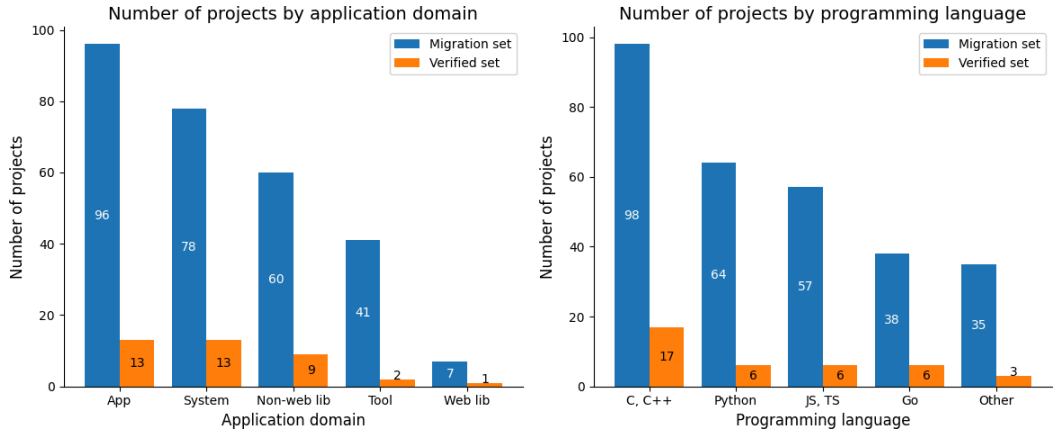


Figure 11: The number of languages per application domain or programming language, for both the migration set and verified set.

stopped responding or missed their scheduled time slot. The 38 projects that we successfully interviewed make up our final *verified set*, with a complete list provided in Appendix A.

4.5 Collected characteristics

In order to provide a clearer picture of the scale and scope of the migrated projects, we analysed four characteristics across the final migration set: star count, fork count, repository size, and the proportion of the codebase written in Rust. As outlined in the methodology, we used K-means clustering to categorise these metrics as “small”, “medium” or “high”. The resulting distributions are shown in Figure 10. The data shows that, for the stars and forks metrics, the high group covers a wider range but contains fewer projects. The reverse

is true of the percentage in Rust: the high group is smaller and contains a higher proportion of projects.

In addition to continuous characteristics, we also categorised the projects by their application domain and their original programming language (Figure 11). This figure compares the distribution of our *verified set* against the full *migration set*. The comparison confirms that our verified subset broadly mirrors the distribution of the wider dataset across origin languages and application domains. This alignment is significant for our methodology as it demonstrates that our interview sample is representative of the full dataset, thereby reducing the risk of sampling bias.

4.6 Improved bounds using the migration and verified sets

In order to assist with future research and evaluate our initial heuristic bounds, we investigated whether these thresholds could be optimised mathematically using the migration set and the verified set as the ground truth labels. Specifically, we optimised for the F_1 score, which represents the harmonic mean of precision and recall. However, this optimisation yielded mixed results across both datasets.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

Based on the broader migration set, optimising the F_1 score yields a maximum score of 40.3%, which is achieved with movement and overlap bounds of 137 and 21,945, respectively. With these thresholds, the model achieves a precision of 30.4% and a recall of 59.7%.

Conversely, optimising based on the verified set achieves a maximum F_1 score of 32.0%, with a movement bound of 298 and an overlap bound of 21,945, which is the same as before. At these thresholds, the precision is 34.0% and the recall is 31.0%. While the maximum F_1 score is lower when optimised to the verified set, the resulting bounds are more robust from a methodological perspective, as they are corroborated by the developers themselves rather than just our manual verification.

These outcomes demonstrate that, although our original conservative bounds were imprecise (21.1%), empirical optimisation only provides marginal improvements. Notably, in both scenarios, the optimal F_1 score maximises the overlap bound. This effectively renders the parameter's filtering capability ineffective, suggesting that measured identifier overlap is not a reliable indicator of software migration.

Using the same optimisation process for cross-repository migrations results in significantly lower performance. Based on the migration set, the maximum F_1 score is 13.2% (precision: 8.0%; recall: 37.6%), which is achieved with a Jaccard similarity bound of 13.4%. Similarly, optimising the verified set results in a maximum F_1 score of 12.0% (precision: 7.5%; recall: 30.1%) at a bound of 14.4%. These results demonstrate that the F_1 score for cross-repository detection is consistently lower and less reliable than for in-repository detection.

Chapter 5

Interview results

This chapter presents the qualitative findings from interviews with the authors of 38 verified migrated projects, addressing the second research question. The analysis covers the developers' prior experience with Rust, their motivations for migrating, the strategies they employed, their perceptions of the impact on software quality, and their reflections on the challenges they faced.

5.1 Context

Before analysing the motivations and outcomes of the migration, we established the context of each project. This involved determining each participant's specific role, their prior experience with Rust, and the overall scope of the transition.

During the interviews, we assessed each participant's responsibilities to understand their perspective within the project. As shown in Table 2, our thematic coding revealed four distinct roles. Participants were classified as the *main author* (53%) if they were the repository owner and lead contributor, but still managed a broader team of contributors. Participants who executed the entire migration independently, save for minor bug fixes or ideas from external users, were categorised as the *sole author* (32%). Participants who primarily oversaw the migration, coordinated the team, and conducted code reviews, but wrote less code themselves, were assigned to the *project manager* (8%) role. Finally, participants were coded as co-authors (5%) if they shared coding and management responsibilities equally with other core contributors. In one instance, the interviewee's specific role was not mentioned.

Table 2: The responsibilities of the participant in the interview project.

Code	#	%
Participant was the main author.	20	53
Participant was the sole author.	12	32
Participant was the project manager.	3	8
Participant was a co-author.	2	5
Not mentioned.	1	3

Furthermore, we asked participants to evaluate their prior experience with Rust, as well as that of their wider team (Table 3). Please note that, for some categories, multiple codes could be assigned to a single project (designated as 'multiple possible'). If a topic did not arise during the interview, we coded it as 'not mentioned' or 'none mentioned'.

Overall, 12 participants (32%) reported having prior experience with Rust from personal, open-source or commercial projects. However, this means that a significant proportion of the developers were learning the language while carrying out the migration. For instance, one participant noted that their entire team was new to Rust and reflected on the collective learning curve: "*I think for me, it was harder, for example, than for some of these people, but we really had to learn as a team. And so it was a bit of a leap of faith.*"

Conversely, the presence of experienced team members often mitigated this friction. In terms of team dynamics, five participants reported working in teams where some members had prior Rust experience. Of these five participants, three said that they were new to the language themselves, while the other two did not specify their level of proficiency. Similarly, two participants mentioned having a dedicated Rust expert on their team: one of these participants was new to Rust, and the other did not specify their level of experience. To highlight explicitly that no participant reported being in a team where everyone was already experienced with Rust, we included the 'No team members were new to Rust' code in the table.

One participant emphasised the importance of this in-house expertise in overcoming technical obstacles: *"I essentially got stuck and luckily one of my guys on the team is really into Rust and very deeply knows it. So, he implemented those hard parts that would have been very inefficient otherwise, or borderline impossible."*

Table 3: The mentioned experience with Rust for the participant or their team.

Group	Code	#	%
Participant *(multiple possible)*	Was new to Rust.	16	42
	Has experience on personal projects.	8	21
	Has experience on open source projects.	3	8
	Has experience on commercial projects.	2	5
	None mentioned.	12	32
Team	All team members were new to Rust.	2	5
	Some team members were new to Rust, some not.	5	13
	No team members were new to Rust.	0	0
	Not mentioned.	31	82
Other	Some team member was described as Rust expert.	2	5

Finally, to provide a full context, we categorised the scope of the migrations (Table 4) and asked whether the software had been expanded with new functionality during or since the migration. A migration was coded as involving the porting of all original functionality if the participant explicitly aimed to match the exact behaviour of the legacy version. Migrating most original functionality often involved intentionally removing features. As one developer noted: "*I did remove a few things*", and another said: "*the plan is to migrate all the basically business code but keep vendor provided drivers and the lower level code.*"

Examining the intersection between the migration scope and the addition of features reveals a few overlaps. Of the ten projects that coded for adding *new functionality*, six indicated that they had migrated *all* original functionality, two had migrated *most* functionality, and the remaining two fell into the other scope categories. One participant succinctly summarised the balance of porting and expanding: "*and I was innovating on 20% to 25% of it at the time.*"

Furthermore, three projects were coded for adding *new optimisations*, meaning they indicated the addition of performance improvements that did not exist in the original architectures. Notably, all three of these projects also reported migrating *all* original functionality. Lastly, in cases where developers decided that new code would always be written in Rust, they all reported that they had previously only migrated a specific component. This suggests that these projects consider their migration to be successful enough to fully adopt Rust without rewriting their entire previous codebase.

Table 4: The mentioned scope of the migration.

Group	Code	#	%
Migrated	All original functionality.	18	47
	Most original functionality.	6	16
	A specific original component.	11	29
	Only the core ideas of the original version.	2	5
	Not mentioned.	1	3
Included additions *(multiple possible)*	New functionality was added.	10	26
	New optimisations were added.	3	8
	New code will always be written in Rust.	2	5
	None mentioned.	24	63

5.2 Motivating reasons

This section explores the reasons behind projects migrating to Rust. As the events leading up to a migration, the reasons for selecting Rust and the targeted quality improvements are all closely linked, we have combined them into a single thematic coding structure. As detailed in Table 5, these motivations were categorised by their overarching software quality attributes. To ensure consistency across all projects, the responses were mapped to the Goal-Question-Metric (GQM) format. This enabled us to standardise the goals while preserving the original intent of the participants.

Examining the distribution of goals, the data shows that migrations are rarely driven by a single goal. The majority of projects cited two (21%), three (34%) or four (26%) concurrent goals. A smaller subset of projects reported more complex migrations involving five or six goals (10% combined), while just 8% were driven by a single objective.

5.2.1 Most common goals

The motivations for migrating to Rust largely centred on four software quality attributes: maintainability (cited by 68% of projects), performance efficiency (58%), security (34%),

Table 5: Motivating reasons for migration are mentioned, separated by applicable quality attributes. Each project may have multiple goals. Languages shows the count per origin language, coloured as: C and C++, Go, Python, JavaScript and TypeScript, and other languages.

Quality Attribute	Goal	#	%	Languages
Functional suitability	Change away from specific GUI framework.	1	3	1
	Improve specification correctness.	1	3	1
Performance efficiency	Improve time behaviour.	13	34	2 2 4 3 2
	Improve time behaviour consistency.	4	11	1 2 1
	Maintain time behaviour.	4	11	3 1
	Reduce distribution size.	3	8	1 1 1
	Reduce memory utilisation.	3	8	2 1
	Improve performance over other Rust migrations.	2	5	2
Compatibility	Integration with specific project.	2	5	1 1
	Maintain API compatibility.	2	5	2
	Access to specific dependency.	1	3	1
	Allow embeddability into specific hardware.	1	3	1
Interaction capability	Improve ease of use.	1	3	1
Reliability	Reduce code defects.	12	32	6 3 2 1
Security	Reduce memory safety issues.	12	32	9 2 1
	Prevent abuse by bad actors.	1	3	1
Maintainability	Improve maintainability.	10	26	2 3 3 2
	Improve contributor engagement.	6	16	3 1 2
	Reduce code complexity.	5	13	2 2 1
	Improve analysability.	4	11	3 1
	Improve developer experience.	4	11	3 1
	Improve modifiability.	3	8	2 1
	Improve testability.	2	5	2
	Reduce code duplication.	2	5	1 1
	Reduce time to fix.	2	5	1 1
	Improve error handling.	1	3	1
	Reduce defects after changes.	1	3	1
Flexibility	Improve cross-platform compatibility.	4	11	3 1
	Reduce proprietary dependencies.	1	3	1
	Reduce restrictive licenses included.	1	3	1
Safety	*none*			
Other	Learn Rust.	7	18	4 2 1
	Learn about original project.	1	3	1

,

and reliability (32%). Additionally, a notable proportion of participants cited 'learning Rust' as their goal, with one project indicating this as their sole motivation for migration.

Maintainability encompassed a wide range of goals, frequently obtained from struggles with the original language's ecosystem, paradigm, or idioms. For instance, all four participants who explicitly mentioned improving the "developer experience" cited difficulties with their original build tools. As one developer noted: "*it [Rust] has a built-in build system, for example, and we don't need to spend hours dealing with the C build system.*" Another significant factor influencing maintainability was the scalability of the

previous language. One participant stated: "*I made a decision saying that Golang won't scale … maintainability is a big problem and the type system is biting.*"

In terms of performance efficiency, the main focus was on improving "time behavior", particularly execution speed and latency reduction. For many projects, the goal was not only to increase execution speed, but also to improve performance consistency. Developers migrating from garbage-collected (GC) languages such as Java and Go often cited unpredictable latency spikes as a critical issue. Rust's deterministic memory management model, which does not utilise runtime garbage collection, provides the predictable performance required for high-load systems. One participant summarised this succinctly: "*In our previous experience with Go-based storage systems, garbage collection pauses introduced unpredictable latency spikes under high loads, which is unacceptable for enterprise-grade storage.*"

Finally, the goals of reducing general code defects (reliability) and eliminating memory safety issues (security) were closely intertwined in the participants' responses, reflecting the fact that Rust's compiler addresses both issues simultaneously. Many participants expressed a preference for Rust's "correctness by construction" philosophy. They specifically noted that the strict type system helps prevent general logical errors and that the borrow checker systematically eliminates memory vulnerabilities. Highlighting the value of catching these bugs during compilation rather than in production, one developer explained: "*C makes it easy to shoot yourself in the foot. Rust removes entire classes of memory bugs at compile time without giving up low-level control.*"

5.2.2 Goals per language

When we categorised developers according to the original programming language of their projects, several patterns emerged regarding their reasons for adopting Rust.

As expected, reducing memory safety issues is the main reason cited by projects migrating from C and C++. However, it is interesting to note that this goal was also cited for some migrations from memory-managed languages such as Go and Python. Although these higher-level languages use garbage collection to prevent classic vulnerabilities such as buffer overflows, they are still susceptible to runtime errors, null reference exceptions and concurrency data races. Our findings suggest that developers migrating from these languages broadly categorise these issues under the umbrella of "memory safety" even though they are not necessarily memory vulnerabilities. This is partly an awareness issue, but developers also actively seek Rust's strict compile-time guarantees to eliminate other classes of runtime errors.

In terms of performance efficiency, these goals were more prevalent among projects migrating from non-C languages. The notable exception is the goal of maintaining time behaviour, which was exclusively a priority for C/C++ migrations. This trend reflected the trade-offs associated with higher-level, interpreted or garbage-collected languages, which generally have higher latency and greater resource overhead than systems-level languages. One participant migrating from Python explicitly articulated this limitation: "*I was dissatisfied with the application startup latency and runtime performance in certain*

scenarios because Python has a relatively long interpreter startup time and it's generally slow."

Finally, the distribution of maintainability goals demonstrated broad but nuanced appeal. As expected, the majority of projects migrating from C and C++ (76%) cited maintainability as a goal, reflecting the historical burden of manual memory management and associated tools. However, a significant proportion (57%) of projects migrating from all other languages also reported at least one maintainability-related goal. This suggests that the maintainability benefits of Rust, such as its expressive type system, standardised build tooling (`cargo`), and strict compiler, are not viewed purely as a fix for ageing C/C++ codebases. Rather, developers actively expect improvements in maintainability even when migrating away from modern, high-level languages.

5.3 Strategies applied

When participants described the approaches they used during their migrations, two distinct dimensions emerged from the interviews. These patterns provide a framework for categorising the ways in which developers choose to transition features and reconstruct legacy logic.

The first dimension determines the transition state of the software, i.e. whether features are transferred while maintaining a continuously running hybrid version or whether the component must be fully rebuilt before integration. In the former approach, known as an **intermediate** version, both Rust and the legacy language coexist within the software during the transition and typically rely on bindings to interoperate. In the latter approach, the Rust code is developed as a **standalone** component and is only introduced into the system once migration is complete, replacing the original version entirely. Although this standalone approach is often used to replace an entire backend at once, participants have also applied it to smaller components, such as individual microservices.

The second dimension determines the translation method, distinguishing between a top-down approach and a bottom-up approach. In a **top-down** approach, the specific set of features from the previous version is redeveloped in Rust. Participants opted for this method in order to utilise Rust idioms and design patterns from the start of the process. In contrast, the **bottom-up** approach closely mirrors the structure of the legacy codebase, translating the software function-by-function or line-by-line. An example of this is the use of tools such as C2Rust, which strictly map C functions to Rust functions. Participants working bottom-up frequently noted that their initial Rust code was unidiomatic, requiring subsequent refactoring to move away from the legacy architecture.

Crossing over these transition states and translation methods yields four options. The data indicates a preference for complete isolation, with the standalone approach being more common (71% of projects), compared to managing an intermediate version (23%). Projects are more evenly distributed in terms of translation preference, with 57% utilising a top-down approach and 39% opting for a bottom-up approach. One project was coded as using

both translation methods: the developer first migrated a small core component line-by-line as a proof of concept, then shifted to a top-down strategy for the rest of the software.

Beyond these two dimensions, Table 6 shows a number of other strategies that were mentioned by the participants. With regard to the adoption of automated translation tools, existing literature often focuses on dedicated source-to-source transpilers. However, only one of the projects interviewed applied such a tool. In contrast, two separate projects reported using Large Language Models (LLMs) to assist with the translation process, rather than relying on specialised translation methods.

When discussing the testing strategies employed during migration, twelve projects reported reusing test suites from their legacy versions, whereas only three relied on newly created tests. This suggests that developers treat the new Rust version as a drop-in replacement and use the existing test suite to verify backwards compatibility.

Table 6: The mentioned strategies that were applied.

Group	Code	#	%
Strategy *(multiple possible)*	Standalone & top down.	17	45
	Standalone & bottom up.	11	29
	Intermediate & top down.	5	13
	Intermediate & bottom up.	4	11
	None mentioned.	2	5
Testing strategy *(multiple possible)*	Tests from original.	12	32
	Newly created tests.	3	8
	Little to no tests.	1	3
	None mentioned.	22	58
Other strategies *(multiple possible)*	Started with a pilot migration.	3	8
	Started with fork of existing migration.	2	5
	Used LLM.	2	5
	Used automatic translation tool.	1	3
	None mentioned.	31	82

5.4 Perceived results

Before assessing the perceived impact on software quality, we first established the progress of the migration process for each project (Table 7). At the time of the interviews, 95% of migrations had been completed to some extent, providing an accurate basis for evaluating their outcomes.

Table 7: Self-reported completion status of the migration process.

Code	#	%
The migration has been completed.	27	71
The migration is mostly completed.	6	16
The migration is partially completed.	3	8
Not mentioned.	2	5

As the project goals were determined a posteriori from the interviews, the perceived results are shown per quality attribute in Table 8. The data shows that participants were overwhelmingly positive about the impact of the migration on their software. This trend was most pronounced for maintainability. While many projects anticipated improvements in this area, almost all of them reported experiencing tangible benefits post-migration, with none of them reporting a deterioration. This suggests that maintainability benefits are highly predictable prior to migration and reliably realised in practice.

The responses regarding performance efficiency were more nuanced. Several projects reported that performance either remained unchanged or deteriorated, despite improvements being anticipated. Notably, of the projects that specifically aimed to maintain their existing timings, two reported a neutral outcome and two reported a negative outcome. A major theme that emerged from the responses was the loss of predictability in resource consumption. As one participant remarked: "*Before, if you added a line of code, you knew exactly how much the executable would grow. Now, because of all the dependencies, it's very unpredictable.*"

It should be noted that, since outcomes can vary within a quality attribute, one project reported both an improvement and a neutral outcome for different performance efficiency sub-goals. This is reflected as two separate data points in Table 8.

Table 8: We show the number of projects with perceived improvements (blue), deteriorations (orange), a neutral experience (pink) and projects that did not mention that quality attribute (gray). Note that multiple codes are possible per project, per row.

Quality attribute	With goal				All projects			
Functional suitability	1			1				36
Performance efficiency	13	4	2	4	15	6	3	15
Compatibility	3			3	4			34
Interaction capability				1				37
Reliability	7		1	4	9			26
Security	9			4	9			29
Maintainability	19			7	19			17
Flexibility	3			3	3			35
Safety	no data				38			

5.5 Difficulties and reflection

Lastly, participants were asked to reflect on the migration process, detailing specifically the difficulties they had encountered and how they would approach it differently if they were to do it again.

As outlined in Table 9, adapting existing software architectures and features to Rust-specific idioms, particularly the type system and borrow checker, proved challenging for many projects. While this difficulty was most prominent among non-C projects (57%), it was still reported by 29% of C migrations. As one contributor noted: "*Unlike other languages, if I make the architecture wrong, the borrow checker … will make the progress slower.*"

In line with previous studies, a lack of prior experience with Rust was often mentioned as a major obstacle. Notably, when participants listed multiple challenges, this knowledge gap was commonly identified as primary obstacle.

Participants also reported difficulties related to unmet project goals, particularly slow user adoption, prolonged migration timelines and maintaining baseline performance. With regard to slow adoption, one participant observed: "*Large organizations are interested in adoption; it just turns out they move very slowly.*"

Creating software distributions presented additional challenges. For example, participants targeting Debian and Ubuntu reported incompatibilities between the older Rust toolchains required by the operating system and the newer ones required by their dependencies: "*we often fight problems where [our project] suddenly doesn't compile anymore because an indirect dependency has decided to bump the minimum supported Rust version.*" Furthermore, participants noted that, although necessary language features sometimes existed, they were often still classified as unstable.

Several participants struggled with the absence of equivalent library crates for functionalities they had previously relied on in their original language. However, some noted that the ecosystem is maturing: "*When we started there was very little; the library crate ecosystem wasn't very mature and large parts were missing, so we had to build large parts.*"

Finally, individual participants identified various technical challenges, including debugging deadlocks or bugs within language bindings, keeping the Rust migration synchronised with the original codebase and recreating the original software logic.

Table 9: The mentioned difficulties with the migration process.

Code ***(Multiple possible)***	**#**	**%**
Adapting to Rust idioms.	17	45
Being new to Rust.	15	39
Underperforming set goals.	6	16
Issues with creating distributions.	5	13
Issues with Rust ecosystem.	4	11
Other	4	11
None mentioned	4	11

Table 10: Mentioned what they would do differently.

Code ***(Multiple possible)***	**#**	**%**
Apply a different design.	7	18
Use more automation.	7	18
Use different dependency or build system.	5	13
Apply a different starting strategy.	3	8
Take more time to learn Rust beforehand.	2	5
Nothing	6	16
None mentioned	10	26

When asked what they would do differently if they had to repeat the migration process (Table 10), participants highlighted several key areas for improvement. Notably, a significant proportion indicated that they would adopt a different architectural design approach. Specifically, they mentioned isolating components earlier, allowing for modifications to original function contracts, and dedicating more time to upfront structural design. One participant succinctly described the cost of inadequate upfront design: "*it might have been wiser to start with a simpler system and finish much more quickly than implementation. I'd say that probably set us back a month out of the year, just maintaining the extra complexity, which was high cost.*"

Several projects stated that they would leverage more automation, noting that migration tools have progressed substantially since their initial efforts. One participant explicitly mentioned a preference for using Large Language Models (LLMs) as they had had unsuccessful experiences with traditional automatic translation tools in the past.

Other alternative strategies included implementing mechanisms to identify migration issues earlier, prioritising testing from the outset and tracking performance metrics at an earlier stage in the process. In contrast, a subset of projects indicated that they would not change their approach at all.

Chapter 6

Measurement results

This chapter presents the quantitative analysis of software quality changes across the migrated projects, addressing the third research question. It maps the migration goals from the participants to specific, measurable quality metrics, and evaluates the pre, during and post-migration data to determine the objective impact of migrating to Rust on various aspects of software quality.

6.1 Derived metrics

As shown in Table 5, participants identified 32 distinct goals that motivated their migration to Rust. To evaluate whether these goals were achieved, we applied the Goal-Question-Metric (GQM) framework to derive metrics from them. Based on the methodology's established criteria, we excluded a subset of these goals from our analysis. The remaining goals were mapped to metrics and categorised by their relevant quality attributes. The full mapping is summarised in Table 11.

6.1.1 Performance efficiency metric

Most performance efficiency goals require dynamic runtime analysis for evaluation. As dynamic analysis falls outside the scope of this research, our focus is exclusively on the goal of *reducing distribution size*. Unlike runtime metrics, distribution size can be evaluated using existing artefacts. Using the GQM framework, the mapped question becomes: *what is the current distribution size of the software?* To answer this we use a metric derived from GitHub release data.

Mean Distribution Size. This metric is calculated at a specific point in time, t, by determining the mean size (in bytes) of the assets associated with a GitHub release at time t. To ensure that the metric reflects the size of executable distributions rather than supplementary files, such as signatures, two exclusion criteria are applied to the assets. Firstly, we exclude any assets smaller than 128 KB, as these are generally too small to be executables. Secondly, any files containing the term "source" in their filename are excluded to prevent source code archives from being included. This calculation is then repeated for each GitHub release.

Table 11: The mapping from mentioned goals in interviews, to questions for that goal and metrics per question using the GQM framework. Note that multiple metrics may be mapped to a single question, and a specific metric may appear multiple times.

Goal	Question	Metrics
Reduce distribution size	What is the distribution size?	Mean Distribution Size
Reduce code defects	How many code defects are reported?	Defect Issue Count
	Are defects visible in the code?	Reliability Debt Density
Improve maintainability	How well is the code documented?	Documentation Density
	What code maintainability is measured?	Cyclomatic Complexity, Technical Debt Density
Improve contributor engagement	How many contributors are engaged?	(Decayed) Contrib. Truck Factor
	Is contributor engagement spread equally?	(Decayed) Contrib. Gini Coefficient
Improve modifiability	How easily can the codebase be modified?	Cyclomatic Complexity / Unit, % Duplication
Improve analysability	How analysable is the code?	LOC, Cognitive Complexity, LOC / Unit, % Duplication
Improve testability	How much is the code currently tested?	Test Density, Assertions-Complexity Ratio
	How testable is the code?	Cyclomatic Complexity / Unit, LOC
Reduce code complexity	How complex is the code?	Cyclomatic Complexity, Cognitive Complexity
Reduce code duplication	How much code duplication is there?	% Duplication
Reduce time to fix a defect	How long are issues open?	Defect Issue Resolution Time
	How long are defects present?	Mean Time Inducing to Fix Commit
Reduce memory safety issues	How many memory safety issues are reported?	*Not measured*
	Are memory safety issues visible in code?	Unsafe Density, Security Debt Density
Improve cross-platform compatibility	How many platforms are supported?	Distribution Platform Count

6.1.2 Reliability metrics

To evaluate the goal of *reducing code defects*, we ask two questions: *how many defects are reported externally*, and *how many defects are detected within the codebase*? To answer the first question, we analyse GitHub issues that report defects.

Defect Issue Count. For this metric, we extract issue data from GitHub and filter it for reports containing defect-related keywords in their title or body. We use the validated keyword set proposed by Ray et al. [25], and aggregate the identified defect issues on a monthly basis to create a time series.

To answer the second question, we evaluate the relative trends of the **Reliability Debt Density** metric, as measured by SonarQube. We adopt the same methodology as that used by Lenarduzzi et al. [35], who applied this approach to analyse a migration from a monolithic system to microservices. Analysing relative trends rather than absolute values provides a more robust comparison, given that SonarQube applies different analysis rules depending on the programming language. Furthermore, using the density-based version of this metric normalises the data according to project size, preventing our results from being skewed by size.

6.1.3 Improving maintainability

Participants frequently cited motivations directly related to the software quality attribute of maintainability. The most common objective within this category was the general idea of *improving maintainability*. As this goal is inherently broad, and as some interviewees did not specify a more specific rationale for their project, we formulated two high-level questions using the GQM framework. Firstly, *what is the measurable maintainability of the codebase*? Secondly, *to what extent is the code documented*?

To answer the first question, we used two well-established metrics. The first is **Cyclomatic Complexity**, which was introduced by McCabe in 1976 [55], to quantify the number of linearly independent paths through a program's source code. A systematic mapping study by Arvanitou et al. [56] identified multiple empirical studies supporting its use as an indicator of maintainability. We use the `RCA` tool to collect our cyclomatic complexity measurements. The second metric is **Technical Debt Density**, which is measured using SonarQube. We justify this on the basis that it has previously been shown by Lenarduzzi et al. [35] to be useful for analysing code quality for migrations.

In answer to the second question regarding documentation, we used the **Documentation Density** metric. This is calculated as the ratio of comment lines to total lines of code within the program. We collect these values from `RCA`, which also counts lines ending in a comment as comment lines.

6.1.4 Improving contributor engagement

Quantifying the goal of *improving contributor engagement* creates methodological challenges: what constitutes a contributor, and what constitutes engagement? To address these questions systematically, we evaluated two dimensions of engagement: the absolute number of active contributors and how authorship is distributed among them.

We also adopted a strict definition of 'contributor' by using the degree-of-authorship model proposed by Fritz et al. [57]. According to this model, a developer's authorship of a file depends on whether they created it and the proportion of subsequent modifications to which they contributed. This formula was then empirically validated by weighting these factors according to the developer's knowledge of their own files, as tested. We applied this established authorship model, defining a contributor as someone who is the author of at least one file, to obtain the metrics for both dimensions of engagement.

Contributor Truck Factor. This metric, which was introduced by Avelino et al. [58], tracks the minimum number of contributors needed to account for at least 50% of a project's relevant files. Files that are not relevant are detected by GitHub's Linguist tool and excluded from the analysis. We calculated these values using our custom `truck-factor-rs` tool, which implements the original Avelino algorithm and includes some performance optimisations.

Contributor Gini Coefficient. We adapted the standard statistical measure of wealth inequality, the Gini coefficient, to evaluate the distribution of author files per contributor [59]. Here, a coefficient of zero indicates perfect equality in terms of authorship, whereas a value of one means there is maximum centralisation (i.e. a single contributor). An important

nuance arises: an influx of new contributors making minor changes may increase the Gini coefficient if the core developers retain authorship of most of the files. This metric is also calculated using our `truck-facto-rs` tool.

Finally, we added two metrics to account for knowledge obsolescence, augmenting both by incorporating the concept of authorship decay [60]. Older codebase changes were given less weight than recent ones to more accurately show how many contributors have active knowledge of the software. These adjustments yield two additional metrics: **Decayed Contributor Truck Factor** and **Decayed Contributor Gini Coefficient**.

6.1.5 Improving modifiability, analysability and testability

The sub-attributes of maintainability (modifiability, analysability and testability) can be mapped to three related questions: *How easily can the code be modified, how analysable is the code, and how testable is the code?* To map these questions to metrics, we use the 2025 iteration of the maintainability evaluation model developed by the Software Improvement Group (SIG) and TÜV Nord [61]. This updated model is based on the work of Heitlager et al. [62] and provides an established expert consensus on how source code properties translate into maintainability sub-attributes.

In terms of modifiability, the SIG/TÜV model identifies code duplication, unit complexity and module coupling as determining properties. To evaluate duplication, we used the **% of Duplication** metric provided by SonarQube. This is the percentage of lines of code that appear multiple times throughout the codebase. Unit complexity is measured by taking the mean cyclomatic complexity of the smallest executable units of code (e.g. functions or methods), which is our **Cyclomatic Complexity / Unit** metric, collected with `RCA`. We do not analyse module coupling, because the modularisation is not comparable between programming language paradigms.

Analysability is mapped onto the properties of code volume, duplication, and unit size. We quantified volume using two metrics collected with `RCA`: **Lines of Code (LOC)** and **Cognitive Complexity**. The inclusion of *LOC* is also supported by the systematic mapping study conducted by Arvanitou et al. [56]. *Cognitive Complexity* was included because it has been successfully applied to measure codebase volume in a similar migration study [63]. Introduced by Campbell [64], it serves as an extension to cyclomatic complexity by incorporating rules that better approximate human readability and comprehensibility. Duplication is measured using the aforementioned SonarQube metric. Finally, unit size is measured by calculating the mean lines of code per unit (**LOC / Unit**), again using `RCA`.

According to the model, testability is associated with volume, unit complexity, and component independence. As with the previous sub-attributes, we use *LOC* and *Cylomatic Complexity / Unit* to measure volume and unit complexity, respectively. Component independence is excluded for the same reason as module coupling.

To further evaluate the goal of improving testability, we introduce the following: *to what extent is the codebase currently tested?* As dynamic runtime analysis is outside the scope of this research, it is not possible to directly compute execution-based test coverage. However,

to bridge this gap, we use two static metrics proposed by Athanasiou et al. [65]: Test Density and the Assertions-Complexity Ratio.

Test Density is the ratio of the number of test-specific lines of code to the total number of lines of code. In `RCA`, we implemented this by counting the lines containing test-indicatory identifiers, depending on the language being measured. The *Assertions-Complexity Ratio* divides the total number of test assertions present in the source code by the cyclomatic complexity of the non-test codebase. The idea is that achieving complete coverage requires at least one assertion for every linearly independent execution path (as quantified by cyclomatic complexity). However, it is important to acknowledge the limitations of this static approximation, as it gives an inherent undercount: single assertions may be executed multiple times, for example in a loop. Furthermore, identifying assertions statically again depends on language-specific heuristics, meaning our parsed set of assertions is an estimation rather than an exhaustive count.

6.1.6 Other maintainability metrics

Participants identified another maintainability goal: *reducing the codebase complexity*. This goal is mapped to the question: *how complex is the current codebase?* We evaluate this using two metrics collected with `RCA`: **Cyclomatic Complexity** and **Cognitive Complexity**.

The goal of *reducing code duplication* translates directly to the question of how much duplication exists within the project. We measure this by using the previously defined **% Duplication** metric.

Lastly, we analyse the goal of *reducing the time required to fix defects*. This is analysed from two perspectives: the length of time that defect issues are open in the issue tracker and the length of time that defective code is present in the repository. Each perspective is mapped to its own metric.

Defect Issue Resolution Time. This metric defines the resolution timespan, d, as the period between an issue being created and the first closure event occurring. Using the methodology of Kikas et al. [66], issues that remain open are excluded from the analysis. Furthermore, we exclude issues that were opened and closed in different phases of the migration process for two reasons. Firstly, we found that cross-phase issues are often closed not because they have been resolved, but because maintainers deem them obsolete once migration has occurred. Secondly, using issues from only one phase reduces temporal bias. Since the post-migration phase occurs later, issues opened in the pre-migration phase have a longer potential lifespan. However, we must acknowledge that this does not fully eliminate the bias, as phase durations are unequal across projects and longer phases may still correlate with longer average issue resolution times. As with the previously discussed defect issue metric, we strictly include issues classified as defects.

Mean Time from Inducing to Fix Commit. To calculate the duration between the introduction of a defect and its subsequent fix, we must first identify the relevant "fix" and "inducing" commits. Fix commits are identified using an established keyword heuristic [25] and for each identified commit, we trace the modified or deleted lines of code back to the commits that originally introduced them. These are defined as the *inducing* commits.

This identification process replicates the algorithm implemented by Spadini et al. [67] in the PyDriller framework. Our metric provides a value at the time of the fix commit, t, representing the average time between the fix commit and all its inducing commits. We have implemented this within our custom `szz-rs` tool.

6.1.7 Security metrics

The only goal we include regarding the quality attribute of security is *reducing memory safety issues*. We have mapped this goal to two questions: *to what extent are memory safety vulnerabilities reported externally*, and *to what extent are potential memory safety issues present within the codebase?*

For the first question, we initially aimed to quantify the number of reported Common Vulnerabilities and Exposures (CVEs) categorised specifically under CWE-1399 (Comprehensive Categorisation of Memory Safety) [68]. However, our initial analysis showed that almost no vulnerabilities were reported for the projects in our verified set. Therefore, this metric lacks the necessary data points to enable meaningful comparison and has been excluded from our analysis.

To answer the second question regarding the codebase, we used two metrics. Firstly, we used the **Security Debt Density** metric from SonarQube, replicating the approach used by Lenarduzzi et al. [35]. Secondly, we used a custom metric developed specifically to quantify the volume of source code containing potentially unsafe memory operations.

Unsafe Density. This metric calculates the proportion of lines of code containing potentially memory-unsafe operations, normalised against the size of the entire codebase. We define "unsafe" in a language-dependent manner. In Rust, unsafe code is easily identified as code within `unsafe` blocks or `unsafe` functions. In C and C++, this includes operations that are inherently unsafe, such as pointer arithmetic and invocations of standard library functions that have historically been vulnerable (e.g. `strcpy`, `gets`). For other languages, we counted invocations of specific standard API functions that are known to bypass memory safety guarantees.

Firstly, this metric serves as an indicator of the review burden, showing the proportion of the codebase that requires more rigorous auditing to ensure safe memory access. Secondly, it still allows for some comparisons across languages. In Rust, the explicit `unsafe` keyword ensures that our measurement represents the strictest possible upper limit of potentially memory-unsafe behaviour. Conversely, in languages like C, our heuristic-based count provides a lower bound. Therefore, if migration from C to Rust results in a decrease in the measurement, we can be confident that the overall volume of potentially memory-unsafe behaviour has decreased.

6.1.8 Flexibility metric

Lastly, with regard to the quality attribute of flexibility, participants identified the goal of *improving cross-platform compatibility*. Under the GQM framework, we addressed this goal by asking: *how many distinct platforms does the software currently support?* We answered this question by analysing the deployment targets indicated in the release artifacts on GitHub.

Distribution Platform Count. We quantify the number of supported platforms for each GitHub release by parsing the filenames of the included assets. To achieve this, we use a keyword-matching heuristic based on standard file extensions and the naming conventions of major operating systems: Windows, Darwin (macOS), Linux, Android, iOS and BSD. Furthermore, WebAssembly (WASM) and generalised cross-platform distributions are treated as separate platforms. The metric value for a given release is calculated by determining the total size of the unique set of targeted platforms identified within its assets.

6.2 Collected measurements

We successfully collected data for all 22 defined metrics, depending on the availability of data and the compatibility of the tool for each project. Before presenting the overall results and the results grouped by factor, we will first explain any constraints or missing data that occurred during the measurement process.

6.2.1 Missing measurements

As shown in Table 12, it was not possible to collect measurements for every project for every metric. Two projects did not have a pre-migration phase: they followed a pattern of implementing the functionality in the original language first and then migrating it to Rust shortly afterwards, thus maintaining parallel implementations for both languages. Furthermore, for several projects, the analysis is limited to comparisons between the pre-migration phase and either the during- or post-migration phase.

This is because these projects only have two observable phases: either the repository lacks a clear during-migration period, or the migration was still ongoing at the time of data collection. Specifically, five projects only have an observed during-migration phase, while 16 projects only have an observed post-migration phase. Of the latter, 11 were cross-repository migrations, which our methodology does not define as a ‘during-migration’ phase.

Regarding specific metrics, data could only be collected for 45% and 47% of projects for *Mean Distribution Size* and *Distribution Platform Count*, respectively. This is due to the absence of releases on GitHub for many repositories. Similarly, issue-related metrics were collected for 82% of projects; the remaining 18% did not use GitHub’s issue-tracking system.

Lastly, four projects are missing metrics from SonarQube due to two technical limitations. Firstly, some codebases exceeded the 1M LOC threshold, which was the maximum permitted by our licence. Secondly, some projects that originally used Java relied on obsolete dependencies that are no longer available, breaking SonarQube’s automated collection pipeline.

Table 12: The number of projects measured per metric. The colours indicate the number of projects for which data was collected for both phases (green), only the during phase (blue), only the after phase (yellow) or none of these phases (gray).

	Metric	Measured counts			
GitHub	Defect Issue Count	9	2	19	6
	Defect Issue Resolution Time	9	2	19	6
	Distribution Platform Count	5	3	10	18
	Mean Distribution Size	5	3	9	19
RCA	Assertions-Complexity Ratio	15	5	16	
	Cognitive Complexity	15	5	16	
	Cyclomatic Complexity	15	5	16	
	Cyclomatic Complexity / Unit	15	5	16	
	Documentation Density	15	5	16	
	LOC	15	5	16	
	LOC / Unit	15	5	16	
	Test Density	15	5	16	
	Unsafe Density	15	5	16	
SonarQube	% Duplication	14	4	14	4
	Reliability Debt Density	14	4	14	4
	Security Debt Density	14	4	14	4
	Technical Debt Density	14	4	14	4
Other	Contrib. Decayed Gini Coefficient	15	5	16	
	Contrib. Decayed Truck Factor	15	5	16	
	Contrib. Gini Coefficient	15	5	16	
	Contrib. Truck Factor	15	5	16	
	Mean Time Inducing to Fix Commit	14	5	17	

6.2.2 General measurement results

Table 13 shows the outcomes for each metric categorised into one of four groups: a statistically significant increase or decrease ($p < 0.05$), or a non-significant change. Please note that projects for which no data was collected are not included here, but can be found in Table 12. This categorisation compares pre-migration baseline data with both during-migration and post-migration data. For each category, the number of projects that explicitly stated a goal mapped to the respective metric is shown in a separate column. The precise comparison method depends on the metric's comparability: (A) we either use the full set of results, (B) the 10 most recent results or (C) the slope of the linear trend across all results in a given phase.

The aggregated results in Table 13 indicate a nuanced set of changes to the software quality throughout the migration process. Averaging the results across all the metrics during migration shows that 17% of them significantly increased, 14% decreased, 16% had no significant changes, and 53% lacked sufficient data. For post-migration, these proportions were 29% increasing, 22% decreasing, 28% showing no significant change, and 20% lacking data.

As the desired direction of improvement depends on the metric's goal (for example, an increase in *Documentation Density* is positive, whereas a decrease in *Unsafe Density* is also positive), we must adjust for this *directionality*. When we accounted for this during

Table 13: Metric shifts from pre-migration baseline to during- and post-migration phases for Rust code. Desired trends are marked ↗ (maximise) or ↘ (minimise). Colors show the number of projects with significant increases (blue), decreases (orange), or no change (pink). The letters A, B & C indicate the different methods used to compare the phases.

	Metric	vs. Rust during migration		vs. Rust after migration	
		With goal	All projects	With goal	All projects
GitHub	Defect Issue Count (B) ↘	1 1 3	9	1 1 6	5 22
	Defect Issue Resolution Time (A) ↘	1	2 2 7	1	3 11 14
	Distribution Platform Count (B) ↗	no data	2 1 5	2 1	5 2 8
	Mean Distribution Size (B) ↘	1	5 3	1 2	7 5 2
RCA	Assertions-Complexity Ratio (B) ↗	1 1	11 6 3	no data	23 5
	Cognitive Complexity (B) ↘	1 2	8 11	4 2 1	16 13
	Cyclomatic Complexity (B) ↘	3 4	11 8	7 4	20 9
	Cyclomatic Complexity / Unit (B) ↘	1 3	8 12	2 1	14 16
	Documentation Density (B) ↗	4 2	8 11	4 5	10 19
	LOC (B) ↘	2 1	13 7	2	20 8
	LOC / Unit (B) ↘	1 1	7 13	1 1	14 16
	Test Density (B) ↗	2	13 6	no data	22 4 5
	Unsafe Density (B) ↘	1 1 3	7 3 10	5 4	14 9 8
SonarQube	% Duplication (B) ↘	1 1 2	7 6 5	5 1 1	17 7 4
	Reliability Debt Density (C) ↘	9	18	8	28
	Security Debt Density (C) ↘	4	17	1 6	26
	Technical Debt Density (C) ↘	2 4	6 2 10	5 2 2	10 12 6
Other	Contrib. Decayed Gini Coefficient (B) ↘	3	11 6 3	2 2	17 11
	Contrib. Decayed Truck Factor (B) ↗	2 1	5 13	1 1 2	9 19
	Contrib. Gini Coefficient (B) ↘	3	11 6 3	2 2	15 12 4
	Contrib. Truck Factor (B) ↗	2 1	3 15	1 3	4 24
	Mean Time Inducing to Fix Commit (A) ↘	1 1	3 5 11	1 1	6 7 18

migration, an average of 14% of projects improved and 16% deteriorated. Post-migration, 25% of projects showed improvements, while 27% showed deterioration.

A more granular analysis of the data reveals distinct patterns. Firstly, of the code analysis metrics measured with `RCA`, only a small minority of projects showed non-significant changes. On average, 12% of projects showed non-significant changes when the pre- and post-migration phases of these ten metrics were compared. Instead, the vast majority of projects demonstrated either a significant increase (49%) or a significant decrease (30%).

Conversely, the SonarQube metrics for *Security Debt Density* and *Reliability Debt Density* showed mostly non-significant changes. Further inspection of the collected measurements revealed that both metrics consistently approach zero for Rust codebases. This suggests that SonarQube's analysis rules rarely identify security or reliability debt in Rust, either because these are not present, or because the ruleset is smaller.

Examining the metrics that showed the most substantial improvements post-migration, *Test Density* improved for 58% of projects and worsened for only 13%. This positive trend may be attributable to Rust's integrated testing framework and was corroborated by feedback from interviewees. There was also a smaller positive shift in *Defect Issue Resolution Time*, which improved for 29% of projects and worsened for 8% post-migration.

In contrast, analysing the metrics with the highest rates of post-migration deterioration shows that *% Duplication* worsened for 53% of projects and improved for only 13%. Notably, compared to the during-migration phase, the gap between improvements and deteriorations has changed considerably. This suggests that, while code duplication remains relatively stable for many projects during the migration process itself, the majority of projects struggle with increased duplication after completing the migration to Rust. Other post-migration deteriorations are visible in *LOC* (53% worse, 21% improved) and *Cyclomatic Complexity* (53% worse, 21% improved). However, it is important to note that *Cyclomatic Complexity/Unit* shows an almost equal distribution of projects that have worsened versus improved. This suggests that, while overall codebases are growing in size and total complexity, many projects successfully maintain or even improve complexity in terms of function size.

Finally, Table 14 shows the changes in metrics for the non-Rust code within these projects. These results reveal no obvious correlation between shifts in the quality of the non-Rust code and the previously discussed shifts in the Rust code. Please note that this table excludes metrics that cannot differentiate between Rust and non-Rust code. These are all shown in Table 13.

Table 14: Metric shifts from pre-migration baseline to during- and post-migration phases for non-Rust code. Metrics lacking language differentiation are excluded. Desired trends are marked ↗ (maximise) or ↘ (minimise). Colors show the number of projects with significant increases (blue), decreases (orange), or no change (pink).

	Metric	vs. Rust during migration		vs. Rust after migration	
		With goal	All projects	With goal	All projects
RCA	Assertions-Complexity Ratio ↗	1 1	4 8 8	no data	6 14 11
	Cognitive Complexity ↘	1 2	7 12	2 4 1	8 20
	Cyclomatic Complexity ↘	4 2 1	7 12	5 5 2	9 18 4
	Cyclomatic Complexity / Unit ↘	2 1 1	7 10 3	2 1	7 20 4
	Documentation Density ↗	4 1 1	6 13	3 5 1	8 20
	LOC ↘	1 2	7 12	1 1	9 19
	LOC / Unit ↘	2	6 9 5	1 1	7 21
	Test Density ↗	1 1	3 8 9	no data	9 11 11
	Unsafe Density ↘	3 2	7 12	2 5 2	13 15
SonarQube	% Duplication ↘	1 1 2	3 7 8	2 2 3	5 13 10
	Reliability Debt Density ↘	4 2 3	7 4 7	3 1 4	9 4 15
	Security Debt Density ↘	2 2	3 14	1 1 5	3 3 22
	Technical Debt Density ↘	1 1 4	4 5 9	3 3 3	9 7 12
:	Mean Time Inducing to Fix Commit ↘	1 1	7 5	1	2 9

6.2.3 Results for the migration goals

By aggregating the metric outcomes according to their migration goal, we obtain the goal-level distributions shown in Table 15. As in the previous section, we adjust for metric directionality, ensuring that both statistical increases and decreases are correctly categorised as improvements or deteriorations based on the objective.

The aggregated data shows varying degrees of success for the different migration objectives. For instance, the objective of improving testability demonstrates a predominantly positive trend, with metrics indicating more improvements than deteriorations in all projects during and after the migration phases. In contrast, some frequently cited goals show negative trends. Metrics for improving maintainability worsened more often than they improved post-migration. Similarly, the goal of reducing memory safety issues showed a similar number of improvements and deteriorations.

Furthermore, comparing the subset of projects targeting specific goals with the full set shows no significant differences in outcomes. The distributions of improvements, deteriorations and neutral results are largely consistent between the two groups. This suggests that prioritising specific quality attributes does not necessarily lead to better outcomes than those achieved by the full project population.

Table 15: Metric outcomes aggregated by migration goal for the during- and post-migration phases. The $\#_m$ column indicates the number of metrics per goal. Colors show total occurrences of goal-aligned improvements (blue), deteriorations (orange), or no significant change (pink).

Goal	$\#_m$	vs. (Rust) during migration		vs. (Rust) after migration	
		With goal	All projects	With goal	All projects
Improve cross-platform compatibility	1	no data	2 1 5	2 1	5 2 8
Improve analysability	4	2 5 1	37 35 6	1 7	44 67 10
Improve contributor engagement	4	4 6 2	16 30 34	6 5 5	29 45 50
Improve maintainability	3	7 7 4	18 28 12	10 15 2	31 49 10
Improve modifiability	2	3 1	18 15 5	3 1 2	23 31 5
Improve testability	4	5 2 1	43 33 4	no data	69 43 12
Reduce code complexity	2	2	19 19 2	3 5 2	22 36 4
Reduce code duplication	1	no data	6 7 5	2	7 17 4
Reduce time to fix	2	2 1	7 5 18	1 1 1	18 9 32
Reduce distribution size	1	1	5 3	2 1	5 7 2
Reduce code defects	2	1 1 12	27	1 1 14	5 50
Reduce memory safety issues	2	1 1 7	3 8 27	4 6 6	10 15 34

6.2.4 Influence of factors on migration success

In order to better understand how various project characteristics and strategic choices influence the success of a migration, we analyse the metric outcomes by grouping projects according to these different factors.

Comparing the aggregated success rates across these groups of factors (see Table 16) reveals subtle but discernible patterns. In terms of the application domain, software libraries demonstrate slightly higher rates of improvement than applications or software systems. Furthermore, projects characterised by a larger overall codebase or a high proportion of Rust code also demonstrate a positive impact. In contrast, metrics for community engagement and popularity, such as the number of stars and forks, do not reveal any clear differences in migration outcomes.

Regarding the applied strategies, the aggregates indicate that intermediate migrations show more improvement during the migration phase. Note, however, that the top-down

intermediate strategy results in a significantly higher number of deteriorations post-migration. Additionally, the top-down approach generally achieves better metrics than the bottom-up approach.

When we analyse the results based on the original programming language, a clear trend emerges: projects migrating from system-level languages (C and C++) demonstrate greater improvement during and after the migration process than those using higher-level languages. This may be because, when shifting to Rust, these languages have more to gain from the commonly mentioned goals.

Table 16: Aggregated metric outcomes grouped by project characteristics and migration strategies. Columns represent the sum of all metric evaluations within the specified group. Colors indicate the total occurrences of goal-aligned improvements (blue), deteriorations (orange), or no significant change (pink).

Type	Group	**vs. during migration**			**vs. after migration**		
Stars	Low	50	57	67	124	129	112
	Medium	64	56	55	68	75	81
	High	6	20	14	16	14	12
Forks	Low	23	14	35	98	83	84
	Medium	72	90	79	83	116	91
	High	25	29	22	27	19	30
Project size	Low	41	35	57	81	90	98
	Medium	47	51	56	83	97	90
	High	32	47	23	44	31	17
% in Rust	Low	48	45	45	42	48	52
	Medium	28	50	46	47	77	72
	High	44	38	45	119	93	81
Domain	App	43	72	68	74	92	92
	System	60	48	44	57	58	59
	Tool	8	4	6	6	9	7
	Library	9	9	18	71	59	47
Strategy	Int. Top down	25	11	16	3	13	22
	Int. Bottom up	19	31	12	12		7
	Sta. Top down	54	70	75	118	109	98
	Sta. Bottom up	12	19	27	75	81	68
Language	C/C++	56	35	40	91	77	85
	Go	25	12	25	50	35	45
	TS/JS	16	23	35	36	50	40
	Python	14	43	25	16	29	17
	Other	9	20	11	15	27	18

A breakdown of migrations by their start and end years (Table 17) provides an insight into how they evolve over time. For this analysis, we compare pre-migration data with the most recent data collected for each project. This means that we use post-migration results where available and default to during-migration results otherwise. The results suggest a non-linear trend for the start year of migrations: projects that started migrating in both the earliest and most recent years show greater improvement than during the intervening period. However, this trend is less clear when projects are grouped by their migration completion year.

Table 17: Aggregated metric outcomes grouped by the year the migration was initiated and completed. To maximise data coverage, this table uses post-migration results where available, defaulting to during-migration results otherwise. Colors indicate total occurrences of improvements (blue), deteriorations (orange), or no significant change (pink).

Year	Migration started in			Migration finished in		
2016	10	17	13	11	4	1
2017	10	6	2	22	18	20
2018	no data			17	28	15
2019	15	14	7	no data		
2020	21	23	36	no data		
2021	18	30	31	55	43	57
2022	18	56	32	5	28	11
2023	28	42	40	48	56	76
2024	47	38	44	58	84	63
2025	46	31	45	68	41	75
2026	no data			8	11	3

When we analyse the influence of the original programming language on individual metrics (see Table 18), we find clearer indications of why migrations from C and C++ perform better. Total code volume and complexity (*LOC* and *Cyclomatic Complexity*) generally deteriorate for projects migrating from TypeScript, JavaScript, Python and other high-level languages. In contrast, these metrics tend to improve for projects migrating from C, C++ and Go. A similar trend is visible for *LOC/Unit* and *Cyclomatic Complexity/Unit*, suggesting that the phenomenon cannot be fully explained by the idea that C, C++ and Go projects tend to be more mature and thus have fewer additions later on than projects using other languages.

The *Unsafe Density* metric shows the expected language-dependent pattern. Projects migrating from C and C++ generally show decreases in unsafe operations, while increases mostly occur in projects migrating from memory-safe languages. Since these higher-level languages contain few, if any, strictly memory-unsafe operations, the introduction of Rust's 'unsafe' blocks will register as an increase from a baseline of nearly zero.

Metrics such as *% Duplication*, *Documentation Density* and *Mean Distribution Size* also vary according to the original language, suggesting that the original language paradigm influences how these specific quality attributes change during migration.

Finally, an examination of migration strategies (see Table 19) shows that the top-down approach does not have an overall advantage over the bottom-up approach in terms of any single metric. Rather, it appears to be a cumulative result of marginal improvements across multiple metrics. Only the *Mean Distribution Size* and *Technical Debt Density* metrics show more projects improving when working top-down than bottom-up.

Table 18: Metric changes categorised by original programming language, using post-migration data (or during-migration if unavailable). Desired trends are marked ↗ (maximise) or ↘ (minimise). Colours show project counts with significant increases (blue), decreases (orange), or no change (pink).

	Metric	C and C++	Go	TS and JS	Python	Other
GitHub	Defect Issue Count ↘	4 8	6	1 1 3	1 3	3
	Defect Issue Resolution Time ↘	2 2 8	2 4	3 2	1 3	2 1
	Distribution Platform Count ↗	2 2	2 3	2 2	1 1	1 2
	Mean Distribution Size ↘	3	3 1 1	1 2 1	2	2 1
RCA	Assertions-Complexity Ratio ↗	10 4 2	6	4 2	3 2	2 1
	Cognitive Complexity ↘	4 10 2	2 4	5 1	4 1	3
	Cyclomatic Complexity ↘	7 8	3 2 1	6	4 1	2 1
	Cyclomatic Complexity / Unit ↘	6 9	2 4	4 2	1 4	2 1
	Documentation Density ↗	14	3 2 1	6	5	3
	LOC ↘	6 8 2	3 2 1	6	4 1	3
	LOC / Unit ↘	4 11	2 4	4 2	2 3	3
	Test Density ↗	11 3 2	6	4 2	3 1 1	2 1
	Unsafe Density ↘	5 9 2	3 1 2	4 2	2 3	2 1
SQ	% Duplication ↘	8 4	2 3 1	3 1 2	4 1	2
	Technical Debt Density ↘	6 5 2	2 4	2 2 2	3 2	1 1
Other	Contrib. Decayed Gini Coefficient ↘	8 6 2	3 2 1	4 2	4 1	2 1
	Contrib. Decayed Truck Factor ↗	2 5 9	1 1 4	1 3 2	1 1 3	3
	Contrib. Gini Coefficient ↘	7 6 3	3 2 1	4 2	3 2	2 1
	Contrib. Truck Factor ↗	2 2 12	1 5	2 4	1 1 3	1 2
	Mean Time Inducing to Fix Commit ↘	3 6 7	1 1 4	1 1 4	2 1 2	1 2

Table 19: Metric changes categorised by migration strategy. Desired trends are marked ↗ (maximise) or ↘ (minimise). Colours show project counts with significant increases (blue), decreases (orange), or no change (pink).

	Metric	Intermediate		Standalone	
		Top down	Bottom up	Top down	Bottom up
GitHub	Defect Issue Count ↘	1	1 2	4 11	2 7
	Defect Issue Resolution Time ↘	1	1 2	6 8	5 4
	Distribution Platform Count ↗	no data	1 1	2 6	2 3
	Mean Distribution Size ↘	no data	2	6 2	1 3 1
RCA	Assertions-Complexity Ratio ↗	1 2	3 1	13 3	8 2
	Cognitive Complexity ↘	2 1	2 2	7 9	7 4
	Cyclomatic Complexity ↘	2 1	2 2	10 6	8 3
	Cyclomatic Complexity / Unit ↘	1 1 1	1 3	7 10	5 6
	Documentation Density ↗	2 1	4	6 10	3 8
	LOC ↘	2 1	2 2	10 6	8 2
	LOC / Unit ↘	1 1 1	1 3	6 11	5 6
	Test Density ↗	1 1 1	3 1	13 2 2	8 2
	Unsafe Density ↘	1 2	2 2	8 5 4	6 3 2
SQ	% Duplication ↘	1 1	2 2	8 5 2	7 2
	Technical Debt Density ↘	1 1	3 1	7 5 3	3 5 2
Other	Contrib. Decayed Gini Coefficient ↘	1 1 1	3 1	9 7	6 4
	Contrib. Decayed Truck Factor ↗	2 1	2 2	2 4 11	4 6
	Contrib. Gini Coefficient ↘	1 1 1	3 1	7 7 3	6 4
	Contrib. Truck Factor ↗	1 2	2 2	2 3 12	9
	Mean Time Inducing to Fix Commit ↘	1 1 1	2 2	3 6 8	3 7

Chapter 7

Threats to validity

This chapter discusses potential threats to the validity of the research findings and is structured according to the guidelines provided by Wohlin et al. [4]. It evaluates the study's conclusion, internal, construct, and external validity. It acknowledges the limitations of automated data collection tools, the representativeness of the sample and the potential confounding variables in both the qualitative and quantitative analyses.

7.1 Conclusion validity

There are several issues of validity concerning the relationship between the applied "treatment" (migrating to Rust in this case) and the observed outcomes (changes in software quality).

Firstly, the reliability of the custom software used to collect measurements must be considered. Although these tools were verified through preliminary testing, there is still a margin of error in automated data collection. To mitigate the risk of misdetecting migrations, we manually verified the dataset and received corroboration from the authors during the interview phase. Automated collection was generally successful for the collection of our metrics, though there were some technical limitations. Specifically, SonarQube failed due to some legacy Java dependencies, and its licence is limited to one million lines of code. To ensure transparency, all instances of missing measurements have been documented.

Furthermore, the sampling strategy that we used introduced a selection bias, thereby limiting our ability to draw statistical conclusions about migrations as a whole. As the project selection relied on GitHub popularity metrics, the resulting sample does not strictly represent all existing Rust migrations. As demonstrated by Munaiah et al. [69], popularity is an precise filter for identifying "engineered" (i.e. actively maintained and intended for production use) software projects, but it has low recall compared to their proposed heuristic. This means our methodology excludes well-engineered but less popular projects. Consequently, our findings indicate trends within more prominent open-source projects rather than trends within all open-source projects. The reasons why we made this trade-off are discussed in Chapter 3.

Another threat to validity is the random heterogeneity of the selected projects. Due to the diverse nature of the dataset, including variations in project domains, developer experience and applied strategies, individual differences may outweigh the isolated effect of the Rust migration itself. To mitigate this confounding factor, our methodology analyses metric changes within individual projects rather than aggregating them before comparing across phases. Additionally, we address this issue by analysing patterns within specific characteristic groups. However, ethical research guidelines constrained this more granular analysis: to preserve participant anonymity, groups could not be highly fragmented, and direct correlations between interview responses and individual project metrics were avoided to prevent cross-referencing that could lead to identification.

Lastly, due to the resource constraints of a Master's thesis, a single author conducted the coding of the interview transcripts. Wohlin et al. [4] note that single-coder thematic analysis introduces the risk of subjective bias. To mitigate this, a conservative coding approach was adopted: rather than merging codes into broad themes, they were grouped to preserve the participants' original nuances, and individual explanations and quotes were included wherever possible.

7.2 Internal validity

Internal validity refers to the factors that may have influenced our results, besides those we explicitly controlled for.

Firstly, our analysis does not isolate the "rewrite effect". This means that some of the observed outcomes, including perceived experiences and measured metrics, may be due to the code being rewritten from scratch rather than to the specific choice of using Rust. While it is difficult to separate these two effects, analysing the results based on different migration strategies provides some context. We would argue that it is reasonable to assume that the "rewrite effect" is less prominent with a bottom-up approach than with a top-down approach.

Secondly, we did not control for the timing of the migration, which introduces a history threat. For instance, projects that migrated later may have had access to superior Rust tooling and libraries that were unavailable to earlier projects. To account for this, we included a breakdown of migrations based on the year they were completed.

A similar threat is maturation, which means that projects improve and refine themselves over time. A project is likely to be more stable once migration is complete than during the process itself. To guard against this, we strictly separated our analysis into "during migration" and "post-migration" phases.

Furthermore, it is likely that there is a selection bias regarding the projects that participated in our case study. This is not due to our selection criteria, but rather because developers who were already enthusiastic about Rust may have been more willing to volunteer for an interview. This bias could skew the qualitative results, making the perceived experiences more positive.

Lastly, our definition of the different migration phases is strictly divided, whereas, in reality, the start and end of a migration do not necessarily represent distinct points in time. To make our analysis feasible, we applied a heuristic to create mutually exclusive phases and labelled only two phases instead of three for a project when distinguishing between the "during" and "post" migration phases was difficult.

7.3 Construct validity

The threats to our construct validity are whether the metrics we applied actually measure what the participants intended to improve or what we intended to investigate.

Firstly, not every project had the same migration goals. This means that a measured deterioration in a specific metric may have been anticipated or considered an acceptable trade-off by the developers. To account for this, we separated the results of projects aiming for a specific goal from the total dataset. However, it is still possible that a project abandoned one of its goals during the migration without mentioning this in the interview.

As discussed in the methodology, another threat is that it is difficult to compare metrics across different programming languages. This is more challenging when languages use different paradigms, such as object-oriented versus functional programming. We attempted to mitigate this issue by selecting metrics that are generally more reliable across different languages, and where possible, this was supported by existing literature.

A more fundamental threat lies in the number of steps required to translate a project's real-world goal into a measurable metric. Firstly, the goal is described by the participant, who may overlook some nuances. Secondly, the description is transcribed and coded, which can introduce subjective interpretation. Finally, the coded goal is mapped to a limited selection of available metrics. Each of these steps can cause a slight shift in meaning, meaning that the final metric may not fully capture the original intention.

Lastly, the fact that so many migrations need to be manually verified highlights the low precision of our automated detection method. Although we are confident that our final dataset is highly accurate, meaning there are few false positives, the manual filtering process itself may have introduced reviewer bias. For instance, our verification process relied heavily on English keywords. Consequently, valid migrations in projects documented in other languages were likely overlooked, which reduces our recall.

7.4 External validity

External validity deals with how well our conclusions can be applied to the wider field of software engineering, rather than just to the specific projects that we studied.

One limitation of our study is that we only looked at open-source projects. Therefore, our conclusions may not be applicable to commercial or closed-source software. The team structures, goals and resources of open-source projects are usually different to those of

commercial companies. The reasons for migrating to Rust and the problems encountered during the process could be very different in a corporate environment.

Finally, the migrations that we analysed were subject to survivorship bias. We only considered projects that had successfully migrated to Rust and continued to use it. We did not consider projects that attempted migration but abandoned it, nor projects that switched to Rust and then back again. As our analysis only considered successful cases, it may overlook significant challenges associated with adopting Rust, potentially providing a more optimistic view of the migration process than is representative of the entire industry.

Chapter 8

Conclusions

This research proposes a novel technique for detecting software migrations. It involves tracking source code identifiers extracted from Abstract Syntax Trees (ASTs) throughout a project's lifetime, in order to identify migrations within a repository (in-repository) or between multiple repositories (cross-repository). Using this technique and manual verification, we successfully identified 285 migrations to Rust. Of these, we interviewed the project authors and analysed their project for 38 migrations.

Our findings suggest that participants primarily employ a standalone migration strategy, wherein the original codebase is replaced only once the new Rust implementation has reached a sufficient level of maturity. Furthermore, developers generally favour a top-down architectural approach, focusing on reimplementing high-level features rather than translating the original code line-by-line or function-by-function. For those considering a language transition for new projects, this highlights a key fact: cross-language migration is rarely a straightforward translation, but rather a process of architectural evolution that prioritises clean boundaries over legacy preservation.

The reasons for switching to Rust vary depending on the original programming language. While migrations from C and C++ are mostly driven by Rust's memory-safe design, projects originally written in languages such as Go, Python and JavaScript often migrate in the expectation of improving performance efficiency. Developers consistently cite expected improvements in maintainability as an important motivation across all observed original languages. This expectation reflects a broader trend within computer science: as codebases become more complex and larger, the focus is shifting towards languages that offer strict compile-time guarantees and rigid type systems, which help to reduce the load on developers.

Participants who have completed their migration report positive experiences, emphasising the perceived improvements in maintainability, reliability, and security. However, empirical measurements of the same projects reveal a more nuanced result. While the quantitative data shows that testing practices improve in many projects after migration, several structural metrics tend to deteriorate. Notably, a significant proportion of projects experience an increase in code duplication. Additionally, some projects migrating from inherently memory-safe languages incorporated `unsafe` Rust during the

migration, introducing potential memory safety risks that were not present in the original implementations.

For projects and teams planning future migrations, these findings serve as a caution. They show that adopting a systems-level language such as Rust requires a significant change in approach, such as satisfying the borrow checker or managing explicit safety boundaries. If the system isn't designed correctly from the outset, this can lead to issues such as duplication or further reliance on unsafe code. Ultimately, these results emphasise the important discrepancy between the perceived benefits of migrating to Rust and the measurable trade-offs involved in the process.

8.1 Implications

Our case studies show that practitioners faced significant upfront investments when migrating to Rust, particularly in terms of learning the language and redesigning projects to fit Rust-specific idioms. Once these challenges were overcome, developers reported feeling significantly more confident in their code post-migration and consistently perceived an improvement in software maintainability. Based on their experiences, participants emphasised the importance of dedicating time at the outset to designing a software architecture that aligns with Rust's ownership model, rather than directly translating the original code.

Beyond the development team itself, these findings present critical implications for project managers and organisations as a whole. While the data underscores a steep initial learning curve, from a risk management perspective, the long-term payoffs align with a broader societal push for secure-by-design software. Amidst an increased global focus on cybersecurity, organisations should weigh short-term productivity losses against the long-term financial and reputational benefits of proactively eliminating memory safety vulnerabilities. However, these security benefits must be balanced against empirical operational realities: our measurements highlight a number of risks, such as increased code duplication and the introduction of memory-unsafe code blocks, which teams must actively manage during a transition.

From a methodological perspective, this research introduces a novel approach to detecting software migrations by tracking the movement of identifiers within Abstract Syntax Trees (ASTs). Because this approach is language-agnostic by nature, it provides researchers with a robust means of identifying and investigating cross-language migrations in the wider field of software engineering. The datasets collected and published alongside this thesis serve to support future empirical work in this domain.

Furthermore, this work highlights a discrepancy between the subjective success perceived by developers and the objective results obtained using automated metrics. This discrepancy suggests one of two possibilities: either developer perceptions are influenced by qualitative factors that standard structural metrics fail to capture, or existing metrics are inadequate for evaluating code quality across fundamentally different programming

paradigms. Regardless of the underlying cause, these findings highlight an urgent need for more reliable cross-language evaluation tools to accurately assess software quality.

8.2 Limitations

The biases outlined in Chapter 7 constrain this research. Notably, our dataset is subject to survivorship bias, as it only includes projects that successfully migrated to Rust. Additionally, the qualitative findings are based on interviews with developers who volunteered to participate. This self-selection bias means that our participants may be more favourable towards Rust than the average developer, which could skew their feedback.

In terms of construct validity, the comparison of software metrics across different programming languages is a novel area with limited prior literature. A major limitation of cross-language comparisons is the inability to control for the 'rewrite effect'. Consequently, it is difficult to determine whether the observed changes in metrics are directly related to the properties of the Rust language or simply the result of rewriting a legacy codebase from scratch.

8.3 Future work

This study identifies several promising areas for future research.

Firstly, to address the limitations of survivorship bias, future studies could examine projects that attempted to migrate to Rust but failed or reverted to their original language. Although this would require adapting detection techniques to identify aborted migrations, it would provide a balanced perspective on the practical difficulties of adopting Rust.

Secondly, further investigation is necessary to determine the specific causes of the increased code duplication that has been observed in many Rust migrations. Future research could analyse whether this duplication is an inevitable consequence of Rust's idioms or whether it stems from developer behaviour.

Thirdly, future research should incorporate dynamic performance analysis. As dynamic runtime analysis was outside the scope of this static study, time behaviour and resource consumption during runtime could not be objectively measured. As many participants cited performance improvements as their motivation for migrating, runtime analysis is necessary to verify whether these expectations are met in practice.

Lastly, this study's methodology can be scaled up and adapted to analyse the migration of closed-source commercial software. Conducting the study within a corporate environment would help establish whether the motivations, challenges and metric outcomes observed in open-source projects can be applied to the commercial software industry. Specifically, future work could examine the impact of conflicting stakeholder incentives, such as the drive of a product manager for rapid feature delivery versus the desire of an engineering lead for safety and maintainability, on the strategy and success rate of commercial Rust migrations.

Bibliography

[1] MSRC Team, "A proactive approach to more secure code." [Online]. Available: https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code

[2] J. Vander Stoep, "Rust in Android: move fast and fix things." [Online]. Available: https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html

[3] J. Hong and S. Ryu, "Automatically Translating C to Rust," *Communications of the ACM*, vol. 68, no. 11, pp. 58–65, Oct. 2025, doi: 10.1145/3737696.

[4] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2024. doi: 10.1007/978-3-662-69306-3.

[5] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, in ICSE '14. ACM, May 2014, pp. 92–101. doi: 10.1145/2597073.2597074.

[6] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 69–80, Jun. 2003, doi: 10.1145/780731.780743.

[7] The Chromium Projects, "Memory Safety." [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[8] International Organization for Standardization, "Software engineering — Software Life Cycle Processes — Maintenance," Geneva, Switzerland, technical report ISO/IEC/IEEE 14764:2022, 2022.

[9] M. L. Brodie and M. Stonebraker, *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. in The Morgan Kaufmann Series in Data Management Systems. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995.

[10] N. Weiderman, L. Northrop, D. Smith, S. Tilley, and K. Wallnau, "Implications of Distributed Object Technology for Reengineering," Pittsburgh, PA, USA, technical report CMU/SEI-97-TR-005, 1997. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/ADA326945.pdf

[11] C. Verhoef and A. Terekhov, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000, doi: 10.1109/52.895180.

[12] E. Visser, “A Survey of Rewriting Strategies in Program Transformation Systems,” *Electronic Notes in Theoretical Computer Science*, vol. 57, pp. 109–143, Dec. 2001, doi: 10.1016/s1571-0661(04)00270-1.

[13] E. D. Demaine, “C to Java: converting pointers into references,” *Concurrency: Practice and Experience*, vol. 10, no. 11–13, pp. 851–861, Sep. 1998, doi: 10.1002/(sici)1096-9128(199809/11)10:11/13<851::aid-cpe385>3.0.co;2-k.

[14] J. Hong and S. Ryu, “Type-migrating C-to-Rust translation using a large language model,” *Empirical Software Engineering*, vol. 30, no. 1, Oct. 2024, doi: 10.1007/s10664-024-10573-2.

[15] M. C. Feathers, *Working Effectively with Legacy Code*, 14. print. in Robert C. Martin Series. Upper Saddle River, N.J: Prentice Hall, 2013.

[16] D. A. Garvin, “What Does ``Product Quality" Really Mean?,” *Sloan Management Review*, vol. 26, no. 1, pp. 25–43, 1984, [Online]. Available: https://sloanreview.mit.edu/article/what-does-product-quality-really-mean/

[17] J. A. McCall, P. K. Richards, and G. F. Walters, “Factors in Software Quality,” Griffiss Air Force Base, NY, USA, technical report RADC-TR-77-369, 1977. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/ADA049014.pdf

[18] International Organization for Standardization, “Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product Quality model,” Geneva, Switzerland, technical report ISO/IEC 25010:2023, 2023. [Online]. Available: https://www.iso.org/standard/78176.html

[19] G. Starke and P. Hruschka, “ISO/IEC 25010:2023 Detailed Product Quality Model [Diagram].” [Online]. Available: https://quality.arc42.org/images/articles/iso-25010/ISO-25010-2023-detailed.png

[20] V. R. Basili and D. M. Weiss, “A Methodology for Collecting Valid Software Engineering Data,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, Nov. 1984, doi: 10.1109/tse.1984.5010301.

[21] D. Binkley, “Source Code Analysis: A Road Map,” in *Future of Software Engineering (FOSE ’07)*, IEEE, May 2007, pp. 104–119. doi: 10.1109/fose.2007.27.

[22] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[23] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, Jul. 2002, doi: 10.1145/567793.567795.

[24] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, “Have things changed now?: an empirical study of bug characteristics in modern open source software,” in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, in ASPLOS06. ACM, Oct. 2006, pp. 25–33. doi: 10.1145/1181309.1181314.

[25] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, in SIGSOFT/ FSE'14. ACM, Nov. 2014, pp. 155–165. doi: 10.1145/2635868.2635922.

[26] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, Apr. 2015, doi: 10.1007/ s10664-015-9381-9.

[27] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the Impact of Programming Languages on Code Quality: A Reproduction Study," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 4, pp. 1–24, Oct. 2019, doi: 10.1145/3340571.

[28] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, USENIX Association, Aug. 2021, pp. 597–616. [Online]. Available: https://www.usenix.org/ conference/soups2021/presentation/fulton

[29] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of Rust: a mixed-methods study," in *Proceedings of the 44th International Conference on Software Engineering*, in ICSE '22. ACM, May 2022, pp. 1269–1281. doi: 10.1145/3510003.3510164.

[30] A. Zeng and W. Crichton, "Identifying Barriers to Adoption for Rust through Online Discourse," Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, p. 5:1–5:6. doi: 10.4230/OASICS.PLATEAU.2018.5.

[31] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA: USENIX Association, Jul. 2024, pp. 425–443.

[32] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on Java library migrations: prevalence, trends, and rationales," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE '21. ACM, Aug. 2021, pp. 478–490. doi: 10.1145/3468264.3468571.

[33] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "MigrationMiner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Sep. 2019, pp. 414–417. doi: 10.1109/icsme.2019.00072.

[34] K. Tuusjärvi, J. Kasurinen, and S. Hyrynsalmi, "How Does Migration to Microservices Happen? A Survey of the Finnish Software Industry," in *Proceedings of the Annual Doctoral Symposium of Computer Science (TKTP 2024)*, Vaasa, Finland, Jun. 2024.

[35] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?," *Journal of Systems and Software*, vol. 169, p. 110710, Nov. 2020, doi: 10.1016/j.jss.2020.110710.

[36] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Mining revision histories to detect cross-language clones without intermediates," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, in ASE'16. ACM, Aug. 2016, pp. 696–701. doi: 10.1145/2970276.2970363.

[37] D. Perez and S. Chiba, "Cross-Language Clone Detection by Learning Over Abstract Syntax Trees," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, May 2019, pp. 518–528. doi: 10.1109/msr.2019.00078.

[38] M. Peters, G. L. Scoccia, and I. Malavolta, "How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps?," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, Sep. 2021, pp. 36–46. doi: 10.1109/scam52516.2021.00014.

[39] M. Martinez and B. Gois Mateus, "Why Did Developers Migrate Android Applications From Java to Kotlin?," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4521–4534, Nov. 2022, doi: 10.1109/tse.2021.3120367.

[40] B. G. Mateus, M. Martinez, and C. Kolski, "Learning migration models for supporting incremental language migrations of software applications," *Information and Software Technology*, vol. 153, p. 107082, Jan. 2023, doi: 10.1016/j.infsof.2022.107082.

[41] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, Oct. 2021, doi: 10.1145/3485498.

[42] M. Sirlanci, C. Yagemann, and Z. Lin, "C2RUST-BENCH: A Minimized, Representative Dataset for C-to-Rust Transpilation Evaluation." [Online]. Available: https://arxiv.org/abs/2504.15144

[43] OSS Insight, "OSS Insight Explore: GitHub Data Explorer." 2026.

[44] M. Brunsfeld *et al.*, "tree-sitter/tree-sitter: v0.26.8." [Online]. Available: https://doi.org/10.5281/zenodo.19357078

[45] A. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, in SEQUEN-97. IEEE Comput. Soc, pp. 21–29. doi: 10.1109/sequen.1997.666900.

[46] O. Ertl, "SuperMinHash - A New Minwise Hashing Algorithm for Jaccard Similarity Estimation." [Online]. Available: https://arxiv.org/abs/1706.05698

[47] H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Oct. 2016, pp. 334–344. doi: 10.1109/icsme.2016.31.

[48] P. E. Strandberg, "Ethical Interviews in Software Engineering," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, Sep. 2019, pp. 1–11. doi: 10.1109/esem.2019.8870192.

[49] C. Curtain, K. Dröge, J. Missaghieh–Poncet, and L. Salomón, "QualCoder." [Online]. Available: https://github.com/ccbogel/QualCoder/releases/tag/3.8.2

[50] R. van Solingen and E. Berghout, *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. London: The McGraw-Hill Companies, 1999.

[51] M. Tufano *et al.*, "There and back again: Can you compile that snapshot?," *Journal of Software: Evolution and Process*, vol. 29, no. 4, Dec. 2016, doi: 10.1002/smr.1838.

[52] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, in ISSTA '08. ACM, Jul. 2008, pp. 121–130. doi: 10.1145/1390630.1390647.

[53] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994, doi: 10.1109/32.295895.

[54] L. Ardito *et al.*, "rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, Jul. 2020, doi: 10.1016/j.softx.2020.100635.

[55] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/tse.1976.233837.

[56] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou, "A mapping study on design-time quality attributes and metrics," *Journal of Systems and Software*, vol. 127, pp. 52–77, May 2017, doi: 10.1016/j.jss.2017.01.026.

[57] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, in ICSE '10. ACM, May 2010, pp. 385–394. doi: 10.1145/1806799.1806856.

[58] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating Truck Factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, IEEE, May 2016, pp. 1–10. doi: 10.1109/icpc.2016.7503718.

[59] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "Measuring and analyzing code authorship in 1 + 118 open source projects," *Science of Computer Programming*, vol. 176, pp. 14–32, May 2019, doi: 10.1016/j.scico.2019.03.001.

[60] E. Jabrayilzade, M. Evtikhiev, E. Tüzün, and V. Kovalenko, "Bus factor in practice," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE '22. ACM, May 2022, pp. 97–106. doi: 10.1145/3510457.3513082.

[61] Software Improvement Group and TÜV NORD, "SIG/TÜV NORD CERT Evaluation Criteria Trusted Product Maintainability," Amsterdam, Netherlands, technical report Version 17.0, Mar. 2025.

[62] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *6th International Conference on the Quality of Information and*

Communications Technology (QUATIC 2007), IEEE, Sep. 2007, pp. 30–39. doi: 10.1109/quatic.2007.8.

[63] H. Alrubaye, D. Alshoaibi, E. Alomar, M. W. Mkaouer, and A. Ouni, "How Does Library Migration Impact Software Quality and Comprehension? An Empirical Study," in *Reuse in Emerging Software Engineering Practices*, Springer International Publishing, 2020, pp. 245–260. doi: 10.1007/978-3-030-64694-3_15.

[64] G. A. Campbell, "Cognitive complexity: an overview and evaluation," in *Proceedings of the 2018 International Conference on Technical Debt*, in ICSE '18. ACM, May 2018, pp. 57–58. doi: 10.1145/3194164.3194186.

[65] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test Code Quality and Its Relation to Issue Handling Performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, Nov. 2014, doi: 10.1109/tse.2014.2342227.

[66] R. Kikas, M. Dumas, and D. Pfahl, "Issue Dynamics in Github Projects," in *Product-Focused Software Process Improvement*, Springer International Publishing, 2015, pp. 295–310. doi: 10.1007/978-3-319-26844-6_22.

[67] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE '18. ACM, Oct. 2018, pp. 908–911. doi: 10.1145/3236024.3264598.

[68] MITRE, "CWE-1399: Comprehensive Categorization of Memory Safety Issues." [Online]. Available: https://cwe.mitre.org/data/definitions/1399.html

[69] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Apr. 2017, doi: 10.1007/s10664-017-9512-6.

Appendix A

Verified project set

Table 20: Projects that are verified to contain migrated code to Rust and were interviewed.

Project	Original project	Original language	Domain	Stars	Forks	Size	% in Rust
BitBoxSwiss/bitbox02-firmware		C	System	349	127	8025432	26.7
KiwiTalk/KiwiTalk		TypeScript	App	738	90	414612	49.9
NixOS/nixos-search		Python	App	538	150	302594	31.9
OISF/suricata		C	System	6116	1678	19570385	24.0
SeaDve/Kooha		Python	App	3317	93	218788	96.2
amir20/dtop		Go	App	1008	26	289386	95.7
apache/arrow-rs	apache/arrow	C++	System	3425	1140	15798256	99.6
appsinacup/godot-rapier-physics		C,C++	Non-web lib	835	55	1323477	70.8
ariebovenberg/whenever		Python	Non-web lib	2320	33	2281508	41.2
brndnmtthws/dryoc	jedisct1/libsodium	C	Non-web lib	332	19	582529	100.0
cjdelisle/cjdns		C	System	5377	600	3135547	12.4
cooklang/cookcli		Swift	App	1225	87	871446	56.3
eraserhd/parinfer-rust	parinfer/parinfer.js	JavaScript	Tool	615	50	185026	73.6
fosskers/aura		Haskell	App	1886	121	996894	29.7
gschup/ggrs	pond3r/ggpo	C++	System	640	32	248936	100.0
intentee/paddler		Go	System	1512	83	928338	69.5
ixy-languages/ixy.rs	emmericp/ixy	C	System	316	37	442127	99.9
jely2002/youtube-dl-gui		JavaScript	App	7951	585	696322	37.4
jneem/nnnoiseless	xiph/rnnoise	C	Non-web lib	339	18	101691	90.3
kkoomen/vim-doge		TypeScript	App	1061	53	165475	70.7
kubetail-org/kubetail		Go	App	1640	121	1326636	13.1
libjxl/jxl-rs	libjxl/libjxl	C++	Non-web lib	528	36	2308240	98.3
m-labs/artiq		C	System	513	225	3992940	21.0
max-baz/wluma		C	Non-web lib	907	41	143357	98.0
memorysafety/rav1d	videolan/dav1d	C	Non-web lib	619	68	12750965	17.1
nautechsystems/nautilus_trader		Python	App	21723	2613	52130425	64.2
neondatabase/appdotbuild-agent		Python	Tool	751	113	2575667	22.8
noahbald/oxvg	svg/svgo	JavaScript	Web lib	523	10	1613750	97.5
numaproj/numaflow		Go	System	2446	151	6733234	57.6
oreboot/oreboot	coreboot/coreboot	C	System	1767	114	1523680	96.0
pubky/pkarr		JavaScript	Non-web lib	409	41	360824	79.6
rustfs/rustfs	minio/minio	Go	System	24341	1045	13999575	98.3
spieglt/FlyingCarpet		Go	App	5018	234	295634	50.9
stacked-git/stgit		Python	App	648	71	1643860	58.9
tectonic-typesetting/tectonic		C,C++	System	4672	194	5930744	27.2
trifectatechfoundation/zlib-rs	madler/zlib	C	Non-web lib	617	41	1369020	89.0
typedb/typedb		Java	System	4275	364	6098498	98.0
yuezk/GlobalProtect-openconnect		C++	App	2053	234	266270	85.5