Under Review 2025

ANONYMOUS AUTHOR(S)*

In the Rust programming language, functional-style code is considered idiomatic, but is idiomatic always best? We recruited participants to review code written with functional and imperative approaches, analyzing participants' ability to find bugs and the time required for review. We also analyzed open-source code to understand how usage of techniques from each paradigm compare across different contexts. Imperative styles generally required participants less time to review, but mutation significantly increased review times. Functional styles such as iterator methods and higher order functions are more prevalent in contexts where correctness is highly important, but function-level immutability is not as necessary to maintain.

CCS Concepts: • **Software and its engineering → Language features**; *Software development techniques.*

Additional Key Words and Phrases: Rust, Code Reviews, Programming Paradigms, Functional Programming, Imperative Programming

ACM Reference Format:
Anonymous Author(s). 2026. Under Review 2025. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (FSE '26).* ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Writing clear and well-structured code is fundamental to building software that is reliable, maintainable, and extendable over time [3]. Improved code clarity in software engineering helps minimize bugs, simplify modification, and support effective collaboration [5]. To maximize code quality and clarity, developers are often encouraged to follow a programming language's idioms—the commonly used code patterns and practices [36, 45]. However, it is not clear how programming language designers should choose which idioms to recommend or whether those idioms are consistently the most understandable and maintainable programming style.

The question of idioms is particularly important in multi-paradigm languages, where programmers must choose which paradigm to use for each section of code. Rust, for example, supports both functional and imperative styles. Rust is a constantly evolving language that has been voted by programmers as the "most admired" since 2016 [38], making analysis of this question relevant, important, and timely. New results could be adopted by the Rust community, resulting in codebases that are easier to read and maintain. The topic is particularly relevant for Rust, which may be harder to understand than other languages [16] and for which other researchers have studied type system-related comprehension questions [9]. We seek to develop and apply methods that would provide evidence regarding which choices would improve readability of Rust code.

Often, readability studies assess readability by asking participants questions about specific aspects of behavior, such as program output or memory management, by asking participants to fill in blanks or answer closed-ended questions [13, 27, 37, 42]. We are interested in a more holistic approach that encourages participants to understand *all* important aspects of the code.

FSE '26, July 5–9, 2026, Montreal, Canada
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

To encourage our participants to analyze code thoroughly and in a realistic manner, we ask them to *review* code snippets. This reflects modern peer code review practice, which is a widespread practice in industrial software engineering; reviewers check both for bugs and for opportunities for stylistic or design improvement to code [1]. Compared to well-written code, confusing code should take longer to review (because it is harder to understand) and produce more critical comments from reviewers, who would suggest ways of making the code easier to read. Code review quality and cost have become even more important with the rapid adoption of LLM-based code generation, since LLM-generated code may be incorrect or otherwise unfit for use as is [25].

To gather data regarding readability and usability of each paradigm in Rust, we recruit Rust programmers to complete code reviews to look for bugs and refactoring opportunities in provided code snippets. Then, we analyze the time required to complete the reviews and studied the quality and content of the reviews. We prepare functional- and imperative-style versions of each code snippet and randomize which version of each snippet participants see so that each participant sees only one style or the other for a given snippet.

To understand how programmers use functional and imperative language features, we analyze a total of 15M functions to detect these features among open-source codebases. To observe expert use cases from large, popular Rust codebases, we analyze 939 programs from *Awesome Rust*, a list of commendable software written in Rust, curated by the Rust community [35]. Additionally, we analyze single-developer codebases to compare language feature use with larger codebases.

In this study, we address the following research questions:

RQ1: How do functional and imperative Rust compare in comprehension tasks? We hypothesize that functional-style Rust will make it easier to find bugs during code review.

RQ2: In which contexts is an imperative style more commonly used than a functional style, and vice versa? We hypothesize that functional-style code will be more prevalent in domains where correctness is highly critical.

Overall, we found that imperative styles were generally more readable. Iteration resulted in significantly shorter code review times than functional approaches, reducing the times by 22%. Mutation had a statistically significant impact on review times, increasing the times by 87%. `match` statements were faster to review on average than `if/else` trees, but the difference was not statistically significant. Higher-order functions and closures were *slower* to review than the imperative alternative on average, but the difference was not statistically significant. Participants' opinions, as expressed in their reviews, corresponded with design choices that would minimize review time in the case of `match`, but recommended the opposite regarding functional-style iterators and closures for performance and idiomatic reasons.

By analyzing large open-source codebases, we found three correctness-essential domains that contain significantly more use of iterator methods and higher order functions than others. For smaller single-developer codebases, we found that the large team-based codebases on average also contained a higher concentration of iterator methods, however, the solo codebases contain a significantly higher fraction of non-mutating functions. These results suggest that different contexts of Rust programs make use of individual functional language features at different frequencies, depending on the relative requirements for correctness and efficiency.

2 METHOD

Our study comprised (1) two Qualtrics-based code review quizzes to compare how programmers understand and critique functional vs. imperative Rust code, and (2) usage analysis of paradigms in public codebases. We incentivized participation with entry in a raffle for a $50 gift card.

2.1 Functional vs. Imperative Rust

The official Rust Programming Language textbook defines *iterators*, *closures*, and *pattern matching* as functional language features [24]. Additionally, Rust's *immutable by default* model leverages guarantees of immutability, an important pillar of pure functional programming [19]. Based on prior work on functional programming [20], we identified five signals that we used to distinguish imperative from functional programming in Rust: loops vs. iterator methods; mutating vs. non-mutating functions; pattern matching; and use of closures and higher-order functions.

Loops vs. Iterator Methods. Imperative Rust relies more heavily on explicit loops that involve mutation, whether they are written as `for`, `while`, or `loop`. Functional Rust, on the other hand, uses iterator methods to enable sequences of calls such as `map`, `filter`, `fold`, and `collect`. We analyze usage on a basis of whether a given function contains a loop, chain of iterator methods, or both. Other multi-paradigm languages such as JavaScript offer fewer iterator methods—JavaScript provides 11 static instance methods compared to Rust's 76. [28, 33].

Closures and Higher Order Functions. Rust's closures are first-class functions that implement one or more of the traits `Fn`, `FnMut`, and `FnOnce`. Closures can be passed into higher order functions to parameterize behavior, making code more modular and flexible. This enables concise and expressive patterns of computation that are common in functional languages. In contrast, the imperative style typically relies on instruction-driven logic and lacks first-class treatment of functions. Therefore, functional-style Rust will contain more functions that either take as a parameter or return closures.

Mutation. We identify *mutating* functions as those that mutate the state of data outside of their scope by receiving mutable references (`&mut`) as parameters. We also included functions that use the `unsafe` keyword because this enables code that dereferences raw pointers or otherwise escapes the safe semantics of Rust. In contrast, we consider functions with immutable inputs and outputs to be functional-style. Any effects of mutating local variables are encapsulated inside functions, so we do not consider this to affect whether a *function* is functional. While we consider mutating functions in our paradigm analysis, we consider local mutation of structs in our code review quiz.

Pattern Matching. Rust's pattern matching via `match` and `if let` enables expressive branching over values and algebraic data types, aligning with functional paradigms. In contrast, it is more common in imperative languages (which often lack pattern matching) to see more conditional branching. Although conditional branching is ubiquitous across functional and imperative programming, ideally functional code will make use of match statements whenever possible.

2.2 Code Review Quizzes

2.2.1 Quiz Structure. Each quiz consists of code snippets written with two paradigms of Rust programming: functional and imperative. Each question presents one snippet alongside a description of its intended behavior. We ask participants to review the code for potential improvements and bugs, and identify them in a provided input box. The instructions we provide are shown in Fig. 1.

The questions are randomly assigned, presenting a given code snippet in either functional or imperative form, and the order of questions is randomized to minimize any impact of question order. A hidden timer is embedded in each question to measure how long participants took to complete each question. Although we have no way to ensure that participants are completely focused on the tasks, any distractions are likely similar across experimental conditions, especially since each participant experiences both conditions across multiple questions.

We identify questions that deal with **I**teration as In, questions with **P**attern matching as Pn, **M**utation as Mn, and higher order functions containing **C**losures as Cn.

Review the code given, looking for any improvements that should be made before integrating this code into a large, stable codebase. Be sure to identify any opportunities for improving clarity, and if you find any bugs, mention those too.

Write your review in the text box below, explaining what changes you recommend making and why, along with their corresponding line numbers. You can also write high-level comments about the entire change. See the example below:

```
Line 1: [code review]
Line 2-5: [code review]

Overall: [comments]
```

If you believe no changes are necessary, be explicit by writing "LGTM" ("looks good to me").

Fig. 1. Code review instructions

2.2.2 Quiz 1 Content. The first Qualtrics quiz contains ten review questions, each with two versions (functional and imperative), with the exception of P2, which has two functional variants. In this quiz, we evaluate three hypotheses:

H1: Imperative loops (such as `for` and `while`) are more readable than functional approaches (`map`, `fold`).
H2: `Match` is more readable than `if` for cases with more than two branches.
H3: Mutation impedes readability relative to immutable designs.

(a) Question I1: Functional version

```rust
fn product_even_squares(numbers: &[i32]) -> i32 {
    numbers.iter()
        .filter(|&&num| num % 2 == 0)
        .map(|&num| num * num)
        .fold(0, |acc, x| acc * x)
}
```

(b) Question I1: Imperative version

```rust
fn product_even_squares(numbers: &[i32]) -> i32 {
    let mut product = 0;
    for &num in numbers {
        if num % 2 == 0 {
            product *= num * num;
        }
    }
    product
}
```

Fig. 2. Question I1 showed code that was intended to calculate the product of squares of all even numbers in a list. The functional implementation used map, filter, and fold; the imperative implementation used a `for` loop. Both versions included a bug: the accumulator is initialized at zero rather than one.

2.2.3 Quiz 2 Content. Inspired by trends observed in the first quiz (such as iter/loop questions having shorter task completion times in the imperative paradigm vs. the functional paradigm) and interested in exploring certain findings (such as a preference for pattern matching over if-else branching, and for loop logic over iterator methods) in greater depth, we develop a second set of code review questions for a follow-up quiz.

Q	Code Description	Bug
I1	This function is intended to calculate the product of squares of all even numbers in a list.	The initial value of the product is 0 instead of 1.
I2	This function is intended to determine if a list contains any pair of elements that sum to a target.	There is a redundant `if`.
I3	This function is intended to print out a grid of random numbers.	Because we are starting the index at the current index of the outer loop rather than 0, we print a triangle instead of a grid.
I4	This function is intended to compute the Cartesian product of two lists (in other words, it returns all possible ordered pairs from two or more sets). For example, if A={1, 2} and B={x, y}, the Cartesian product A×B={(1, x), (1, y), (2, x), (2, y)}.	No bug.
I5	This function is intended to find the last occurrence of a specific error message in a log, in the form of the furthest right index where the error message is present, based on its timestamp and content.	No bug.
I6	This function is intended to print out a grid of 0s with diagonal 1s.	Actual output is a grid of 1s with diagonal 0s.
P1	This function is intended to match integer power levels to the PowerMode enum shown below.	We are missing the “Normal” variant condition check.
P2	This function is intended to assign grades to students based on their exam scores.	No bug.
M1	The following snippet defines a User struct and instantiates an example User.	In the function generating an example username and email, the username and email of one of the users is swapped.
M2	The following snippet defines a Circle struct and instantiates an example Circle.	No bug.

Table 1. Survey 1 questions

The format of the questions for the second quiz are identical to those of the first; we again have a timer, functional and imperative versions, code blocks, and free response code review boxes. This quiz is shorter than the first; where the first quiz has 10 questions, our second has six.

We evaluate three hypotheses:

H4: Pattern matching is more readable than if/else trees for complex algebraic data types.
H5: Closures and higher-order functions impede readability.
H6: Loops are more readable than iterators in cases where the loop needs to terminate early.

2.2.4 *Analytical Method.*

Quantitative Analysis. For our primary quantitative measurement, we collect the time taken to complete each question. Since the timer is hidden from participants, they are unaware of this measurement, preventing them from modifying their behavior based on time constraints. Because times are often not normally distributed, we use the Mann-Whitney U test to assess the timing differences between the paradigms of each question.

Q	Code Description	Bug
P3	The following snippet takes in a reference to a Point, shown below. If the point is on the x-axis, increase its y-value by one and return a new point. Otherwise, return none.	This code checks whether x is zero, not if it's on the x-axis (it should be checking if `y==0`).
C1	The following snippet creates a new vector that filters for numbers that are strictly above a given threshold.	We should only return values strictly greater than the threshold, but the code uses greater than or equal to instead.
C2	The following snippet implements a running average and passes the test case.	There is a cast after division that will drop the decimal part of our average.
I7	The following snippet takes a vector of strings and parses them into a vector of `i32`s, returning an error on the first failure to parse.	No bug.
I8	The following snippet returns the first two perfect squares.	No bug.
P4	The following snippet implements `Display` for `HttpResponse`.	No bug.

Table 2. Quiz 2 questions

Qualitative Analysis. Our qualitative data consists of participants' written code reviews where they provide feedback on the given Rust code snippets. These code reviews are free-text responses, which we manually analyze for the participants' correctness, thought processes, and engagement with different programming paradigms.

To analyze these written code reviews, we employ *thematic analysis* [4], a method widely used in qualitative research that focuses on identifying and analyzing patterns within data.

2.3 Recruitment

For recruitment, we reach out through "This Week in Rust," a popular newsletter catered for Rust programmers and the r/learnrust Reddit page, catered for beginners. We also ask a well-known Rust programmer to publicize our study via the Bluesky social network.

We follow the same channels for recruitment for the second quiz, sending followup emails to the people who provided their emails in the first quiz. Additionally, we advertise our study through the Rust Programming Language Community Discord server.

2.4 Paradigm Use Analysis

2.4.1 Analysis Tool Design. This work is part of the development of Situationally Adaptive Language Tutor (SALT), a VS Code [26] IDE extension to perform longitudinal programming studies. In order to interpret Rust code, SALT calls a Rust compiler plugin built on top of `rustc-plugin` [6], based on the implementation of Flowistry, another Rust IDE tool [10]. When executed on a program at compile time with `cargo`, it takes inferences from the program's intermediate representations, both high-level and mid-level (HIR and MIR), to identify functional and imperative language features. These include locations of features within functions, types of iterator methods, and input and output types of functions. While the SALT plugin can be executed within the VS Code extension to monitor code changes, it can also be run independently to analyze any compilable Rust projects.

2.4.2 Analysis Method. To understand how paradigm usage is reflected in real-world Rust programs, we use the SALT plugin to analyze different kinds of open-source repositories:

Large, Popular Rust Codebases. To acquire code that reflects high standards for Rust programming, we based our selection on projects in *Awesome Rust*, a curated list of commendable Rust frameworks, libraries, and software grouped by application context [35]. We analyze 939 codebases across Gitlab, Github and Crates.io that successfully compile with `cargo` using the SALT plugin.

Using the context provided by *Awesome Rust*, we organize them into 11 broader contextual domains. For example, cryptography and authentication libraries fall under Security while cloud and email fall under Networking. From there, we identify three domains that pertain to *computing infrastructure*—Development Tools, Language Design, and Systems—based on their necessity for correctness to support other computing domains [17]. Based on the arguments that functional programming promotes correctness [7] and that those who value correctness will take steps to promote it, we hypothesize that these domains will make heavier use of functional language features than others.

Solo-developer Rust Projects. We also consider single-developer (or "solo") projects to compare paradigm use. In this context, code review and code quality are likely less of a focus to maintain correctness and readability by collaborators. We analyzed 312 recent solo projects, updated as of September 1, 2025 which contain a majority of Rust code and are at least one year old. In these codebases, we hypothesize that they will contain less functional language features on average compared to the average of the large, popular codebases.

3 RESULTS

3.1 RQ1: Functional and Imperative Readability

3.1.1 *Quiz 1 Results.*

Demographics. 241 participants started our quiz and 39 participants fully complete it. Our participants have an average of 13 years of programming experience and 85% have a college degree. They have a median of three years of Rust experience, and every participant who completed the quiz has written at least 100 lines of Rust before. Over half of our participants have some experience in functional languages, but 77% of participants who have used a functional language such as Haskell report they are only at a "Novice" level. The vast majority of our participants (84%) are in industry.

Quantitative Results. Figure 3 shows the distribution of code review times. Before analysis, we remove results that were beyond two standard deviations from the mean to ensure that we do not include outliers in our data, as those participants likely did not complete the study in a focused way. On average, we find that functional tasks take 210 seconds to complete (a median of 176 seconds) and imperative tasks take 203 seconds to complete (median of 170 seconds). Details of the time distributions by paradigm for each task are shown in Figure 3.

To compare completion times between the functional and imperative paradigms across all tasks, we fit a linear mixed model with task as a random effect to account for baseline differences in task difficulty. The difference was not statistically significant ($p = 0.814$), indicating that the paradigm does not affect the time required for code reviews across all questions put together.

Although there is no significant difference overall, this could be because either paradigm may perform better within different language features. Therefore, we also test separately for each hypothesis. Questions I1-I6 explore differences between iteration and loops, Questions P1 and P2 examine `if` statements vs. `match` structures, and Questions M1 and M2 explore aspects of mutation.

H1. For questions about iteration (I1-I6), we fit a linear mixed model. The average time to complete the functional paradigm is 266 seconds, and on average, imperative tasks take 59.9 seconds less than functional tasks, which is statistically significant ($p = 0.009$). Thus, for iteration questions, the imperative paradigm leads to significantly shorter task completion times. The effect size (Cohen's d) is 0.338, indicating a small to medium effect size.

For `if/else` vs. `match` (P1-P2) and mutation (M1-M2), we use ordinary linear regression instead of a linear mixed model due to the small task size, since there are only two questions each for these two hypotheses [39].

H2. For `if` statements vs. `match` structures (P1-P2), tasks take 134 seconds on average for the functional paradigm, with the imperative paradigm being about 31.6 seconds slower on average. However, this effect is not statistically significant (p=0.346), so we cannot conclude that there is a difference in time between the two paradigms.

H3. For mutation (M1-M2), the functional paradigm takes an average of 121 seconds, and the imperative paradigm is on average 105 seconds slower. The p-value of 0.01 indicates this result is statistically significant, suggesting that the imperative paradigm is slower to interpret compared to the functional paradigm. The effect size (Cohen's d) is 0.602, indicating a medium effect size.

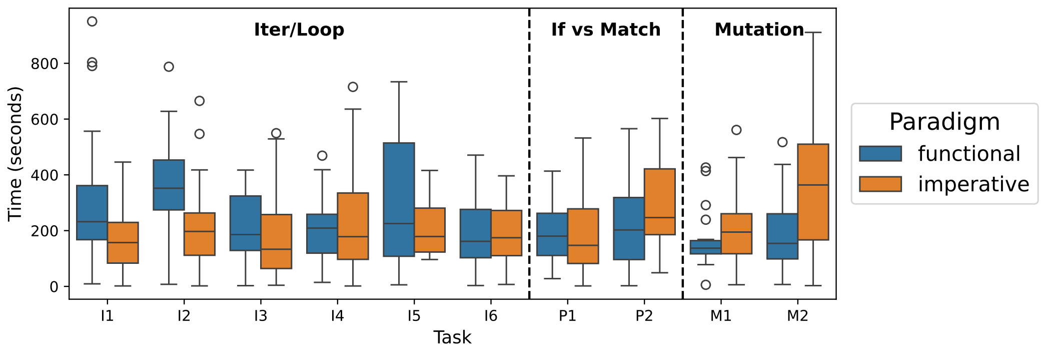


Fig. 3. Average code review times, separated by hypothesis

Bug identification. Figure 4 summarizes the percentage of participants who successfully identify bugs for each condition, split into four response categories: "Yes" (the participant identifies the bug correctly), "No" (the participant doesn't identify the bug), "Partially" (the participant finds either part of the bug or mentions a consequence of the bug that was not the bug itself), and "Identified Other Bug" (the participant finds a bug or improvement we did not intend for them to focus on).

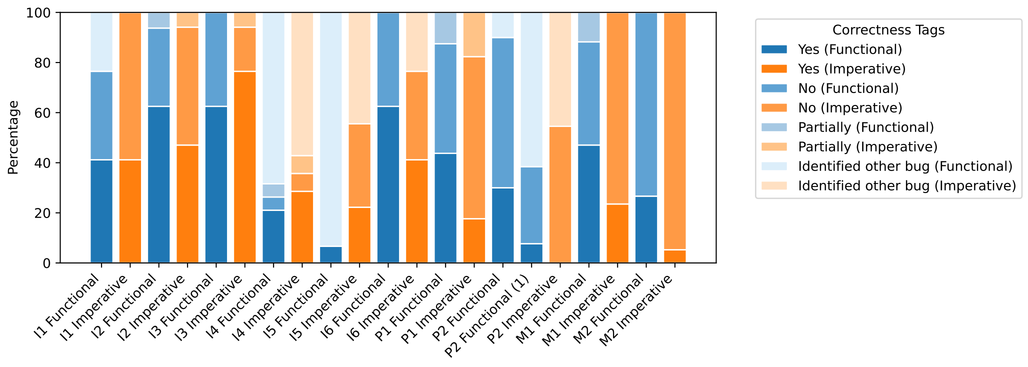


Fig. 4. Correctness rate of identifying bugs for each task

Table 3. Qualitative results of the first quiz

Theme	**Questions mentioned in**	**% of reviews mentioning**
Dislike of `fold`	I1 Functional, I2 Functional	41%
Iterator methods preferred over loops	I1 Imperative	59%
Loops preferred over iterator methods	I3 Functional, I6 Functional	45%
Simplifying iterators	I4 Functional	47%
Functional code preferred	I4 Imperative, I5 Imperative	40%
Pattern matching preferred over if-else	P1 Functional, P1 Imperative	61%
Use builder patterns	M1 Functional	39%
Avoid mutation in design patterns	M2 Imperative	32%

To investigate whether the condition affects participants' success in finding bugs, we perform hypothesis tests using Fisher's Exact Test. The results of the test reveal no significant associations between the functional & imperative conditions and the likelihood of correctly identifying bugs, as indicated by the p-values > 0.05 for all comparisons.

Qualitative Results. We conducted thematic analysis [4] by coding the reviews of our 39 participants and identifying recurring themes. Details about reviews are shown in Table 3.

Theme 1: Preference for Loop Logic Over Iterator Methods Many participants (6/17 for I2, 9/16 for I3, 9/19 for I4) express a clear preference for imperative loop constructs (i.e. `for` loops) over functional iterator methods such as `map`, `filter`, and `fold`. Participants often find looping logic to be more readable, especially in nested or multi-step computations. In contrast, participants in the functional condition (9/19 for I4) mention that iterators were unnecessarily complex and harder to follow for some tasks, specifically those involved in questions like generating grids or performing simple transformations.

Theme 2: Preference for Pattern Matching over If-Else Branching Participants favor Rust's `match` syntax over multiple `if/else` branches, since it is more idiomatic, concise, and expressive. Several comments note that `match` statements reduce the likelihood of missed cases and improve code maintainability.

Theme 3: Preference for Functional Design Patterns Over Mutable State Participants prefer to avoid storing values that could be derived from existing fields. They also suggest using enums or structured types to enforce constraints at compile time and avoid parsing or validation at runtime.

3.1.2 *Quiz 2 Results.*

Demographics. 39 participants (and four pilots) fully completed our second quiz. Our participants have an average of 12 years of programming experience and 64% have a college degree. Our participants have a median of four years of Rust experience, and every participant who completed the quiz has written at least 100 lines of Rust before. 73% of our participants are from industry.

Quantitative Results. Figure 5 shows the distribution of code review times for functional and imperative questions. We perform the same outlier removal as for quiz 1. On average, functional tasks take on average 239 seconds to complete (median of 317 seconds) and imperative tasks take on average 266 seconds to complete (median of 345 seconds). Details of the time distributions by paradigm for each task are shown in Figure 6.

We fit a linear mixed model with task as a random effect across the six tasks. On average, participants take 29 seconds longer to complete imperative tasks than functional ones, but this

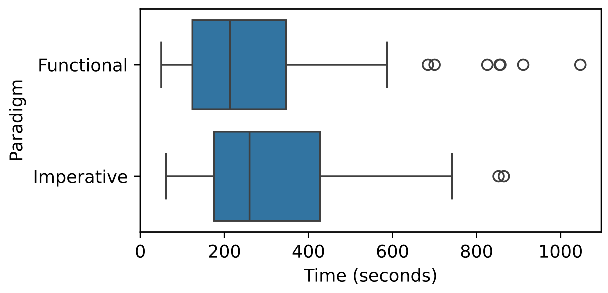


Fig. 5. Code review times by paradigm

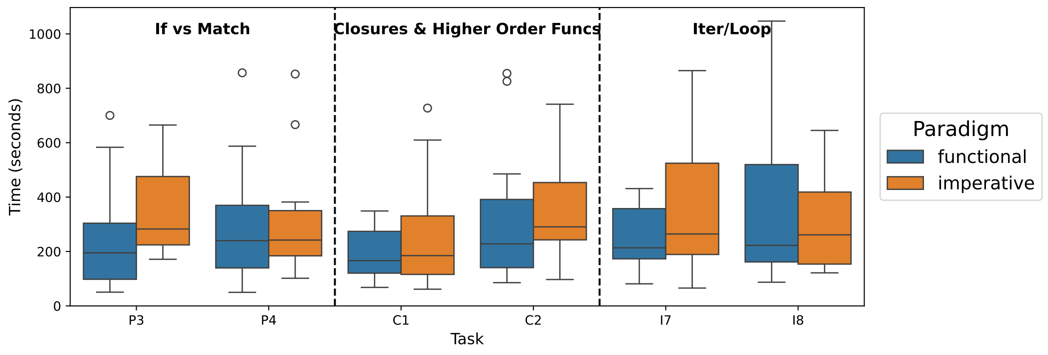


Fig. 6. Code review times by task

difference is not statistically significant (p = 0.375). To determine if there are any hypothesis-level differences, we test questions about `if/else` vs. `match` (P3, P4), closures and higher order functions (C1, C2), and iter methods vs. loops (I7, I8) separately using linear regression.

H4. For `if/else` vs `match`, the average time for the functional paradigm is 259 seconds, and the imperative paradigm is on average 66 seconds slower. The difference is not significant (p = 0.254).

H5. The functional approach (closures and higher order functions) average 177 seconds; the imperative paradigm is 46.9 seconds slower. The difference is not significant (p = 0.424).

H6. For iterators vs. loops, the functional paradigm takes 210 seconds on average, while the imperative 15.7 seconds slower. The difference is not significant (p = 0.807).

Bug identification. Figure 7 summarizes the percentage of participants who successfully identified bugs for each condition, split into two response categories: “Yes" (the participant identifies a bug correctly), “No" (the participant does not identify the bug).

After grouping questions across hypotheses across both quizzes, we perform Fisher’s Exact Test to determine if the condition affects participants in finding bugs for each hypothesis. The groupings are as follows:

- Iter/loop: I1, I2, I3, I4, I5, I6, I7, I8
- If vs. match: P1, P2, P3, P4
- Mutation: M1, M2
- Closures and higher order functions: C1, C2

The results of the test reveal that only the mutation group is likely to have an association between the functional and imperative conditions and the likelihood of correctly identifying bugs, with a

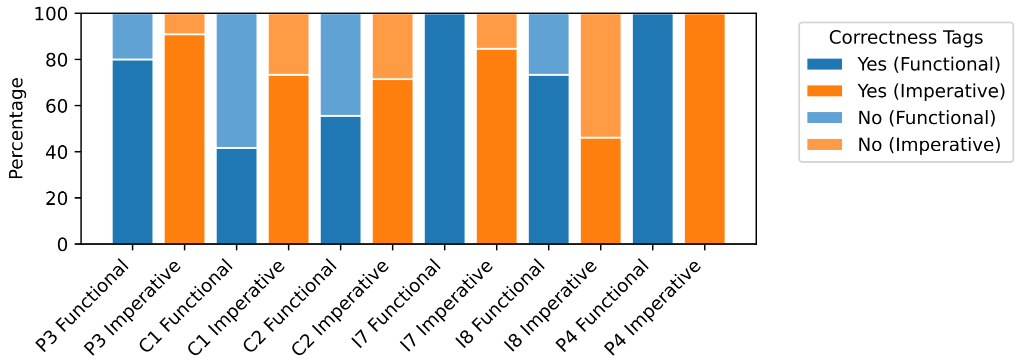


Fig. 7. Correctness by question

p-value of 0.048. However, it may be appropriate to interpret this result as *not significant* due to the need for a Bonferroni correction for multiple comparisons.

Qualitative Results. For questions with multiple sets of comments, we take the union of the codes of both reviewers. Table 4 shows details for the qualitative results discussed.

Theme 1: Preference for functional code Participants in both the imperative and functional versions of C1 mention preferring to use the functional iter-filter-collect chain (despite that the language feature we intended testing was a higher order function), arguing that this is less verbose and more idiomatic. This preference is also reflected in responses to the imperative version of I7, where participants express a desire for the code to be written more functionally using iterators.

Theme 2: Preference for iterators Many participants note a preference for iterators to avoid unnecessary allocations. Participants in the imperative version of I7 note that the code should be written more functionally with iterators to make the code more efficient. Participants in both functional and imperative versions of I8 also preferred iterators, noting this would be more flexible, concise, and readable.

Theme 3: Preference for closures In the functional version of C2, some participants find the use of a closure to maintain internal state unconventional, suggesting that using a struct would be more appropriate and idiomatic for Rust.

Theme 4: Preference for pattern matching In the imperative version of P4, nearly all participants recommend replacing the existing `if-else` logic with a `match` expression. Participants note this could reduce repetitive code, improve maintainability, and be more idiomatic.

3.2 RQ2: Functional and Imperative Paradigm Use

To understand how imperative and functional programming is used in large, popular codebases, we analyze repositories listed in *Awesome Rust*. From 1240 cloned popular Rust codebases, we compile 985 without errors with `cargo`. We remove 43 outliers with less than 25 lines of valid Rust code (excluding vertical whitespace and comments) leaving 939 codebases with an average of 22,968 lines each. Each analysis is normalized by codebase such that large-scale projects do not disproportionately influence the results. All together, the data encompasses 1,486,333 unique functions with an average of 1583 functions per codebase ($\sigma = 4677$).

We define the following domains from the contexts provided by *Awesome Rust*: AI/ML, Data Analysis, Graphics/Gaming, Media, Networking, Security Utilities, Web/Mobile, Dev Tools, Language Design, and Systems. The last three—Dev Tools, Language Design, and Systems—represent

Table 4. Qualitative results of the second quiz

Theme	**Questions mentioned in**	**% of reviews**
Functional code preferred	C1 Functional, C1 Imperative, I7 Imperative	75%
Imperative code preferred	I7 Functional	15%
Structs preferred over closures	C2 Functional	26%
More modular code for readability	C2 Imperative	29%
More code tests required	C2 Functional, C2 Imperative	40%
Iterators preferred	I8 Functional, I8 Imperative	41%
Match preferred over if-else	P4 Imperative	95%

computing infrastructure domains, which are contexts where correctness is especially critical, since they must maintain correctness across a broad range of conditions to support the function and development of the other domains. These are represented by 331 repositories, while the other domains are represented by 608 repositories in total.

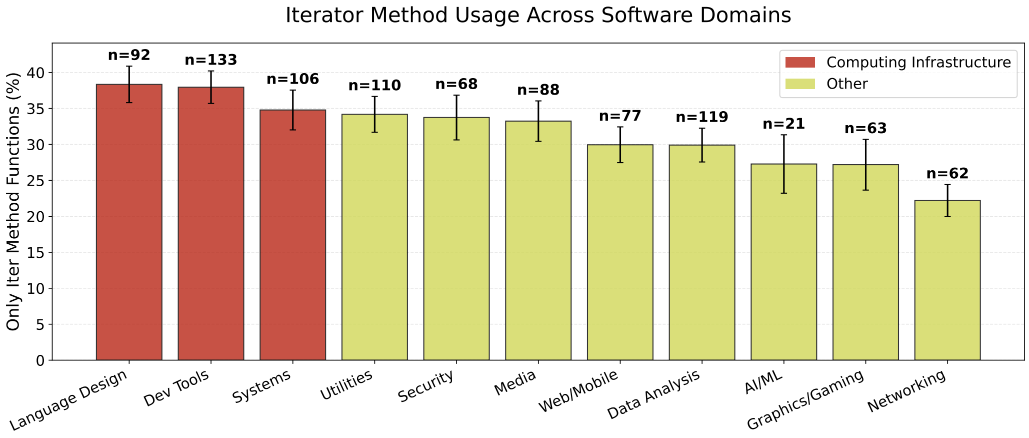
In the following experiments, we test our hypothesis that computing infrastructure domains contain higher concentrations of functional language features. While we are able to confirm that these domains use iterator methods and higher order functions more often than other contexts, we find no difference in the frequency of non-mutation functions or pattern matching.

3.2.1 Loops vs. Iterator Methods. To compare the use of functional and imperative approaches to performing iteration, we analyze 44,039 functions containing only iterator methods, 78,398 functions containing only loops, and 19,547 functions containing both. We examine the fraction of functions that contain only iterator methods of all functions that use iterator methods or loops.

The top three computing infrastructure domains correspond to the top three domains for highest concentration of pure iterator method use, as seen in Figure 8. On average, they contain 37.03% functions with purely iterator methods, while the other domains contain 29.71%, resulting in a difference of 7.32%. We found that the results are statistically significant using a t-test ($p = 0.018$), indicating that computing infrastructure domains lean towards *functional-style* in this aspect, writing a higher concentration of functions that purely use iterator methods for iteration than other domains with an effect size of 1.96 (Cohen's d). However, since this is an observational study, we cannot be sure of the cause of this difference.

As for the types of iterator methods used, all 11 of JavaScript's semantically similar iterator methods appear in the top 20 most commonly used Rust iterator methods overall when normalized by codebase. The top two most common Rust iterator methods, `collect` and `map`, have similar functionality to JavaScript's `toArray` and `map` respectively. The top four appear in over half of all codebases: `collect` (75.9%), `map` (73.9%), `next` (64.3%), and `enumerate` (53.7%).

3.2.2 Closures and Higher Order Functions. We identify higher order functions as ones that take a closure as a parameter or return a closure. Across 1.4M functions, we only found 12,217 higher order functions—only 0.82% of functions overall. However, computing infrastructure domains write higher order functions at a concentration of 2.13%, almost three times the concentration of other domains averaging at 0.76%. We found statistically significant results with a t-test ($p = 0.0024$) and effect size of 2.82, supporting the claim that computing infrastructure domains write higher order functions at a much higher rate than others. In Figure 8, we observe that Language Design contains more higher order functions on average at 3.00%.



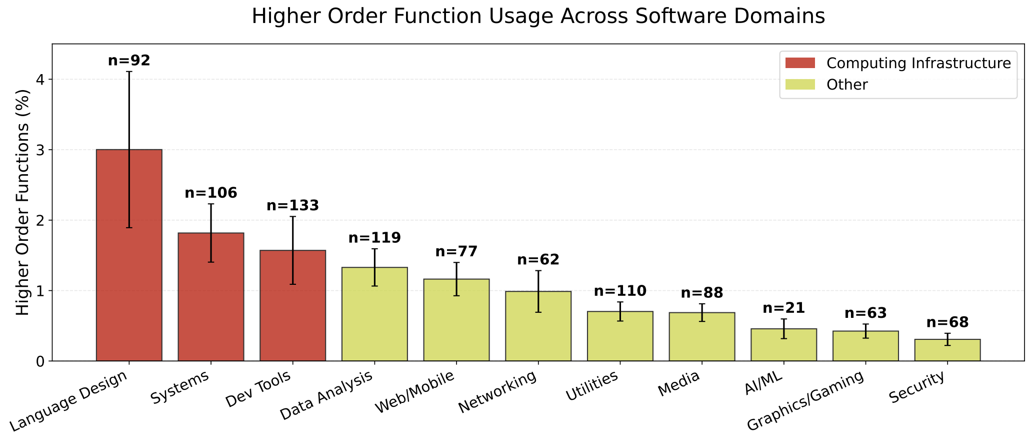


Fig. 8. Concentrations of functions that use only iterator methods and of higher order functions. Language Design, Dev Tools, and Systems domains rank in the top three of both. Error bars show standard error.

3.2.3 Mutating vs. Non-mutating Functions. Analyzing mutability and immutability at the function level, we look at signals in the function type to determine whether a function allows for mutation outside its scope. A function can mutate data if it is declared as `unsafe` or any of the parameters or return value contain a mutable reference or value. Analyzing 1.4M total functions, we detect 534,850 mutating functions and 951,483 remaining non-mutating functions. Figure 9 summarizes mutating vs. non-mutating function usage by software domain.

Only 67.8% of functions in these domains are non-mutating compared to 69.1% in other domains, a difference that is not significant ($p = 0.49$). Additionally, codebases in the Language Design domain contain the smallest percentage of non-mutating functions (63.0%).

3.2.4 Pattern Matching. We measure pattern matching use by detecting functions that contain `match` statements and `if let` expressions. Overall, we detect 340,191 functions using pattern matching out of 1.4M functions, including 299,483 `match` statements and 77,243 `if let` expressions. The difference between the two groups is not statistically significant ($p = 0.74$), with computing infrastructure domains consisting of 26.4% pattern matching functions and the remaining at

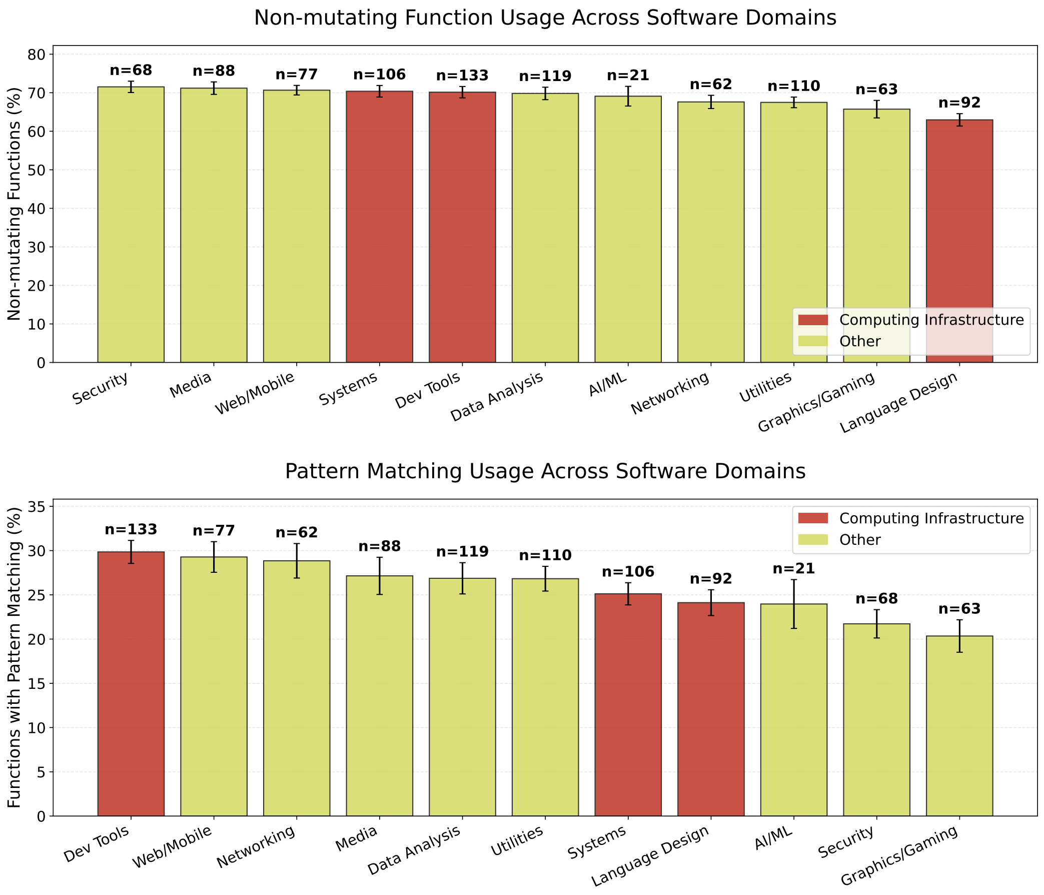


Fig. 9. Usage of non-mutating functions and functions containing pattern matching by domain. Error bars represent standard error of the mean.

25.6%, suggesting that pattern matching is used relatively frequently regardless of the correctness-criticality of context. Dev Tools uses the highest rate of pattern matching at 30.0%. Figure 9 summarizes pattern matching usage by software domain.

3.2.5 Paradigm Use in Solo Projects. The way individuals choose which paradigm to use in their solo projects can show how the average Rust programmer makes design decisions in their own work (where the stakes of correctness and readability might not be as high) compared to large teams working on large, widely used codebases. From 618 cloned solo codebases from Github, we successfully compile 939 from 298 users with cargo (A 50.6% success rate compared to *Awesome Rust's* 79.4%). The most recent commits of these codebases together consist of only 110,438 functions with an average of 359 functions per repository with high standard deviation (σ = 762). These results are not surprising, as solo projects are likely to be *much* smaller than popular codebases, with less emphasis on ensuring the project is rebuildable with just a cargo compile.

For popular codebases, functions containing iterator methods without any explicit loop statements appear in 31.0% of functions performing iteration. In the solo codebases, we identify an average concentration of only 21.9%, more than nine percentage points below the concentration

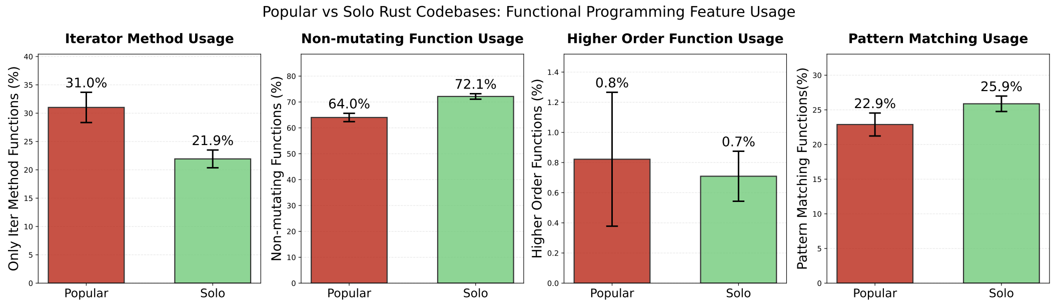


Fig. 10. Usage of language features in popular Rust projects vs. a sample of solo Rust projects. Error bars represent standard error of the mean.

in popular codebases. A t-test found a significant difference ($p = 0.0033$). This result again suggests that correctness-critical contexts use iterator methods on their own more frequently than other contexts. Meanwhile, we found that solo codebases use a significantly higher amount of non-mutating functions at 72.1% compared to popular codebases' 64.0% average ($p = 0.000026$). The solo average is higher than that of the most non-mutating domain we identified, Security, at 71.5%. For higher order functions and pattern matching, we do not find statistically significant results, but their relative percentages can be found in Fig. 10.

4 DISCUSSION

Our results suggest that whether functional or imperative code is "better" in Rust is highly context dependent. While Rust design pattern guides indicate that functional Rust is the idiomatic way [34], the readability and use varies by language feature and context of use.

Task-specific differences. The performance differences between paradigms were not uniform across tasks. For iteration-related questions (I1-I6), imperative loops led to significantly shorter completion times, suggesting that participants found loop-based reasoning more straightforward when identifying bugs. Despite this, Fig. 3 shows that the functional version of I6 (printing out a grid of 0s with diagonal 1s using `for_each()`) was completed slightly faster than the imperative version (using `for...in...`). However, for mutation-related questions (M1-M2), functional solutions led to faster completion, and for `if` vs. `match` (P1-P2), no significant difference was found. These findings indicate that each paradigm can provide advantages depending on the task. We hypothesize that loops help with step-by-step reasoning, while functional design reduces mutable state and therefore simplifies bug identification in code that would otherwise be heavily stateful.

Some questions showed a strong preference for one approach over the other. For example, I1 and I2 in Quiz 1 showed large differences in favor of the imperative approach, and many participants stated that `fold()` and other iterator-based approaches were unintuitive. M2 from Quiz 1 showed a strong preference for the functional approach, and participants noted that it is problematic for some variables to be mutable in the imperative version. Furthermore, participants performed very closely for both paradigms of I6 in Quiz 1, suggesting neither approach was far better than the other, reflecting the difficulties in designing "one-size-fits-all" guidelines for idiomatic use.

Preferences and opinions. From participant responses, we can glean some information on why some questions leaned towards one paradigm or another. Many reviewers explicitly stated that imperative loops were easier to follow for multi-step computations, while functional iterators could seem "too clever" for simpler tasks. Pattern matching was seen as more idiomatic than chained conditionals, and participants preferred functional approaches that eliminated the mutable state.

Iterator Method Readability vs. Actual Use. We noticed a discrepancy between longer times to do code review on iterator methods and use in areas where correctness is essential. If iterator methods are potentially less readable than looping logic, then there is concern for whether programmers' perception of the code they or others have written with iterator methods is correct. Especially for methods such as `fold`, where the name of the method is not as intuitive as that of `reverse`, there is an objective for programmers to learn and memorize these methods or avoid using obscure ones.

Mutating Functions and Ownership. The minimal difference between concentrations of non-mutating functions across different computing domains may demonstrate that regardless of context, the presence of mutating functions does not have as much of an impact on perceptions of correctness as we originally hypothesized. Perhaps Rust's ownership model alleviates programmers' concerns of mutability across function boundaries by guaranteeing memory safety.

On the other hand, we observed from the sample of solo projects that the average percentage of non-mutating functions was significantly higher than in the popular codebases. If solo projects are more likely to be novice Rust programmers, this could be explained by a preference to *copy* data rather than take an explicit mutable reference to avoid dealing with ownership rules [8].

Essential Functional Features. There are certain functional language features in Rust that appear to be essential to writing clear and concise Rust. Most of our code review quiz participants in the imperative groups for pattern matching suggested implementing it over `if/else`. With similar concentrations in real codebases (22.9% of popular codebases and 25.9% of solo), functions with pattern matching make up close to a quarter of Rust programs. This data reinforces the claim that pattern matching is an idiomatic practice in Rust.

5 LIMITATIONS

While our study offers insights into Rust code review and programming practices, there are several limitations that should be considered when interpreting the results.

Participant Sample Bias. Because of the small sample size and dependency on volunteer sampling, our participant pool may be skewed toward self-motivated learners or language enthusiasts rather than an unbiased sample of all professional Rust developers. However, because of Rust's relatively nascent standing, self-motivated learners may be overrepresented in the Rust population in general compared to the general developer community. A future, larger Rust community may be different.

Code Snippet Generalizability. The code review tasks were intentionally simple and short to make the review process feasible in a short amount of time. As a result, our results may not generalize to complex code located in large, real-world Rust databases. However, they may serve as a good proxy, since our experiment concerns the differences between conditions and not absolute times.

Time-Based Measurement. Review tasks are not representative of all tasks that programmers undertake; they may be particularly different from *authoring* tasks. Also, review time relates to comprehension speed, but the relationship may be confounded by other factors. More experienced developers may spend more time reviewing to give a more comprehensive response, while less experienced developers may speed through the quiz as they have far less feedback. However, participants were randomly assigned to the functional and imperative categories so any variation in experience should be evenly distributed across groups.

Compilation Success Rate. A portion of the cloned repos failed to compile in both the popular and solo codebases—20.6% and 49.4% respectively. The most frequently failing contexts of popular codebases are Databases (26/84 failed) and Blockchain libraries (21/44 failed) which mainly rely on external dependencies that could not be installed from Rust's package manager, `cargo`. For solo codebases, there is likely less of a requirement for reproducibility and stable commits, hence the higher compilation error rate.

6 RELATED WORK

Code Quality and Review Studies. The importance of high-quality code in the efficiency of software development is well established [3, 5, 42], with details such as variable naming [14], structural regularity [22], and usage of types [18, 29] all affecting programmer productivity. Many quantitative code quality studies have focused on showing a relationship between implementation choices and programmer efficacy as measured by task completion times or task correctness rates [13]. Stamelos et al. [40] took another approach by relating properties of software to user satisfaction, finding that average function size of an application was negatively correlated with end-user satisfaction ratings.

Bacchelli et al. [1] interviewed, surveyed, and analyzed code reviews by programmers at Microsoft. While motivations for performing code review lie in finding defects and code improvement, code review comments frequently emphasize style and architecture. However, a separate study found that code that is committed *without* review is more than twice as likely to contain defects [2]. Jahncke [21] studied how code review quality was affected by cyclomatic complexity of the reviewed code snippet. Missed defects and frustration were positively correlated with higher complexity, but limitations due to the environment (such as lack of pressure) may impact the applicability of the results in a realistic code review setting.

Empirical Studies of Language Usage. Several other studies conduct static analysis to detect language features among open-source codebases. Dyer et al. [12] designed a language, Boa, and used it to analyze the ASTs of 31k Java projects to understand adoption and frequency of use of language features. They demonstrate the adoption curve and eventual saturation of new feature use over several years by measuring whether or not a feature appears in a particular file, and found the adoption of new features to be limited, suggesting the design of recommendation systems to suggest refactoring code to use these new features to prevent bugs.

For Python, Peng et al. [31] developed PYSCAN to recognize language features and analyzed *why* certain features were used among 35 popular Python projects. Looking at feature usage normalized by the number of functions in a project, they discovered that several safety features were rarely used and that projects from different domains use different features. Dong et al. [11] studied 1.3M general Bash scripts and 14k top Github-starred Bash scripts to find common language features and bugs. They found that feature use between the general and top-starred scripts was similar, while 80% of the general and 50% of the top-starred scripts contained code smells.

Paradigm Studies. Historically, quantitative experiments regarding programming language design have been scarce [23]. However, comparative studies between imperative, functional, and multi-paradigm languages give insight into the benefits and downsides of each paradigm in the context of different languages.

Poos et al. found that ownership and explicit declarations of mutability in Rust help programmers find the location of an erroneous side effect compared to Java [32]. The challenges of finding the source of incorrect side effects correspond with the preference our participants expressed to avoid mutation when possible. However, it is possible that usage of mutation could be *less impactful* in Rust than in other languages due to Rust's ownership and mutation rules and that our participants are biased due to their experience with other languages.

Pankratius et al. [30] compared Scala, a multi-paradigm language, with Java, a primarily imperative language, in a randomized, counterbalanced, within-subjects study of four-week projects. They found that while Scala programs were more concise, the average debugging time was double that of Java. However, the results were influenced by their participant demographics: students with high levels of expertise in Java but novice experience in Scala. Mirolo et al. [27] compared the difficulty of tasks involving the reversibility of conditional statements in Java and Scheme among students who had novice experience in both languages. Explicit `else` branching in the functional language, Scheme, resulted in higher accuracy and comprehension of the tasks compared to implicit else (with no body) in the imperative condition.

Within multi-paradigm languages, Uesbeck et al. [41] studied the impact of lambdas in C++. They did not find a difference regarding time spent programming, compiler errors, or ability to complete tasks across all experience levels. Floor [15] conducted code comprehension interviews in the multi-paradigm language Kotlin, and found that multi-paradigm programs become less comprehensible when the boundary between the functional and imperative side is unclear. In addition, they identified that professionals struggled more than student participants on programs that were not purely imperative.

Additional Rust Studies. Rust research such as Crichton et al.'s [9] conceptual model for ownership measured the effectiveness of their additions to the Rust textbook via open-ended questions targeting ownership misconceptions. Through A/B testing, they found that their intervention made a positive, statistically significant difference in the percent of correct quiz answers by participants.

Empirical Rust studies like Yu et al. [44] focused on language features that relate to bugs in the Rust programming language itself. They analyzed 10k bug reports to find the majority arise from data types, expressions, and assignment statements, while Rust-specific features such as traits and ownership account for relatively fewer bug reports. On functional vs. imperative language features in particular, Xu [43] found that 10 of the top 15 frequently used features of scientific Rust programs are functional, demonstrating that Rust's functional features are commonly used even outside of software engineering contexts.

7 FUTURE WORK AND CONCLUSION

Our empirical study provides useful evidence that programmers can use when writing software: generally, use iteration rather than functional approaches; avoid mutation; use pattern matching. These results also show the promise of using code reviews as a tool for comparing language designs and stylistic choices; compared with other approaches, code reviews may be more representative of a real-world task that requires participants to thoroughly understand code.

Each programming language design encompasses hundreds of different decisions. Code review-based empirical methods represent a promising approach that could enable gathering data on how relevant design decisions could affect readers of code. As LLM-based programming becomes more prevalent, programmers' time may focus more on reading tasks and even less on writing tasks, making code review an even better proxy for general software engineering work.

8 DATA AVAILABILITY

We provide a replication package that contains data from both the code review study and empirical paradigm analysis. We have made the questions for code review Quiz 1 and 2 publicly available. The quiz responses contain personally identifiable information so they could not be included. However, we have made the analysis scripts available.

We also make the datasets collected from *Awesome Rust* repositories and solo-dev Rust repositories available, with both the output of compiling with `cargo salt` in addition to the commit hashes and version numbers of each project at which point the SALT plugin was run. The raw plugin binary, `salt-ide`, is included for conducting new analyses with installation and use instructions.

REFERENCES

[1] Bacchelli, A., and Bird, C. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 712–721.

[2] Bavota, G., and Russo, B. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015), pp. 81–90.

[3] Boehm, B. W., Brown, J. R., and Lipow, M. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering* (Washington, DC, USA, 1976), ICSE '76, IEEE Computer Society Press, p. 592–605.

[4] Braun, V., and Clarke, V. Using thematic analysis in psychology. *Qualitative Research in Psychology* (2006).

[5] Buse, R. P., and Weimer, W. R. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis* (2008), pp. 121–130.

[6] Cognitive Engineering Lab. rustc-plugin, 2025.

[7] Cousineau, G., and Mauny, M. *The functional approach to programming.* Cambridge University Press, 1998.

[8] Crichton, W. The usability of ownership, 2021.

[9] Crichton, W., Gray, G., and Krishnamurthi, S. A grounded conceptual model for ownership types in rust. *Proc. ACM Program. Lang. 7*, OOPSLA2 (Oct. 2023).

[10] Crichton, W., Patrignani, M., Agrawala, M., and Hanrahan, P. Modular information flow through ownership. *CoRR abs/2111.13662* (2021).

[11] Dong, Y., Li, Z., Tian, Y., Sun, C., Godfrey, M. W., and Nagappan, M. Bash in the wild: Language usage, code smells, and bugs. *ACM Trans. Softw. Eng. Methodol. 32*, 1 (Feb. 2023).

[12] Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, Association for Computing Machinery, p. 779–790.

[13] Feitelson, D. G. From code complexity metrics to program comprehension. *Commun. ACM 66*, 5 (Apr. 2023), 52–61.

[14] Feitelson, D. G., Mizrahi, A., Noy, N., Shabat, A. B., Eliyahu, O., and Sheffer, R. How developers choose names. *IEEE Transactions on Software Engineering 48*, 1 (2022), 37–52.

[15] Floor, D. Code comprehension in the multi-paradigm environment kotlin, February 2024.

[16] Fulton, K. R., Chan, A., Votipka, D., Hicks, M., and Mazurek, M. L. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)* (2021), pp. 597–616.

[17] Gokhale, M., Gopalakrishnan, G., Mayo, J., Nagarakatte, S., Rubio-González, C., and Siegel, S. F. Report of the doe/nsf workshop on correctness in scientific computing, june 2023, orlando, fl, 2023.

[18] Hoppe, M., and Hanenberg, S. Do developers benefit from generic types? an empirical comparison of generic and raw types in java. *SIGPLAN Not. 48*, 10 (Oct. 2013), 457–474.

[19] Hu, Z., Hughes, J., and Wang, M. How functional programming mattered. *National Science Review 2*, 3 (2015), 349–370.

[20] Hudak, P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21*, 3 (Sept. 1989), 359–411.

[21] Jahncke, G. The influence of code complexity on review efficiency, effectiveness and workload in embedded software development. Master's thesis, University of Twente, July 2023.

[22] Jbara, A., and Feitelson, D. G. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, Association for Computing Machinery, p. 189–200.

[23] Kaijanaho, A.-J. Evidence-based programming language design: a philosophical and methodological exploration. *Jyväskylä studies in computing*, 222 (2015).

[24] Klabnik, S., and Nichols, C. *The Rust Programming Language.* 2024.

[25] Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems 36* (2023), 21558–21572.

[26] Microsoft Corporation. Visual studio code, 2025. Source-code editor.

[27] Mirolo, C., and Izu, C. An exploration of novice programmers' comprehension of conditionals in imperative and functional programming. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2019), ITiCSE '19, Association for Computing Machinery, p. 436–442.

[28] Mozilla Developer Network. Iterator (javascript reference), 2025.

[29] Ore, J.-P., Detweiler, C., and Elbaum, S. An empirical study on type annotations: Accuracy, speed, and suggestion effectiveness. *ACM Trans. Softw. Eng. Methodol. 30*, 2 (Feb. 2021).

[30] Pankratius, V., Schmidt, F., and Garretón, G. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. In *2012 34th International Conference on Software Engineering*

(ICSE) (2012), pp. 123–133.

[31] Peng, Y., Zhang, Y., and Hu, M. An empirical study for common language features used in python projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2021), pp. 24–35.

[32] Poos, L., Hanenberg, S., Gries, S., and Gruhn, V. A controlled experiment on the effect of ownership rules and mutability on localizing errors in rust in comparison to java. In *Proceedings of the 20th International Conference on Software Technologies, ICSOFT 2025, Bilbao, Spain, June 10-12, 2025* (2025), M. Mecella, A. Rensink, and L. A. Maciaszek, Eds., SCITEPRESS, pp. 410–421.

[33] Rust Team. *Trait std::iter::Iterator.* The Rust Foundation, 2025.

[34] rust-unnoficial. Rust Design Patterns, 2025.

[35] rust-unofficial. Awesome Rust, 2025.

[36] Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2018), ICSE-SEIP '18, Association for Computing Machinery, p. 181–190.

[37] Siegmund, J. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), vol. 5, pp. 13–20.

[38] Stack Overflow. Stack Overflow Developer Survey, 2024.

[39] Staff, G. Ordinary least squares (ols) regression in r, 2024.

[40] Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L. Code quality analysis in open source software development. *Information Systems Journal 12*, 1 (2002), 43–60.

[41] Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, Association for Computing Machinery, p. 760–771.

[42] Wyrich, M., Bogner, J., and Wagner, S. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv. 56*, 4 (Nov. 2023).

[43] Xu, B. Towards understanding rust in the era of ai for science at an ecosystem scale. In *2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE)* (2024), pp. 1450–1455.

[44] Yu, S., and Wang, Z. An empirical study on bugs in rust programming language. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)* (2024), IEEE, pp. 296–305.

[45] Zhang, Z. *Comprehensive Analysis of Programming Language Idioms and Their Application to Software Quality.* PhD thesis, The Australian National University (Australia), 2024.

Received 11 Sept 2025