# Repository-level Code Translation Benchmark Targeting Rust

Guangsheng Ou
ougsh3@mail2.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Mingwei Liu*
liumw26@mail.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Yuxuan Chen
chenyx677@mail2.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Xin Peng
pengxin@fudan.edu.cn
Fudan University
Shanghai, China

Zibin Zheng
zhzibin@mail.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China



## Abstract

Recent advances in large language models (LLMs) have shown significant capabilities in code translation, often evaluated using benchmarks like CodeTransOcean. However, these evaluations typically focus on simple, function-level translations without considering dependencies, which does not reflect the complexities of real-world software development. Further, their effectiveness in translating to newer, lower-resource languages like Rust in realistic scenarios is still under-explored.

To address this gap, we introduce first repository-level code translation benchmark comprising 375 tasks targeting Rust, complete with relevant dependencies. Using this benchmark, we study four state-of-the-art LLMs, analyzing their erroneous outputs to understand their performance in more complex translation scenarios. Our findings reveal that LLMs exhibit substantially worse performance (41.5%-56.2% Pass@1 drop of GPT-4) on repository-level translations compared to simpler tasks, highlighting limitations in existing evaluation methods. The model that performed the best is Claude-3.5, demonstrating the strongest translation capabilities in both basic functionality accuracy and several relevant additional abilities. Additionally, we discover that LLMs struggle with identifying language differences in complex tasks, and that increased dependencies correlate with greater translation difficulty.

## CCS Concepts

• **Software and its engineering → Software notations and tools**.

## Keywords

Repository-level Code Translation, Rust, Large Language Models,

**ACM Reference Format:**
Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level Code Translation Benchmark Targeting Rust. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Mingwei Liu is the corresponding author.


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Code translation, also known as code migration, refers to the process of porting a project from one programming language to another. This process is often driven by the need to adapt to different runtime environments, optimize performance, or enhance security [1–5, 40]. In today's rapidly evolving technological landscape, with the continuous emergence of new programming languages and architectural paradigms, such as Rust and Cangjie, there is a substantial demand for effective and efficient code translation. Efficiently migrating legacy code to new languages and environments remains a significant challenge for developers and organizations alike [22, 29].

In recent years, large language models have been increasingly utilized for tasks such as code generation and code translation, demonstrating promising results in translating code between widely used languages like Java and Python [34, 35]. However, their effectiveness in translating to newer, lower-resource languages like Rust is still under-explored. Rust presents unique challenges due to its limited training data and significant language-specific features that differ from those found in more established programming languages[6]. Consequently, the translation performance and generalization capabilities of large models for Rust might not be optimal.

Existing code translation datasets do include some coverage of Rust, but they tend to be overly simplistic and fail to capture the real-world demands of code translation in actual development scenarios [45]. Most existing datasets comprise pairs of source and target language code (usually at the function level[20, 25, 39, 49]) and accompanying unit tests to verify functional equivalence post-translation. However, the tasks used in current benchmarks often lack complexity, predominantly involving standalone functions without external dependencies [37, 45, 51]. Even the datasets that do extend to the file level generally only add minimal context to make the functions executable in isolation. Moreover, the sources of these datasets are typically online coding platforms or artificially constructed examples, which significantly diverge from real-world software development practices [20, 25, 37, 39, 45, 45, 49, 51]. In real-world scenarios, developers often maintain the original project architecture and incrementally translate individual functions while relying on the repository's context, which includes complex, inconsistent, and cross-file dependencies. These are aspects current

benchmarks fail to address but are critical in real-world development.

To fill these gap, we propose the RustRepoTrans, a repository-level code translation benchmark targeting Rust. This benchmark was constructed by curating 375 code translation tasks that include relevant dependencies, reflecting the complexities found in actual software development projects.

In our study, we investigate the performance of state-of-the-art LLMs on RustRepoTrans, focusing on their effectiveness in translating code with complex dependencies. We also assess how our benchmark presents greater challenges compared to existing benchmarks, analyzing the factors that influence task difficulty and the types of errors encountered by the models. Additionally, we explore essential capabilities that LLMs demonstrate beyond mere translation accuracy, such as noise robustness, syntactical difference identification and code simplicity.

Through our experiments, we find that existing LLMs struggle significantly with repository-level code translation compared to standalone tasks, the proportion of compilation errors reached as high as 94.8%, indicating that current evaluations do not adequately reflect their capabilities in real-world scenarios. We identify key challenges related to dependency handling and highlight that the complexity of generated code is indicative of translation effectiveness. Errors caused by function and variable dependencies accounted for a total of 61.9%. Overall, our findings provide critical insights into the limitations of LLMs in complex code translation tasks, paving the way for future improvements in model design and evaluation methodologies.

The contributions of this paper are summarized as follows.

- Introduction of RustRepoTrans: We present the first repository-level code translation benchmark, consisting of 375 tasks targeting Rust, which includes relevant dependencies, providing a more realistic evaluation framework.
- Evaluation of LLMs in Complex Scenarios: We conduct a comprehensive study of four state-of-the-art LLMs on this benchmark, revealing their performance in more challenging code translation tasks that reflect real-world software development scenarios.
- Error Analysis and Insights: We analyze the errors generated by the LLMs and categorize them into 10 error causes, offering insights into their limitations and the factors affecting their translation capabilities, particularly in handling dependencies.

## 2 Related Work

### 2.1 Code Translation

Code translation is a crucial tool in modern software development, enabling interoperability among different programming languages and facilitating code reuse. It is particularly important for updating legacy systems and integrating components written in incompatible languages. Studies show that code translation improves productivity by reducing the time and cost of manual translation [25, 37], while also aiding comprehension and maintenance in multilingual projects.

Historically, code translation relied on rule-based systems that used handcrafted syntactic and semantic rules. Early statistical machine translation (SMT) models improved accuracy by mapping frequent code patterns, as seen in datasets like Java-C and CoffeeScript-JavaScript [14, 31]. As research progressed, small neural networks and sequence models began to replace rule-based approaches, learning patterns from smaller datasets. However, early neural models, such as recurrent neural networks (RNNs), struggled with complex syntax and language-specific idioms [17, 47, 48].

Despite advances with large language models (LLMs), challenges persist, including incorrect translations and limited adaptability to coding styles [38, 51]. This has led to increased focus on evaluating translation quality and the impact of prompt engineering. Jiao et al. [20] developed the G-TransEval benchmark to assess translation complexity, revealing that models perform well on simple tasks but struggle with more complex ones. Similarly, Pan et al. [35] found that context-rich prompts can enhance translation reliability by an average of 5.5%. Prompt engineering strategies, such as those proposed by Yang et al. [46], utilize test cases to improve accuracy, particularly in challenging language pairs. Macedo et al. [27] emphasized the importance of consistent output formatting for accurate evaluation, advocating for controlled prompts and post-processing to ensure reliable benchmarking.

Our work introduces RustRepoTrans, which enables a more realistic evaluation of LLMs by incorporating repository-level dependencies and context. This approach allows us to better assess existing code translation techniques in handling complex code translation tasks.

### 2.2 Code Translation Benchmarks

Recent advances in neural code translation models have been fueled by high-quality, large-scale parallel datasets. However, existing datasets exhibit significant limitations in programming language coverage, scale, and balance, particularly for Rust. Many datasets focus narrowly on a few languages, such as the early studies between Java and C [31] and more recent work like CodeTrans [25] and AVATAR [8], which primarily cover Java and Python.

To address these gaps, some datasets, such as CoST [52] and CodeNet [37], have aimed to enhance language diversity and quality. However, issues like erroneous solutions and limited support for Rust persist. The latest benchmark, CodeTransOcean [45], expands coverage to 45 languages but still lacks adequate support for Rust, hindering comprehensive research in this area.

Furthermore, many existing benchmarks rely on data from question-and-answer websites, which do not reflect real-world software development contexts. These datasets fail to capture the complex dependencies and architectural considerations inherent in actual projects, where code translation often requires maintaining the original architecture and addressing intricate cross-file dependencies. This disconnect underscores the need for more representative benchmarks like RustRepoTrans, which aim to provide a more realistic assessment of code translation capabilities.

RustRepoTrans is the first repository-level code translation benchmark focusing on Rust, designed to account for complex dependencies and derived from real-world projects, distinguishing it from previous benchmarks.

# 3 RustRepoTrans Benchmark

We begin by introducing the benchmark format, followed by the construction method of RustRepoTrans, and finally, we present the resulting benchmark.

## 3.1 Benchmark Format

Each code translation task in RustRepoTrans consists of a pair of functions along with their relevant dependencies, formatted as <source function, target function, target function dependencies, target function test cases>, as shown in Fig. 1. The function pairs <source function, target function> represent functionally equivalent code snippets from the source and target languages, along with their respective file paths. The target function dependencies include elements such as function dependencies, data type dependencies, variable dependencies, and library dependencies related to the target function.

In this benchmark, the source function, target function signature, and associated dependencies serve as input, with the goal for LLMs to generate a complete target function. The correctness of this generated function can then be verified using the associated test cases.

## 3.2 Benchmark Construction Method

The construction process is divided into two parts: Functionally Equivalent Code Pairs Extraction and Dependency Extraction. This approach allows us to obtain functionally equivalent code pairs along with their corresponding dependencies and test cases from real open-source projects.

**Functionally Equivalent Code Pairs Extraction** In this part, we focus on extracting functionally equivalent code pairs (source function and target function), with the target function written in Rust, from GitHub projects. The process involves five steps, detailed as follows.

**Migration Project Selection.** In this step, we select projects that have been rewritten in Rust from other languages, specifically C++, Java, and Python, due to their popularity and likelihood of having such rewritten versions. These projects are expected to contain functionally equivalent function pairs between their Rust and original language versions. We identify suitable projects by searching GitHub for terms like "Rust version" and "implemented in Rust". focusing on larger projects (applying qualifier "size:>1000") to ensure sufficient function pairs. After locating these projects, we verify their versions through documentation and code review. In this way, we identify pairs of source project and target project.

**Functions Pools Extraction.** In this step, we extract all functions from a pair of projects (source and target). For the Rust target project, we focus on candidate functions with associated test cases to ensure verifiability. We first identify all test cases and then extract the functions they cover by analyzing the external functions invoked. For the source project, we extract all implemented functions for comprehensive coverage. This extraction uses the static code analysis tool tree-sitter [41]. As a result, we obtain two sets of functions: the source functions pool and the target functions pool with test cases.

**Similarity-based Candidate Function Pair Extraction.** In this step, we extract function pairs from the source and target function pools using a similarity-based approach. Developers often translate code at the function level while maintaining a similar structure across languages. Thus, equivalent pairs usually come from files with similar paths and function signatures [50]. For example, the Rust function $pbkdf2$ at $src/ecdh.rs$ corresponds to $PBKDF2$ in the C project at $src/ecdh\_support.c$. We use the BM25 algorithm [42] to calculate the similarity for each target function, identifying the top 10 candidate source functions. This produces a list of Rust target functions with their top-10 source function candidates.

**LLM-based Equivalent Function Pair Identification.** In this step, we identify the most equivalent function for each Rust target function from its top-10 source function candidates using an LLM, leveraging their code implementation and contextual information (such as file paths). LLMs like GPT-4 excel in code understanding [23]. The prompt used for this identification is shown in Fig. 2. We employ GPT-4o due to its effective balance of efficiency and performance. If none of the candidates are functionally equivalent, the LLM is instructed to select "None."

**Manual Verification.** Due to the limitations of LLMs, such as the potential for hallucinations, we manually verify the functionally equivalent pairs identified by the models to ensure their actual functional equivalence.

**Dependencies Extraction.** In this part, we complement the extracted pairs of functionally equivalent functions (source function, target function, test cases) with dependencies from the target projects. This approach adds a unique repository-level context for code translation, distinguishing real-world functions from those manually constructed or sourced from programming Q&A websites.

We classify dependencies into three categories: invocation function dependencies, invocation variable dependencies, and data type dependencies. To identify these, we perform static analysis on the entire project to extract custom functions, data types, and global variables. For the target function, we gather import statements, call function identifiers, variable dependencies, and data type dependencies. We match in-file dependencies and, using the import statements, match cross-file dependencies to compile a complete list for the function. Due to tree-sitter's limitations, we manually reviewed the automatically extracted dependencies to correct any errors or omissions, ensuring each function has a complete and accurate set of dependencies.

## 3.3 Resulting Benchmark

This process enables us to construct RustRepoTrans, which comprises 375 tasks for repository-level code translation. Detailed information is provided in Table 1, alongside comparisons with other existing benchmarks. The size of RustRepoTrans is comparable to most previous benchmarks, and each task includes a manually verified ground truth for translation and corresponding unit tests, achieving over 90% average test coverage, indicating a high quality.

RustRepoTrans has two key features that distinguish it from previous benchmarks: 1) it is the first code translation benchmark to consider repository-level dependencies; and 2) it specifically targets code translation to Rust in realistic programming scenarios. Next, we will discuss these features in more detail.

**Repository-level Dependency.** RustRepoTrans focuses on code translation tasks with rich repository context, setting it apart from

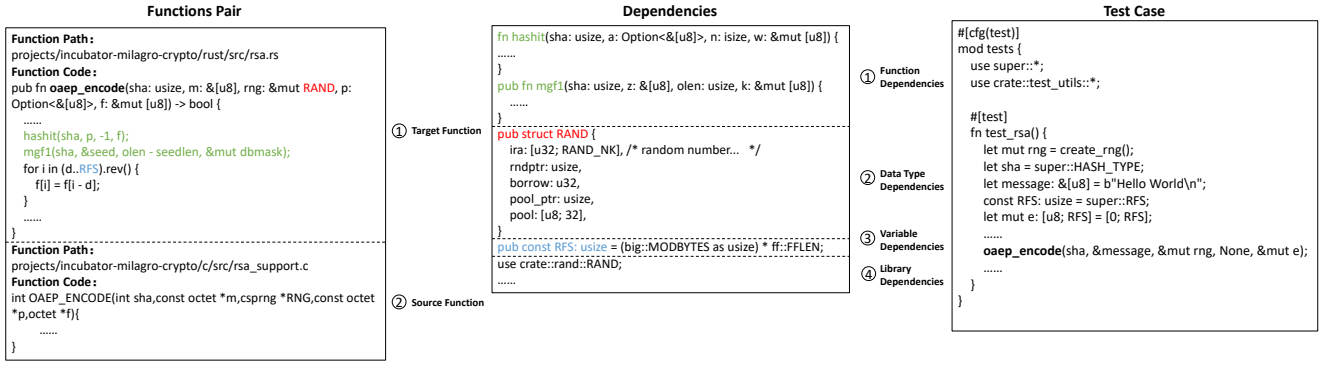


**Figure 1: RustRepoTrans Format Example**

**Table 1: Comparison of Different Datasets for Code Translation.**

| Dataset | Source | Task Level | #Tokens | # Tasks | Source Languages | With Dependencies? | Target Languages Include Rust? | Unit Tests Included ? | Golden Answer Verified? |
|---|---|---|---|---|---|---|---|---|---|
| CodeXGLUE[25] | Lucune, POI, JGit, Antlr | Function Level | 42.3 | 1,000 | Java, C# | ✗ | ✗ | ✗ | ✗ |
| XLCOST[51] | G4G | Program Level | 202 | 901 | C++, Java, C#, PHP, JavaScript, Python, C | ✗ | ✗ | ✗ | ✗ |
| TransCoder-test[39] | G4G | Function Level | 107.0 | 948 | C++, Java, Python | ✗ | ✗ | Partial | |
| HumanEval-X[49] | HumanEval | Program Level | 97.7 | 164 | C++, Java, Go, JavaScript, Python | ✗ | ✗ | ✓ | ✓ |
| G-TransEval[20] | HumanEval, G4G, .NET samples | Function Level | 95.1 | 400 | C++, Java, C#, JavaScript, Python | ✗ | ✗ | ✓ | ✓ |
| CodeTransOcean[45] | Rosetta Code | Program Level | 448.4 | 2,878* | Java, C++, C#, PHP, Python, Go | ✗ | ✓ | ✓ | ✓ |
| **RustRepoTrans** | Github | **Function Level** | **150.1** | 375 | C++, Java, Python | ✓ | ✓ | ✓ | ✓ |

*Translation pairs in languages that could not be parsed by tree-sitter to count the number of tokens were filtered out.

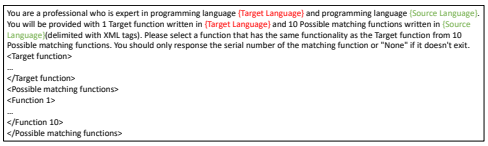


**Figure 2: The Prompt Used for Equivalent Function Pair Identification**

previous benchmarks. Unlike artificially constructed data or data sourced from Q&A websites, real project data exhibits more complex dependency relationships, including function, data type, and variable dependencies. Table 1 presents statistics on the presence of dependencies in existing datasets, revealing that only RustRepoTrans includes these crucial elements. This inclusion makes RustRepoTrans a more realistic benchmark, suitable for evaluating LLMs that must account for intricate file-level interactions and contextual dependencies.

**Rust Programming in Realistic Scenarios.** RustRepoTrans and CodeTransOcean are the only benchmarks specifically targeting code translation to Rust, as listed in Table 1. However, RustRepoTrans is derived from GitHub projects, making it more reflective of real-world development than CodeTransOcean, which relies on data from programming competition websites.

We analyzed 54 Rust keywords from the official documentation, comparing occurrences in CodeTransOcean(shown in Fig. 3) and RustRepoTrans (shown in Fig. 4) datasets. CodeTransOcean has 12 uncovered keywords, while RustRepoTrans has only 8, including critical asynchronous programming keywords `async` and `await`, which appear in 25.1% of the data. In term of keyword proportion differences between CodeTransOcean and RustRepoTrans (shown in Fig. 5), twelve keywords show negative differences, but only three exceed 6%: `extern`, `let`, and `use`. This is due to RustRepoTrans's function-level data, resulting in 0% for library-related keywords like `extern` and `use`. Additionally, RustRepoTrans 's real-world focus leads to less frequent use of the variable declaration keyword `let` compared to CodeTransOcean.

## 4 Evaluation

Based on RustRepoTrans, we further investigate the performance of the state-of-the-art LLMs on repo-level code translation task for Rust. Specifically, we focus on the following RQs.

- **RQ1 (LLMs Performance)**: How do the state-of-the-art LLMs perform on RustRepoTrans in terms of translation effectiveness?
- **RQ2 (RustRepoTrans Effectiveness)**: How effectively does our new benchmark pose greater challenges in code translation compared to existing benchmarks?
- **RQ3 (Difficulty Analysis)**: What factors affect the difficulty of the translation tasks?
- **RQ4 (Failure Analysis)**: What types of errors do LLMs encounter on RustRepoTrans, and what factors contribute to these translation failures?
- **RQ5 (Key Capabilities)**: Beyond translation accuracy, what essential capabilities do LLMs demonstrate on RustRepoTrans, including noise robustness, syntactical difference identification, and code simplicity?

### 4.1 Experimental Setup

**Model Selection.** We selected four state-of-the-art LLMs that have been widely studied in recent code translation research or were recently released and have demonstrated strong performance on

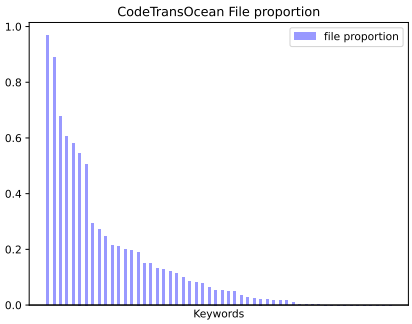


**Figure 3: the proportion of data in the CodeTransOcean that contains each keyword.**

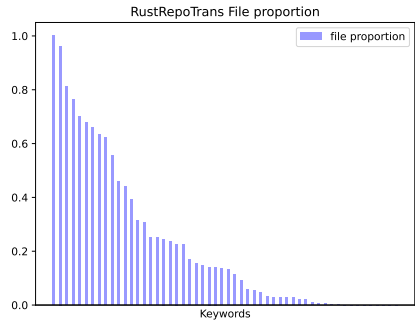


**Figure 4: the proportion of data in the RustRepoTrans that contains each keyword**

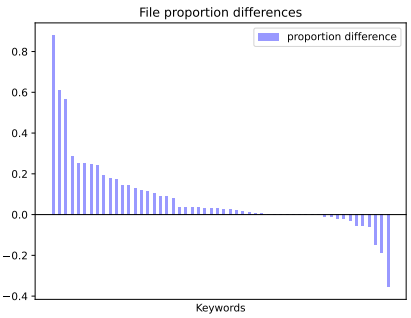


**Figure 5: the difference in the proportion of data containing keywords between RustRepoTrans and CodeTransOcean.**

benchmark leaderboards (*e.g.*,SuperCLUE[1] [44]). Table 2 provides an overview of the four LLMs used in our experiments, along with their release dates (Column "Time") and model sizes (Column "Size"). As shown in Table 2, our study includes both open-source and closed-source LLMs, as well as models designed for general-purpose and code-specific applications. Due to resource constraints, we selected open-source models with sizes below 20B parameters. Note that our primary contribution is a new benchmark designed to provide a more challenging and realistic evaluation of LLMs' code translation capabilities, closely simulating real-world development scenarios. Expanding the scope to include a larger and more diverse set of models is reserved for future work.

**Implementation Details.** For open-source models, we obtained and executed the released versions from their official repositories based on provided documentation. We used greedy sampling [13] as our generation strategy, generating a single solution code sample per task using greedy decoding (*i.e.*,setting the "do_sample" hyperparameter to false). All evaluations were conducted on an NVIDIA A800 80GB GPU. For closed-source LLMs, we accessed each model through its respective API interface [15, 33] as of September 2024. To achieve results similar to greedy decoding, we set the "temperature" hyperparameter to 0. Since some of our data contains extensive dependencies, resulting in very long prompts, we set the "max_tokens" hyperparameter to the upper limit supported by all studied LLMs: 8$k$.

## 4.2 RQ1: LLMs Performance

We evaluated four selected LLMs on RustRepoTrans to assess their translation accuracy and error-correction abilities for complex, repository-level code translation tasks targeting Rust.

*4.2.1 Design.* The evaluation involved testing each model on RustRepoTrans using controlled prompts, with output correctness assessed through specific test cases. The experimental design is structured as follows:

**Evaluation Process.** Each selected LLM was tasked with translating code into Rust, specifically focusing on 375 tasks. For each translation task, a corresponding set of test cases was employed to evaluate the correctness of the generated outputs. To ensure a fair comparison, the same prompt was used for each model, carefully designed based on established best practices in code translation tasks, as illustrated in Fig. 6. These best practices have been shown to enhance translation accuracy in similar studies [21, 28, 35, 46]. The prompt includes elements such as Instruction for translation and Translation's required information(including source code, target function signature and target function dependencies) Given that existing LLM-based code translation research often incorporates feedback on translation error messages to enhance performance [35, 46], further experiments were conducted to understand the extent to which errors in the generated code samples can be corrected by LLMs with feedback.

It is important to note that the goal is to evaluate the LLMs' intrinsic ability to self-correct translation errors, as well as to explore how many translation errors in RustRepoTrans can be automatically corrected by LLMs. Therefore, in addition to self-debugging, the best-performing model on RustRepoTrans, Claude-3.5, was also used to debug the translation outputs of all models. The debugging prompt (Prompt-Fix) shown in Fig. 6 includes Instruction for previous translation, Instruction for debugging, Incorrect translation and error details and Translation's required information, which are adapted from previous work [35, 46].

**Metrics.** Two key metrics were utilized to measure performance:

- ***Pass@k***: This widely-used metric [12] calculates the percentage of tasks correctly solved based on $k$ generated code samples per task, as expressed in Equation 1. A task is considered solved if at least one of the generated code samples passes all the corresponding test cases. Following recent research [35], the focus was on calculating the *Pass@1* metric, where $k = 1$. This approach reflects the model's ability to produce a correct translation on the first attempt.

$$\text{Pass@k} = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

- ***DSR@k***: The DSR@k (Debugging Success Rate@k) metric, proposed in prior work [45], evaluates whether the generated code successfully executes and produces the expected results (*i.e.*,passing all test cases) within $k$ rounds of debugging, as expressed in Equation 2. $S(i,k) = 1$ if the $i^{th}$ code sample succeeds within $k$ attempts; otherwise $S(i,k) = 0$. In this study, *DSR@k* metrics were calculated with $k = 1$, allowing

[1] https://www.superclueai.com/

**Table 2: Studied LLMs**

| Model Type | Model | Company | Open-source | Time | Size |
|---|---|---|---|---|---|
| General LLM | Claude-3.5 [9] | Anthropic | ✗ | 2024.6 | - |
| | GPT-4 [32] | OpenAI | ✗ | 2023.6 | - |
| | Llama-3.1-8B [30] | Meta | ✓ | 2024.7 | 8B |
| Code LLM | DeepSeekCoderV2-16B [16] | DeepSeek | ✓ | 2024.6 | 16B |

for one debugging attempt.

$$\text{DSR@k} = \frac{1}{N}\sum_{i=1}^{N} S(i,k) \tag{2}$$

*4.2.2 Results.* Fig. 7 presents the *Pass@1* and *DSR@1* performance of each LLM on RustRepoTrans. This benchmark challenges models with tasks that include complex dependencies, offering insights into each model's initial translation accuracy and post-debugging improvement potential.

**Initial Translation Performance (*Pass@1*).** In the initial code translation task, Claude-3.5 substantially outperformed the other LLMs, achieving 14.2% higher accuracy than GPT-4, 26.7% higher than DeepSeekCoderV2-16B, and 30.2% higher than Llama-3.1-8B on the *Pass@1* metric. This indicates Claude-3.5's comparative strength in handling repository-level code translation to Rust. However, even Claude-3.5 achieved only a 45.3% *Pass@1* score, emphasizing the difficulty of RustRepoTrans and the challenges inherent in translating code with complex dependencies.

**Post-debugging Performance.** After a single round of self-debugging, each model's *DSR@1* score showed significant improvement, confirming previous research that LLMs can effectively leverage compiler feedback for code translation accuracy [45, 46]. However, improvements from debugging by Claude-3.5 surpassed those from self-debugging, with Llama-3.1-8B showing only a 5.6% increase from self-debugging compared to a 24.8% improvement from Claude-3.5. This underscores Claude-3.5's superior debugging capabilities in complex code translation.

Post-debugging performance rankings revealed Claude-3.5 leading at 56.5%, followed by GPT-4 at 44.5%. Notably, Llama-3.1-8B (38.1%) surpassed DeepSeekCoderV2-16B (36.0%), indicating its initial errors were more correctable. Despite Claude-3.5 's strong performance, 44.5% of its cases remained unresolved, highlighting the inherent challenges of self-correction in intricate translation tasks. In terms of relative improvement, Llama-3.1-8B exhibited the highest percentage gain at 186%, indicating its initial errors were highly correctable. Conversely, Claude-3.5 showed the smallest improvement at 13.0%, suggesting its initial outputs were closer to correct execution. Overall, even after self-debugging, Claude-3.5 maintained its leading position, emphasizing its robust capabilities in repository-level code translation to Rust.

*4.2.3 Summary.* The results reveal that RustRepoTrans effectively challenges LLMs in complex code translation, as even the best

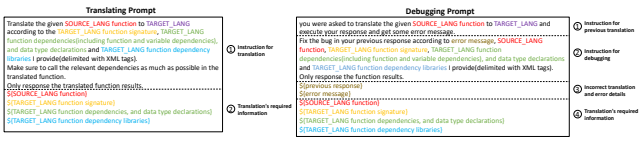


**Figure 6: Translating Prompt and Debugging Prompt**

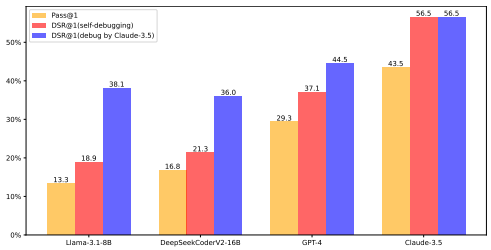


**Figure 7: Pass@1, DSR@1(self-debugging) and DSR@1(debug by Claude-3.5) on RustRepoTrans**

model, Claude-3.5, achieves a *Pass@1* of only 45.3%. After one round of debugging, its *DSR@1* increases to 56.5%, showing limited improvement.

## 4.3 RQ2: RustRepoTrans Effectiveness

To investigate the effectiveness of our newly constructed benchmark, we conducted a comparative analysis based on prior literature.

*4.3.1 Design.* To provide a comprehensive comparison, we specifically analyzed the model's performance on both RustRepoTrans and the previously established benchmarks. This analysis aims to demonstrate how our dataset serves as a more robust test of model capabilities in complex, real-world code translation scenarios.

**Benchmark Selection.** Our analysis is grounded in the work of Pan et al. [35], who conducted a thorough assessment of existing code translation benchmarks. They evaluated the performance of several prominent datasets, enabling us to contextualize our findings within the broader landscape of code translation research. Consequently, we compare our benchmark with datasets such as CodeNet [37], Avatar [8], EvalPlus [24], as shown in Table 3.

**Model Selection.** For our analysis, we selected GPT-4 due to its consistent ranking as the top-performing model across various datasets in prior studies [35]. While Claude-3.5 has emerged as a leading model in RustRepoTrans, it was excluded from our comparison because it is a recent development and was not included in the earlier assessments.

*4.3.2 Results.* The results presented in Table 3 reveal significant differences in the *Pass@1* scores of GPT-4 across various datasets. Notably, while GPT-4 achieves high accuracy rates—ranging from 52.2% to 85.5%—on established benchmarks such as CodeNet, Avatar, and EvalPlus, its performance drastically drops when evaluated on RustRepoTrans. The translation results of GPT-4 on datasets other than RustRepoTrans are directly taken from the findings reported in [35].

Specifically, the *Pass@1* scores for RustRepoTrans are significantly lower, with rates of 29.5% for C to Rust, 31.5% for Java to Rust, and 27.6% for Python to Rust translations, averaging 29.3%. This indicates a decline of approximately 41.5% to 56.2% compared to the

**Table 3: Pass@1 of GPT-4 in translating code from different studied datasets**

| dataset | Source Language | Target Language | Pass@1 (GPT-4) |
|---|---|---|---|
| CodeNet[37] | C | C++, Go, Java, Python | 83.0% |
| | C++ | C, Go, Java, Python | 80.0% |
| | Go | C, C++, Java, Python | 85.5% |
| | Java | C, C++, Go, Python | 81.3% |
| | Python | C, C++, Go, Java | 79.9% |
| Avatar[8] | Java | C, C++, Go, Python | 70.8% |
| | Python | C, C++, Go, Java | 52.2% |
| EvalPlus[24] | Python | Java | 79.3% |
| RustRepoTrans | C | Rust | **29.5%** |
| | Java | | **31.5%** |
| | Python | | **27.6%** |

higher scores observed on other benchmarks. This notable decrease highlights the increased complexity and unique challenges posed by RustRepoTrans, reinforcing the need for models to effectively manage dependencies and navigate intricate code structures in real-world. While GPT-4 is proficient in translating code across more straightforward datasets, it struggles significantly when faced with the complexities inherent in translating to Rust, further validating the robustness of our benchmark in assessing model capabilities.

Additionally, Rust, as a programming language, poses additional challenges due to its stricter syntax requirements and less abundant resources during LLM training compared to more popular languages like Python and Java. This results in a higher level of difficulty when translating to Rust, further validating the robustness of our benchmark in assessing model capabilities in tackling these complex coding tasks.

*4.3.3 Summary.* Our comparison reveals that our benchmark significantly increases complexity and dependency management, resulting in a notable decrease in *Pass@1* scores for GPT-4, from 85.5% to 29.3%. These findings indicate that our benchmark provides a more effective measure of model performance in translating intricate code structures and highlights the necessity for models that can better manage real-world project dependencies.

## 4.4 RQ3: Difficulty Analysis

To understand the challenges that RustRepoTrans presents to LLMs and how they contribute to performance drops, we analyze the code translation results of LLMs across different tasks.

*4.4.1 Design.* We analyzed both successfully and unsuccessfully translated tasks using Claude's results, chosen for its superior performance. Our goal was to identify the types of tasks where it succeeds or fails. Based on our dataset characteristics, we focused on three key factors that influence translation difficulty: the number of dependencies, and the lines of code in both the source and target languages. We conducted statistical analyses of these factors to assess their impact on translation success.

*4.4.2 Results.* Fig. 8 illustrates the distribution of successful and unsuccessful tasks with Claude based on dependencies and source/-target code lines. Successful translations are concentrated on the left side, indicating fewer dependencies and shorter code lines. In contrast, unsuccessfully translated pairs show a wider distribution. Specifically, failed translations have, on average, 138.7% more dependencies, 134.6% more lines of code in the target language, and 108.4% more lines of code in the source language compared to successful ones. This suggests that failed function pairs present more challenging translation tasks, resulting in lower success rates.

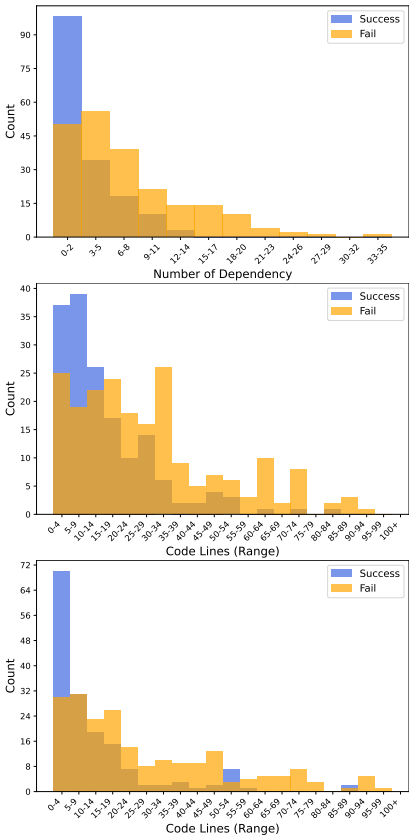


**Figure 8: Distribution of Successful and Unsuccessful Tasks with Claude Based on Dependencies and Source/Target Code Lines**

*4.4.3 Conclusion.* Higher dependencies and longer code lines challenge LLMs, resulting in lower translation success rates.

## 4.5 RQ4: Failure Analysis

To understand the errors causing performance drops in LLMs during complex Rust code translation, we analyzed the erroneous results generated by the models.

*4.5.1 Design.* The analysis comprises two parts: an automatic error outcome analysis and a manual error causes analysis.

**Automatic Error Outcome Analysis.** We collected all unsuccessful translations from LLMs in RQ1 (Section 4.2), resulting in 1,114 code samples that failed to pass all test cases. To identify where LLMs struggle in translating code to Rust, we conducted an automated analysis following the methodology of Pan et al. [35], categorizing the unsuccessful translations into four error outcomes: compilation errors (code fails to compile), runtime errors (code compiles but encounters exceptions), functional errors (code executes

**Table 4: The comparison of the proportions of error types between RustRepoTrans and [35].**

| | Source Language | Target Language | Compilation Errors | Runtime Errors | Functional Errors | Non-terminating Execution |
|---|---|---|---|---|---|---|
| Pan et al. [35] | Java, Python, C | C++ | 74.6% | 2.0% | 22.0% | 1.3% |
| | C++, Python, C | Java | 74.1% | 13.3% | 12.4% | 0.2% |
| | Java, C++, C | Python | 58.3% | 24.9% | 16.4% | 0.4% |
| | C++, Java, Python | C | 83.3% | 0.7% | 15.5% | 0.6% |
| **RustRepoTrans** | **C++, Java, Python** | **Rust** | **94.8%** | **0%** | **5.2%** | **0%** |

successfully but fails test cases), and non-terminating execution (code runs indefinitely or waits for input).

**Manual Error Causes Analysis.** We further conducted a manual analysis of the error causes for the unsuccessful translations with compilation errors (1,056 code samples), which accounted for the vast majority, 94.8%, of the failures as shown in Figure 4 (see Section 4.5.2). Using an open coding approach [18], we examined the error messages for each failing result and classified each compilation error into specific categories. If an error could not fit into an existing category, we created a new one or adjusted the definition of an existing category to include it, followed by revising and reannotating all relevant samples.

Notably, the same type of error could belong to different categories depending on the context. This iterative annotation process involved collaborative discussions to refine the categories and ensure consistent classification. We regularly reviewed and reorganized the categories to maintain clarity in our classification system.

*4.5.2 Results.* **Error Outcome Distribution.** Table 4 shows the error outcome distribution from the RQ1 experiment on RustRepoTrans, alongside data from Pan et al. [35] for translations to other programming languages (C, C++, Python, Java).

In RustRepoTrans, compilation errors account for 94.8% of failures, significantly higher than the 58.3% to 83.3% observed in other benchmarks. This emphasizes the challenge LLMs face when translating to Rust, a language known for stringent compile-time checks due to its ownership model and borrow checker. Unlike other benchmarks, RustRepoTrans recorded no runtime or non-terminating errors, indicating that Rust translations primarily fail at compile-time, underscoring its strong emphasis on memory safety.

The number of compilation errors in unsuccessful translations ranged from 1 to 193, with an average of 9.1 and a median of 4, which complicates debugging. In contrast to languages like Python, where the compiler stops at the first error, the Rust compiler reports all errors, making issue resolution more challenging. We categorized the types of compilation errors based on error codes from [18]. Table 5 presents the top 10 compilation errors encountered, with the five most common errors occurring over 1,000 times each. Many of these stem from using non-existent, unimplemented, unresolved, or undeclared elements, highlighting the hallucination phenomenon also seen in other repository-level code generation tasks [26, 43].

**Table 5: Top 10 Compilation Errors Reported by the Rust Compiler on RustRepoTrans**

| Error Code | Frequency | Description |
|---|---|---|
| E0425 | 5,949 | An unresolved name was used. |
| E0599 | 4,601 | This error occurs when a method is used on a type which doesn't implement it. |
| E0432 | 3,504 | An import was unresolved. |
| E0277 | 2,540 | You tried to use a type which doesn't implement some trait in a place which expected that trait. |
| E0433 | 1,345 | An undeclared crate, module, or type was used. |
| E0609 | 796 | Attempted to access a nonexistent field in a struct. |
| E0308 | 744 | Expected type did not match the received type. |
| E0061 | 222 | An invalid number of arguments was passed when calling a function. |
| E0412 | 174 | A used type name is not in scope. |
| E0282 | 79 | The compiler could not infer a type and asked for a type annotation. |

**Error Cause Taxonomy.** Fig. 9 illustrates the ten error cause categories we identified for failed code translations on RustRepoTrans, specifically those with compilation errors. We further classified these into three main types:

- **Failing to Understand Target Language Features.** This type includes errors stemming from the LLM's insufficient grasp of the syntax and semantics of the target programming language. It encompasses limitations in understanding data types, variable states, and contextual information.
- **Failing to Understand Differences Between Languages.** This type relates to the LLM's inadequate comprehension of the distinctions between the source and target programming languages. It covers issues such as syntax variations, differences in functions, variables, data types, and import paths.
- **Others.** This type includes errors unrelated to the LLM's code translation capabilities. Examples include missing punctuation marks and translations that do not adhere to the provided function signature.

The details of the 10 error causes from the types are introduced as following.

**Data Type Misinterpretation.** This error cause reflects the LLM's limitations in type inference, particularly evident in Rust's strict type system. Unlike Python, where variables can change types freely, or C, which allows implicit conversions, Rust requires exact type matches, making translation errors more prominent. LLMs may attempt incompatible assignments, perform arithmetic on mismatched types, or incorrectly treat non-collection variables (like integers) as if they were indexable. In the example below, the model interprets `imap_connected_here` as a boolean, causing a type mismatch that Rust's compiler disallows. This illustrates how Rust's strict typing exposes LLMs' type inference limitations, often masked in more permissive languages like Python.

```
let mut imap_connected_here = 0;
...
if imap_connected_here {        # error: expected bool, found integer
```

**Variable State Misinterpretation.** This error cause stems from the LLM's misunderstanding of variable states, leading to several misinterpretations. Common issues include treating undeclared variables as declared, making simultaneous mutable borrows, using the same variable as both mutable and immutable, and incorrectly treating private variables as public. In the example below, the model attempts to pass h as both an immutable reference (`&h`) and a mutable reference (`&mut h`), which Rust's strict borrowing rules prohibit, resulting in a compilation error. These instances highlight the LLM's limitations in tracking variable states and usage contexts, often leading to significant translation errors.

```
fn hashit(sha: usize, n: usize, id: &[u8], w: &mut [u8]) -> bool   # hashit function definition
...
hashit(sha, date, &h, &mut h);  # error: using h as both mutable and immutable parameter
```

**Context Misinterpretation.** This error cause arises from the LLM's misunderstanding of context. It includes issues such as failing

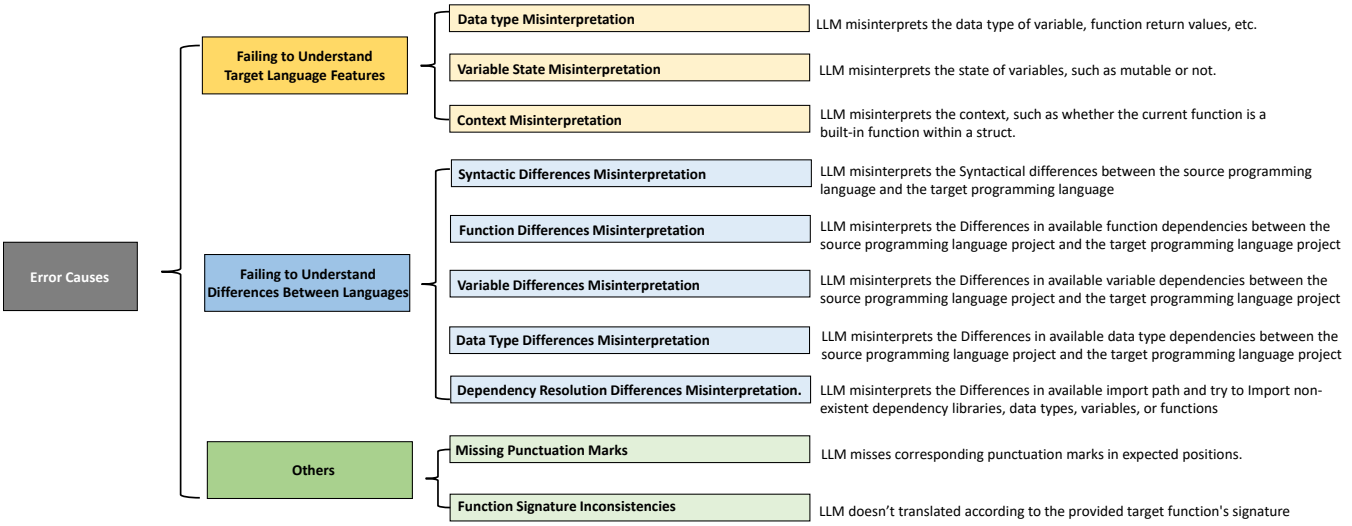


**Figure 9: Error Cause Taxonomy for Failed Code Translations on RustRepoTrans in LLMs**

to recognize when code is within a `struct`'s built-in function, using asynchronous operations in a synchronous block, or attempting modifications inside a function without the `mutable` modifier. In the example below, the model incorrectly assumes that batch is a built-in function of a `struct`. The error occurs when it tries to access `self.left(batch.clone())` without using the keyword `self`, which is necessary in this context.

```
fn batch(batch: &RecordBatch) -> Result<RecordBatch> { # not a struct's built-in function
    ...
    let left = self.left(batch.clone())?; # error: use the key word self
    ...
}
```

**Syntactic Differences Misinterpretation.** This error cause relates to the LLM's understanding of syntactic differences between programming languages. It occurs when the model incorrectly assumes that syntax from the source language is valid in the target language. Examples include using chained assignments (allowed in Python, C, and Java) in Rust, attempting to use the `goto` keyword from C, or employing an unsupported ternary operator "`condition ? expr1 : expr2`" in Rust.

**Function Differences Misinterpretation.** This error cause pertains to the LLM's understanding of functional dependency differences between programming languages. It arises when the model incorrectly assumes that callable functions in the target language are identical to those in the source language. Common issues include assuming a function present in the source language also exists in the target language, that it shares the same scope, or that the invocation method is identical. In the example below, the first line results in an error because the LLM fails to recognize that get should be called within the `stock_str` namespace.

```
ret.text = get(context, StockMessage::NoMessages)?;        # error
ret.text = stock_str::get(context, StockMessage::NoMessages)?;        # correct
```

**Variable Differences Misinterpretation.** This error cause arises from the LLM's understanding of variable dependency differences between programming languages. It involves errors where the model incorrectly assumes that variables in the target language mirror those in the source language. Common issues include presuming that parameters or declared variables from the source language exist in the target language, that their scopes are the same, or that specific member variables exist within a `struct`.

**Data Type Differences Misinterpretation.** This error cause arises from the LLM's understanding of differences in available data types across programming languages. It reflects errors where the model mistakenly assumes that data types in the source language are also present in the target language. For instance, it may assume that a user-defined data type retains the same name and meaning in both languages.

**Dependency Resolution Differences Misinterpretation.** This error cause relates to the LLM's understanding of differences in dependency resolution across programming languages. It highlights errors where the model incorrectly assumes that a specific import path is valid in the target language, such as presuming the existence of a particular dependency library or that certain functions, data types, or elements are present within the imported library.

**Missing Punctuation Marks.** This error cause arises from the LLM's failure to include necessary punctuation in its generated output. Examples include incomplete parentheses, missing commas between parameters, and other critical punctuation omissions.

```
if dc_param_exists(msg->param, DC_PARAM_SET_LATITUDE)) {
# error: Redundant closing parenthesis
```

**Function Signature Inconsistencies.** This error cause arises from the LLM's failure to follow instructions to translate according to the provided function signature. Examples include inconsistencies in function modifiers, mismatches in parameter lists, or discrepancies in return values.

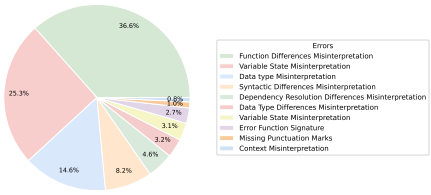


**Figure 10: Overall Distribution of Error Causes**

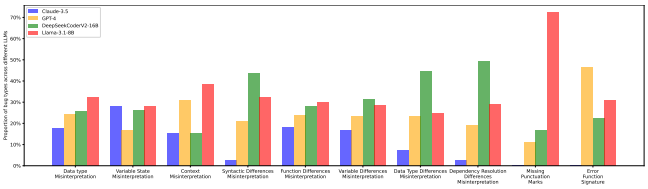


**Figure 11: Comparative Distribution of Error Causes in LLMs**

**Overall Distribution of Error Causes.** As shown in Fig. 10, the most common error cause encountered by LLMs during the code translation task on RustRepoTrans is *Failing to Understand Differences Between Languages*, which accounts for 77.9% of the errors. This is followed by *Failing to Understand Target Language Features* at 18.5%, and *Others* at 3.7%. These results indicate that the most significant challenge for LLMs in code translation is effectively grasping the various differences between the source and target languages, such as dependency and syntax differences.

At a more detailed level, the top three error causes are *Function Differences Misinterpretation* (36.6%), *Variable Differences Misinterpretation* (25.3%), and *Data Type Misinterpretation* (14.6%). This distribution highlights that function and variable dependencies constitute the largest portion of the dependencies involved in RustRepoTrans code translation tasks. Furthermore, since RustRepoTrans targets a strongly typed language, errors related to data types are ranked third.

**Comparison of Error Causes Distribution across Different LLMs.** Fig. 11 shows the distribution of error causes across various LLMs during code translation on RustRepoTrans. The *Failing to Understand Target Language Features* error causes are relatively evenly distributed among LLMs due to their similar training data, primarily derived from publicly available code. This results in comparable understanding of the target language across models. In contrast, *Failing to Understand Differences Between Languages* and *Other* error causes are more prevalent in LLMs with weaker translation capabilities. The former assesses the model's grasp of programming language differences, while the latter reflects its ability to follow instructions, both of which are positively correlated with translation performance.

Notably, Claude-3.5, the top-performing model on RustRepoTrans, exhibits the lowest proportions of *Failing to Understand Differences Between Languages* and *Other* error causes. For simpler error types like *Syntactic Differences Misinterpretation*, *Dependency Resolution Differences Misinterpretation*, *Missing Punctuation Marks*, and *Function Signature Inconsistencies*, the proportions were as low as 2.8%, 2.5%, 0%, and 0%, respectively, indicating superior understanding and adherence to instructions compared to other models.

*4.5.3 Conclusion.* Existing LLMs struggle with code translation tasks involving dependencies, particularly in distinguishing function and variable differences between source and target languages. When the target language is strongly typed, LLMs often lack understanding of data types. Their ability to recognize language differences is positively correlated with translation performance.

## 4.6 RQ5: Key Capabilities

To understand LLMs' code translation Capabilities beyond overall success rates (Pass@1 in RQ1), we analyze three aspects: the Capability of noise robustness, syntactical difference identification, and code simplicity.

*4.6.1 Design.* The key abilities for these three aspects were tested by evaluating each model on a dataset extracted or constructed from RustRepoTrans, tailored to the specific capability being assessed. The same translation prompts used in 4.2 were applied. The experimental design for each aspect is described as follows.

**Noise Robustness.** This capability evaluates the model's capacity to identify necessary dependencies from provided options, which is crucial in real-world scenarios where dependency information is often uncertain or incomplete. We assessed this capability through two angles: redundancy and incompleteness. For redundancy, we created a dataset by selecting functions and data types with high text similarity (using BLEU scores [36]) relative to the target function's dependencies, excluding those in the original set. We introduce a novel metric, *Redundancy Impact Rate (RIR)*, to measure the ratio of successful translations between scenarios with Redundant Dependencies and All Dependencies. For incompleteness, we created various datasets by randomly reducing the target function's dependencies to 75%, 50%, 25%, and 0% of their original size. Using these new benchmarks, we assessed the success rates (Pass@1) of the LLMs and compared them to the original Pass@1 with all dependencies (see RQ1). We introduce a novel metric, *Incompleteness Impact Rate (IIR)*, which measures the LLM's performance by calculating the average of the Pass@1 under 100%(i.e. All), 75%, 50%, 25%, and 0% dependencies.

**Syntactical Differences Identification.** This capability assesses the model's skill in recognizing syntactical differences between source and target programming languages, a key requirement for accurate code translation. For instance, Rust, as a strongly-typed language, does not require checks such as memory exception handling (common in C), type checks (in Python), or null pointer checks (in C and Java) shown in Fig. 12. We selected 89 function pairs from RustRepoTrans that include these checks in the source language and evaluated whether the LLM correctly omits them in the target language. We introduce a novel metric, Syntactical Differences Identification Rate (*SDIR@K*), which measures the LLM's success in identifying and omitting these checks in the translated code across $K$ samples.

**Code Simplicity.** This capability assesses the simplicity of translation results, as simpler code is preferred by developers. We evaluate this by calculating token counts with tree-sitter [41] and measuring cyclomatic complexity [19], where higher values indicate greater complexity. We apply rust-code-analysis [10] to both the reference and translated code that passed the test cases. A higher ratio suggests that the translated code is more concise and closely

**Memory exception handling:**

```
if (context->magic!=DC_CONTEXT_MAGIC){
        .....
}
// Any correct operation on the struct will not modify
context->magic. Therefore, if context->magic is not
equal to the predefined value DC_CONTEXT_MAGIC,
it indicates that a memory exception has occurred.
```

**Type checks:**

```
if not isinstance(other, CharsetMatch):
        raise ValueError
```

**Null pointer checks:**

```
Preconditions.checkNotNull(props, ...);
```

**Figure 12: Example of checks related to language syntactic features to evaluate LLM's capability to identify syntactical differences**

matches the reference. We introduce two metrics for code simplicity: *Token Rate* and *CC Rate*. These compare the token counts and cyclomatic complexities of the reference and translated code. Higher values for both metrics indicate that the generated code is simpler and more aligned with the original.

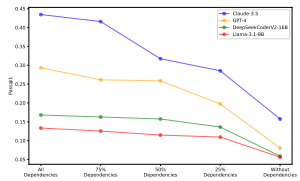


**Figure 13: The *Pass@1* of LLMs under different proportions of dependencies**

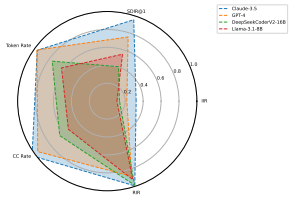


**Figure 14: The performance of LLMs on Key Abilities of code translation.**

*4.6.2 Results.* **Noise Robustness.** From the perspective of redundant dependencies, after introducing interference dependencies, Claude-3.5 and DeepSeekCoderV2-16B demonstrated minimal fluctuations, with RIR values of 99.4% and 101.6%. These variations fall within the expected range for LLM generation. In contrast, GPT-4 and Llama-3.1-8B exhibited larger fluctuations, with RIR values of 88.2% and 92%, highlighting potential areas for improvement in their noise robustness regarding redundant dependencies. Regarding incomplete dependencies, as shown in Fig. 13, DeepSeekCoderV2-16B and Llama-3.1-8B showed stable performance during gradual reductions, with drop ratios of 19.1% and 18.0% at the 25% level. Noticeable declines occurred only at the Without Dependencies stage. In contrast, Claude-3.5 and GPT-4 faced significant drops at nearly every stage, likely because their successful translations often rely on multiple dependencies. Most successful translations for DeepSeekCoderV2-16B and Llama-3.1-8B came from simpler data with fewer dependencies, ensuring consistent availability across stages except at the Without Dependencies stage. As shown in Fig. 14, the IIR ranking among models are the same as the Pass@1 under All Dependencies in 4.2, Claude-3.5 demonstrated the best performance with IIR values of 32.2%.

**Syntactical Differences Identification.** As shown in Fig. 14, Claude-3.5 achieves the highest *ISDR@1* at 95.5%, demonstrating a strong ability to identify syntactical differences between programming languages. This finding is consistent with the results in Section 4.4 regarding the distribution of error causes across different LLMs (see Fig. 11). In contrast, DeepSeekCoderV2-16B has the lowest *ISDR@1* at only 40.5%. This indicates that DeepSeekCoderV2-16B often assumes the checks performed in the source language are also necessary in the target language, leading to errors by incorrectly treating those checks as valid in the translation.

**Claude-3.5 translation result: Without Loop**

```
if characters.chars().any(|character| {character == primary_range})) {
    . . . }
```

**Other LLMs translation result: With Loop**

```
for character in characters.chars() {
    if (character == primary_range){
        . . .
        break;}}
```

**Claude-3.5 translation result: Without Loop**

```
self.pool.fill(0);
```

**reference code: With Loop**

```
for i in 0..32 {
    self.pool[i] = 0;}
```

**Figure 15: Claude-3.5 translation result compared to Other LLMs and reference code**

**Code Simplicity.** Fig. 14 shows that among the successfully translated tasks, the Token Rate for code generated by Claude-3.5 and GPT-4 reached 96.6% and 97.0%, respectively, indicating high simplicity compared to the reference code. In contrast, Llama-3.1-8B had the lowest Token Rate at 63.1%, suggesting it requires more code to achieve the same functionality.

In terms of CC Rate, Claude-3.5 again excels with a CC Rate of 103.7%, while Llama-3.1-8B is notably lower at 53.6%. This difference arises because Claude-3.5 often uses concise built-in functions in place of loops. For instance, as illustrated in Fig. 15, when tasked with finding the first element in an array that meets a condition, Claude-3.5 employs the target language's any method from the iterator library, whereas other LLMs rely on traditional loops. Furthermore, Claude-3.5's CC Rate exceeding 1 indicates that, in terms of cyclomatic complexity, its generated code is simpler than the reference code. Fig. 15 provides an example where C serves as the source programming language. This simplification may be attributed to the Rust ground truth developers using the C2Rust migration tool [11], which translates code via one-to-one pattern matching.

These findings suggest that Claude-3.5 demonstrates a stronger code comprehension and implementation ability, accurately understanding the source code's functionality and translating it concisely based on the target language's features.

*4.6.3 Conclusion.* The complexity of code generated by LLMs in translation reflects their translation capabilities: models with stronger abilities produce code with lower complexity for equivalent functionality. Claude-3.5, which excels on RustRepoTrans, also shows superior performance in Noise Robustness, Syntactical Differences Identification, and Code Simplicity.

## 5 Threats to Validity

One potential threat is data leakage between our benchmark and model training data. However, the training data comprises independent function code from different languages rather than functionally equivalent code pairs, which is crucial for code translation. Another concern is the limited size and variety of programming languages in our benchmark, which may impact the generalizability of our findings. We plan to extend our benchmark in the future. Some projects also lack consistent documentation across language versions, leading to potential functional discrepancies. We mitigated this by manually verifying the functionality of the final function pairs. Additionally, the static code analysis tool tree-sitter may not accurately retrieve all dependencies for Rust functions, which we addressed through manual validation. Lastly, our study involved only four models due to time and resource constraints. The focus of this paper is the benchmark itself, not the models. We intend to incorporate more models in future work, but the conclusions and evaluation capabilities of our benchmark remain valid.

## 6 Conclusion

This work makes the first attempt to evaluate LLMs on repository-level code generation targeting Rust. We first manually construct the first repository-level code translation benchmark RustRepoTrans and perform the first study of 4 state-of-the-art LLMs on repository-level code translation. we find that existing LLMs show much worse performance on repository-level code translation compared to on standalone code translation, LLMs still have significant room for improvement in their code translation capabilities within real-world software development scenarios. Besides, when the target language is a low-resource language with multiple syntactic constraints, such as Rust, LLMs struggle to effectively identify the various differences between the source and target languages, as well as understand the features of the target language.

## 7 Data Availability

All data/code used in this study is provided in the replication package [7].

## References


[1] 2018. Upgrading GitHub from Rails 3.2 to 5.2. https://github.blog/engineering/upgrading-github-from-rails-3-2-to-5-2/.

[2] 2020. GitHub's Journey from Monolith to Microservices. https://www.infoq.com/articles/github-monolith-microservices/.

[3] 2020. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/.

[4] 2020. Transform monolithic Java applications into microservices with the power of AI. https://developer.ibm.com/tutorials/transform-monolithic-java-applications-into-microservices-with-the-power-of-ai/.

[5] 2020. Will code move on to a language such as Rust? https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/.

[6] 2024. Rust. https://doc.rust-lang.org/stable/book/print.html.

[7] 2024. RustRepoTrans. https://github.com/TrustedGPT/RustRepoTrans.

[8] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590* (2021).

[9] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

[10] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. 2020. rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes. *SoftwareX* 12 (2020), 100635. https://doi.org/10.1016/j.softx.2020.100635

[11] C2Rust. 2024. https://github.com/immunant/c2rust

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Siheng Chen, Rohan Varma, Aliaksei Sandryhaila, and Jelena Kovačević. 2015. Discrete signal processing on graphs: Sampling theory. *IEEE transactions on signal processing* 63, 24 (2015), 6510–6523.

[14] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[15] Claude. 2024. *Claude api interface.* https://docs.anthropic.com/en/api/getting-started

[16] DeepSeek. 2024. DeepSeek-Coder-V2-Lite-Instruct. https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct.

[17] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280* (2016).

[18] Rust error codes. 2024. https://doc.rust-lang.org/error_codes/error-index.html

[19] Geoffrey K Gill and Chris F Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering* 17, 12 (1991), 1284–1288.

[20] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1529–1541.

[21] Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6766–6805.

[22] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi Müller, and John Mylopoulos. 2010. Code migration through transformations: An experience report. In *CASCON First Decade High Impact Papers*. 201–213.

[23] Jierui Li, Szymon Tworkowski, Yingying Wu, and R. Mooney. 2023. Explaining Competitive-Level Programming Solutions using LLMs. *ArXiv* abs/2307.05337 (2023). https://doi.org/10.48550/arXiv.2307.05337

[24] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[26] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667* (2024).

[27] Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. 2024. Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 57–68.

[28] Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. 2024. Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 57–68.

[29] Bruno Góis Mateus, Matias Martinez, and Christophe Kolski. 2023. Learning migration models for supporting incremental language migrations of software applications. *Information and Software Technology* 153 (2023), 107082.

[30] Meta. 2024. Llama-3.1-8B-Instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct.

[31] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 651–654.

[32] OpenAI. 2023. GPT-4. https://openai.com/index/gpt-4-research/.

[33] OpenAI. 2024. *OpenAI api interface.* https://platform.openai.com/docs/api-reference/introduction

[34] Rangeet Pan, Ali Reza Ibrahimzada, R. Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, S. Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. *ArXiv* abs/2308.03109 (2023). https://doi.org/10.48550/arXiv.2308.03109

[35] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[36] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[37] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[38] Israt Jahan Rithy, Hasib Hossain Shakil, Niloy Mondal, Fatema Sultana, and Faisal Muhammad Shah. 2022. XTest: A Parallel Multilingual Corpus with Test Cases for Code Translation and Its Evaluation. In *2022 25th International Conference on Computer and Information Technology (ICCIT)*. IEEE, 623–628.

[39] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.

[40] Johannes Thönes. 2015. Microservices. *IEEE software* 32, 1 (2015), 116–116.

[41] tree sitter. 2023. https://github.com/tree-sitter/tree-sitter

[42] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and language models examined. In *Proceedings of the 19th Australasian Document Computing Symposium*. 58–65.

[43] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *arXiv preprint arXiv:2401.06391* (2024).

[44] Liang Xu, Anqi Li, Lei Zhu, Hang Xue, Changtai Zhu, Kangkang Zhao, Haonan He, Xuanwei Zhang, Qiyue Kang, and Zhenzhong Lan. 2023. Superclue: A comprehensive chinese large language model benchmark. *arXiv preprint arXiv:2307.15020* (2023).

[45] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv*

*preprint arXiv:2310.04951* (2023).

[46] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.

[47] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[48] Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720* (2018).

[49] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).

[50] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 195–204.

[51] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

[52] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 36. 11783–11790.