RustRepoTrans: Repository-level Context Code Translation Benchmark Targeting Rust

Guangsheng Ou[1], Mingwei Liu*[1], Yuxuan Chen[1], Yanlin Wang[1], Xin Peng[2], and Zibin Zheng[1]
[1]Sun Yat-sen University, Zhuhai, China.
[1]Email: ougsh3@mail2.sysu.edu.cn, liumw26@mail.sysu.edu.cn, chenyx677@mail2.sysu.edu.cn, {wangylin36,zhzibin}@mail.sysu.edu.cn
[2]Fudan University, Shanghai, China.
[2]Email: pengxin@fudan.edu.cn

***Abstract*—Recent advancements in large language models (LLMs) have demonstrated impressive capabilities in code translation, typically evaluated using benchmarks like CodeTransOcean and RepoTransBench. However, dependency-free benchmarks fail to capture real-world complexities by focusing primarily on simple function-level translations and overlooking repository-level context (e.g., dependencies). Full-repository translation benchmarks significantly exceed the current capabilities of existing models, resulting in performance bottlenecks that fail to provide actionable insights for guiding model development. Furthermore, existing benchmarks do not account for the scenario of incrementally translating new or modified modules from the source to the target language, which demands careful handling of repository-level contexts such as dependencies, cross-module references, and architectural divergence. Moreover, LLMs' effectiveness in translating to newer, low-resource languages like Rust remains largely underexplored.**

To address these gaps, we introduce RustRepoTrans, the first repository-level context code translation benchmark targeting incremental translation, comprising 375 tasks translating into Rust from C, Java, and Python. Using this benchmark, we evaluate seven representative LLMs, analyzing their errors to assess limitations in complex translation scenarios. Among them, DeepSeek-R1 performs best with 51.5% Pass@1, excelling in both basic functionality and additional translation abilities, such as noise robustness and syntactical difference identification. However, even DeepSeek-R1 experiences a 22.2% performance drop (*Pass@1* from 73.7% to 51.5%) when handling repository-level context compared to previous benchmarks without such context. Meanwhile, we propose a set of more fine-grained evaluation metrics and an enhanced evaluation framework, enabling a more comprehensive analysis of LLMs' performance in repository-level context code translation tasks to provide fine-grained insights that can effectively inform the development of code translation techniques.

I. Introduction

Code translation, or code migration, is the process of converting a software project from one programming language to another [1], often driven by the need to adapt to different runtime environments, improve performance, or enhance security [2]–[7]. The rise of languages like Rust [8] and Cangjie [9] has increased demand for translation. However, migrating legacy code remains challenging due to semantic mismatches, differences in standard libraries, and the need to maintain functional equivalence [10], [11].

In real-world projects, translation rarely rewrites an entire repository at once. Developers typically **incrementally translate new or modified modules from the source to the target language** [12]–[15]. Newly translated code must integrate with existing modules, including components migrated earlier or manually refined, to preserve functional correctness and consistency. Incremental workflows also arise when the source project evolves after partial migration, requiring new features to be ported into the target repository. **Such repository-level migration demands careful handling of repository-level context such as dependencies, cross-module references, and architectural divergence.** It further supports human-in-the-loop workflows, where developers refine each step under non one-to-one code mapping, reflecting the realities of cross-language migration.

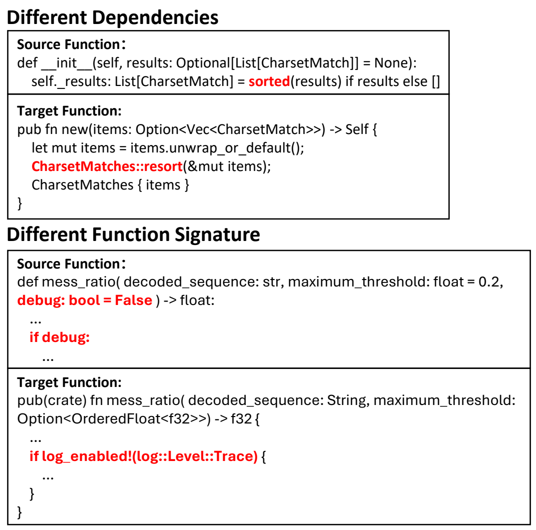
While large language models (LLMs) have shown promise in translating widely used languages like Java and Python [1], [16], but their effectiveness in translating to newer, lower-resource languages like Rust has not been thoroughly explored. Rust is increasingly chosen for migration due to its memory safety and reliability benefits [17], yet its strict ownership model and limited training data pose significant challenges for automated translation [8].

Existing benchmarks can be categorized into dependency-free and full repository, however, both categories of datasets only partially capture these requirements. Most focus on *function-level* translation [18]–[21], where self-contained snippets have minimal dependencies, and even *file-level* benchmarks [22]–[24] provide limited repository context. Many benchmarks are also sourced from online platforms or synthetic examples, diverging from real-world development practices [18]–[24]. In contrast, practical Rust migration demands repository-level reasoning, dependency management, and incremental integration—factors largely absent from dependency-free benchmarks.

While repository-level translation benchmarks have emerged [25], [26] (e.g., RepoTransBench), fully translating entire repositories remains highly challenging for current LLMs. In RepoTransBench [25], for example, the best model, Claude-3.5, achieves only 7.33% Success@1, highlighting a large performance gap. Moreover, full-repository translation complicates fine-grained evaluation, as errors may result from the model or from differences in target-language libraries, making it difficult to pinpoint their source. Furthermore, existing repository-level benchmarks generally assume

* M. Liu is the corresponding author.
[1] G. Ou, M. Liu, Y. Chen, Y. Wang, Z. Zheng are with the School of Software Engineering and the Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China.
[2] X. Peng is with the College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00057



Different Dependencies

```
Source Function:
def __init__(self, results: Optional[List[CharsetMatch]] = None):
    self._results: List[CharsetMatch] = sorted(results) if results else []
```

```
Target Function:
pub fn new(items: Option<Vec<CharsetMatch>>) -> Self {
    let mut items = items.unwrap_or_default();
    CharsetMatches::resort(&mut items);
    CharsetMatches { items }
}
```

Different Function Signature

```
Source Function:
def mess_ratio( decoded_sequence: str, maximum_threshold: float = 0.2,
debug: bool = False ) -> float:
    ...
    if debug:
      ...
```

```
Target Function:
pub(crate) fn mess_ratio( decoded_sequence: String, maximum_threshold:
Option<OrderedFloat<f32>>) -> f32 {
    ...
    if log_enabled!(log::Level::Trace) {
      ...
    }
}
```

Fig. 1. Motivation Examples from RustRepoTrans of Different Architecture Between Source Language (Python) And Target Language (Rust) Version

ideal conditions, where function dependencies and project architecture are perfectly aligned between source and target languages. In practice, however, differences in language features, optimization goals, and coding idioms often lead to divergent architectures. Consequently, one-to-one mapping between source and target codebases is rare, reflecting realistic context shifts. As illustrated in Fig. 1, the same function may have different dependencies (e.g., built-in vs. custom utilities) or distinct signatures (e.g., parameter-driven vs. global configuration) across languages. **These limitations show that current repository-level benchmarks reveal performance gaps but fail to fully assess LLMs' ability to handle repository-level context, dependency interactions, and incremental translation.** Therefore, an intermediate benchmark is needed—one that incorporates repository-level context while remaining tractable for current models. Such a benchmark can bridge the gap between dependency-free function-level translation and full-repository translation, enabling fine-grained evaluation and better guiding the development of robust code translation techniques.

To bridge this gap, we introduce **RustRepoTrans, the first benchmark to evaluate code translation with *repository-level context*, specifically targeting Rust**. Unlike function-level benchmarks that assess translation in isolation, RustRepoTrans captures the complexities of real-world migration by incorporating repository-level context, including *dependencies, cross-file interactions, and architectural constraints*. These elements are crucial for ensuring that translated code integrates seamlessly into existing projects rather than functioning isolatedly. Compared to full repository benchmarks, RustRepoTrans focuses on incremental translation scenarios which demands the careful handling of non one-to-one code mapping such as different dependencies or distinct function signatures. Overall, RustRepoTrans comprises 375 curated translation tasks derived from real-world projects spanning C, Java, and Python to Rust, providing a more faithful assessment of LLMs' translation capabilities. While we focus on Rust due to its growing adoption and inherent challenges of translating to it, our methodology is generalizable to other programming languages, paving the way for broader repository-level evaluation.

Building on RustRepoTrans, our experiments of seven representative LLMs show that LLMs struggle with repository-level context code translation, with compilation errors reaching 92.3%, exposing the gap between current evaluations and real-world performance. DeepSeek-R1 performs best with 51.5% Pass@1, excelling in both basic functionality and additional translation abilities like noise robustness and syntactical difference identification. However, even DeepSeek-R1 experiences a 22.2% performance drop (*Pass@1* from 73.7% to 51.5%) when handling repository-level context. Dependency-related errors, including function and variable resolution issues, account for 67.6% of failures, highlighting the challenge of interconnected code. Meanwhile, we propose a set of more fine-grained evaluation metrics such as evaluating noise robustness, syntactical difference handling, and code simplicity during translation and an enhanced evaluation framework, enabling a more comprehensive analysis of LLM performance in repository-level context code translation tasks (in RQ4).

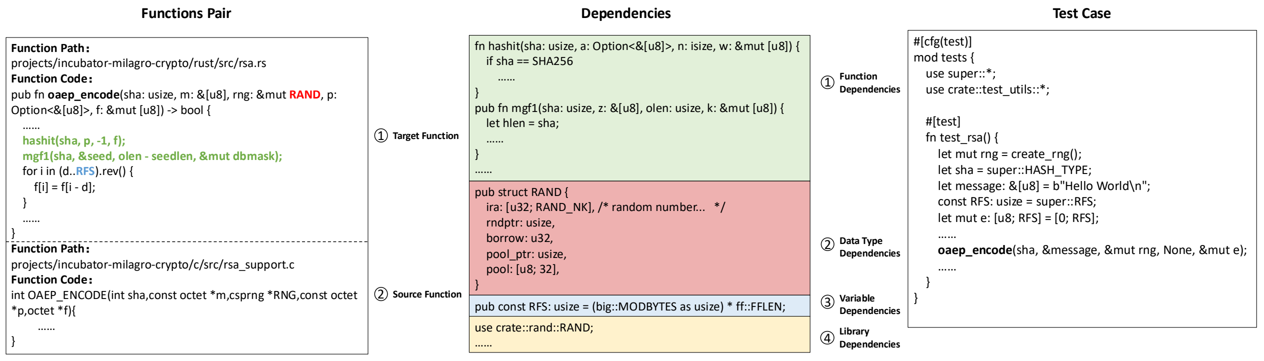
The contributions of this paper are as follows. Data and code are publicly available at [27] and [28].

- The first repository-level context code benchmark, RustRepoTrans, targeting incremental translation with 375 tasks from C, Java, and Python to Rust, is introduced for realistic evaluation;
- Seven representative LLMs are evaluated, assessing their performance in real-world incremental translation;
- LLMs' errors are categorized into 10 types, revealing limitations, especially in handling dependencies;
- A set of more fine-grained evaluation metrics of code translation and an enhanced evaluation framework.

II. Related Work

A. Code Translation

Code translation promotes software interoperability and legacy system modernization, enhancing productivity and reducing manual effort [19], [22]. Early approaches employed rule-based and machine learning [29]–[33] to capture frequent code patterns. Although LLMs have advanced the field, challenges remain in handling translation errors and adapting to complex scenarios [24], [34]. Jiao et al. [21] underscored LLMs' difficulty with intricate cases, while Pan et al. [16] demonstrated that context-rich prompts improve reliability. Prompt engineering strategies by Yang et al. [35] and Macedo et al. [36] further enhanced translation consistency and accuracy. LLM have also been applied to the translation of entire repository [37]–[39]. Nitin et al. [37] leverage C2Rust to convert C into unsafe Rust and then apply an LLM for safer, idiomatic translation. Zhang et al. [38] translate Go to Rust via predefined rules, code partitioning, and localized checks. However, the evaluation data used in these works are not composed of ground-truth code pairs, only the source language version is genuinely available.



Function Path:
projects/incubator-milagro-crypto/rust/src/rsa.rs
Function Code:

```
pub fn oaep_encode(sha: usize, m: &[u8], rng: &mut RAND, p:
Option<&[u8]>, f: &mut [u8]) -> bool {
   ......
   hashit(sha, p, -1, f);
   mgf1(sha, &seed, olen - seedlen, &mut dbmask);
   for i in (d..RFS).rev() {
      f[i] = f[i - d];
   }
   ......
}
```

Function Path:
projects/incubator-milagro-crypto/c/src/rsa_support.c
Function Code:

```
int OAEP_ENCODE(int sha,const octet *m,csprng *RNG,const octet
*p,octet *f){
      ......
}
```

```
fn hashit(sha: usize, a: Option<&[u8]>, n: isize, w: &mut [u8]) {
   if sha == SHA256
      ......
}
pub fn mgf1(sha: usize, z: &[u8], olen: usize, k: &mut [u8]) {
   let hlen = sha;
   ......
}
......
```

```
pub struct RAND {
   ira: [u32; RAND_NK], /* random number...   */
   rndptr: usize,
   borrow: u32,
   pool_ptr: usize,
   pool: [u8; 32],
}
```

```
pub const RFS: usize = (big::MODBYTES as usize) * ff::FFLEN;
```

```
use crate::rand::RAND;
......
```

```
#[cfg(test)]
mod tests {
   use super::*;
   use crate::test_utils::*;

   #[test]
   fn test_rsa() {
      let mut rng = create_rng();
      let sha = super::HASH_TYPE;
      let message: &[u8] = b"Hello World\n";
      const RFS: usize = super::RFS;
      let mut e: [u8; RFS] = [0; RFS];
      ......
      oaep_encode(sha, &message, &mut rng, None, &mut e);
      ......
   }
}
```

Fig. 2. RustRepoTrans Format Example

We propose RustRepoTrans, a benchmark that incorporates repository-level context and dependencies, enabling more realistic evaluation of LLMs and their capacity to handle complex code translation tasks.

B. Code Translation Benchmarks

Existing code translation datasets are generally classified into dependency-free, function-level [18]–[21] (or single-file [22]–[24]) datasets and full repository-level datasets [25], [26]. The former are often mined from Q&A platforms—e.g., CodeTransOcean from Rosetta Code, XCodeEval from GeeksForGeeks—but fail to reflect real-world software development, lacking the architectural and dependency complexities critical for faithful translation across files. In contrast, RepoTransBench [25] introduces a repository-level benchmark with executable tests, while TRANSREPO-BENCH [26] proposes a Skeleton-Guided-Translation framework for Java-to-C# translation, leveraging coarse-grained architectural guidance for holistic evaluation. Nevertheless, full repository translation remains highly challenging; for example, the best model on RepoTransBench achieves only 7.33% Success@1 [25], underscoring current LLM limitations. Additionally, full-repo evaluation hinders fine-grained assessment due to cross-language inconsistencies such as library discrepancies.

RustRepoTrans, the first repo-level context benchmark, bridges the gap between function-level and full-repo translation by offering a repository-level context benchmark with manageable complexity.

III. RustRepoTrans Benchmark

We introduce the benchmark format, construction method, and the resulting RustRepoTrans.

A. Benchmark Format

Each task in RustRepoTrans consists of a pair of functions with their dependencies and test cases, formatted as <source function, target function, target function dependencies, target function test cases> (Fig. 2). The function pairs represent functionally equivalent code snippets from source and target languages, with dependencies relevant to target function such as functions, data types, variables, and libraries. LLMs use the source function, target function signature, and associated dependencies to generate the target function, which is verified for correctness using the test cases.

B. Benchmark Construction Method

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

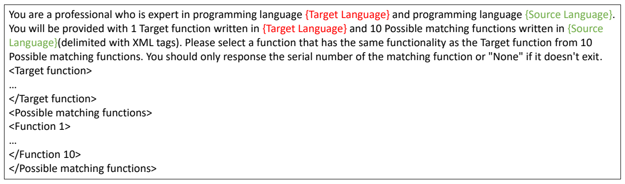
Functionally Equivalent Code Pairs Extraction In this part, we focus on extracting functionally equivalent source–target function pairs. The pipeline consists of five stages:

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. 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 source project, we extract all implemented functions. 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 the static code analysis tool tree-sitter [40]. 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 [41]. 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



```
You are a professional who is expert in programming language {Target Language} and programming language {Source Language}.
You will be provided with 1 Target function written in {Target Language} and 10 Possible matching functions written in {Source
Language}(delimited with XML tags). Please select a function that has the same functionality as the Target function from 10
Possible matching functions. You should only response the serial number of the matching function or "None" if it doesn't exit.
<Target function>
...
</Target function>
<Possible matching functions>
<Function 1>
...
</Function 10>
</Possible matching functions>
```

Fig. 3. Prompt for Identifying Equivalent Function Pairs

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 [43]. The prompt used for this identification is shown in Fig. 3. 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, two of the authors with 4-6 years of coding experience are involved as participants to manually double check the equivalent of verified function pair by LLMs to ensure actual functional equivalence. Since functionally equivalent project pairs in different programming languages, sharing the same version, have already been identified during the Migration Project Selection phase using project documentation, participants only need to validate whether substantial project architectural refactoring has occurred, which results in the absence of functionally equivalent function pairs or in cases where functions with identical names implement different functionalities across the two projects.

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.

Dependencies are classified into three categories: function dependencies, 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.

C. *Resulting Benchmark*

This process constructs RustRepoTrans with 375 repository-level translation tasks. Table I compares it with existing benchmarks. Similar in size to prior datasets, each task includes a manually verified ground truth translation and unit tests, achieving over 90% test coverage, ensuring high quality.

RustRepoTrans has two key features that distinguish it from previous benchmarks: 1) it is the first repository-level context code translation benchmark 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 previous benchmarks. Unlike artificially constructed data or data from Q&A websites, real project data exhibits more complex dependency relationships, including function, data type, and variable dependencies, which are absent in existing datasets as shwon in Table I, 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 I. 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.

IV. Evaluation

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

- **RQ1 (LLMs Performance)**: How do the studied 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 (Failure Analysis)**: What types of errors do LLMs encounter on RustRepoTrans, and what factors contribute to these translation failures?
- **RQ4 (Key Capabilities)**: Beyond translation accuracy, what essential capabilities do LLMs demonstrate on RustRepoTrans, including noise robustness, syntactical difference identification, and code simplicity?

A. *Experimental Setup*

Model Selection. As shown in Table II, we selected seven presentative LLMs that have been widely studied in recent research including general LLM and code LLM, open-source and closed-source models, as well as reasoning models and non-reasoning models to conduct a comprehensive evaluation of LLM's code translation capabilities in real-world scenarios.

Implementation Details. Due to resource constraints, for open-source models with sizes below 20B parameters, we obtained and executed the released versions from their official repositories with greedy sampling [44] as our generation strategy. All evaluations were conducted on an NVIDIA A800 80GB GPU. For closed-source LLMs and other open-source models, we accessed each model through their official API interface [45]–[49]. To achieve results similar to greedy decoding, we set the "temperature" hyperparameter to 0.

B. *RQ1: LLMs Performance*

We evaluated studied LLMs on RustRepoTrans to assess their translation accuracy and self-debugging abilities for repository-level context code translation tasks targeting Rust.

TABLE I
COMPARISON OF DIFFERENT BENCHMARKS FOR CODE TRANSLATION

Dataset	**Source**	**Task Level**	**#Tokens**	# Tasks	**Source Languages**	**Average number of Dependencies**	**Target Languages Include Rust?**	**Unit Tests Included ?**	**Golden Answer Verified?**
CodeXGLUE [19]	Lucune, POI, JGit, Antlr	Function Level	42.3	1,000	Java, C#	0	✗	✗	✗
XLCOST [24]	G4G	Program Level	202	901	C++, Java, C#, PHP, JavaScript, Python, C	0	✗	✗	✗
TransCoder-test [18]	G4G	Function Level	107.0	948	C++, Java, Python	0	✗	Partial	✗
HumanEval-X [20]	HumanEval	Program Level	97.7	164	C++, Java, Go, JavaScript, Python	0	✗	✓	✓
G-TransEval [21]	HumanEval, G4G, .NET samples	Function Level	95.1	400	C++, Java, C#, JavaScript, Python	0	✗	✓	✓
CodeTransOcean [23]	Rosetta Code	Program Level	448.4	2,878*	Java, C++, C#, PHP, Python, Go	0	✓	✓	✓
RustRepoTrans	Github	**Function Level**	**150.1**	375	C, Java, Python	**5.4**	✓	✓	✓

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

TABLE II
STUDIED LLMS

Model Type	**Model Name**	**Open-source**	**reasoning**	**Time**	**Size**
General LLM	DeepSeek-R1-0528 [47]	✓	✓	2025.5	671B
	DeepSeek-V3-0324 [48]	✓	✗	2025.3	671B
	Claude-3.5-Sonnet [52]	✗	✗	2024.6	-
	GPT-4 [53]	✗	✗	2023.6	-
	Llama-3.1-8B [54]	✓	✗	2024.7	8B
Code LLM	Qwen-2.5-coder-32B [55]	✓	✗	2024.9	32B
	DeepSeekCoderV2-16B [56]	✓	✗	2024.6	16B

1) Design: The evaluation involved testing each model on RustRepoTrans using controlled prompts, with output correctness assessed through specific test cases.

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. 4. These best practices have been shown to enhance translation accuracy in similar studies [16], [35], [50], [51]. 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 [16], [35], further experiments were conducted to understand the extent to which errors in the generated code samples can be corrected by LLMs with feedback. The debugging prompt (Prompt-Fix) in Fig. 4 includes instruction for previous translation, instruction for debugging, incorrect translation and error details and translation's required information, adapted from previous work [16], [35].

Metrics. In line with previous work [16], [23], two key metrics: Pass@1 and DSR@1 were utilized. Pass@1 [57] reflects the LLMs' ability to produce correct translation on the first attempt, while DSR@1 [23] reflects the model's ability to produce correct translation allowing for one debugging attempt.

2) Results: Fig. 5 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 self-debugging improvement potential.

Initial Translation Performance (*Pass@1*). In initial code translation task, DeepSeek-R1 substantially outperformed other LLMs, achieving 51.5% accuracy. Among non-reasoning LLMs, DeepSeek-V3 achieving highest accuracy, only 1.4% lower than DeepSeek-R1 and 6.6% higher than Claude-3.5, 15.7% higher than Qwen-2.5-coder-32B, 20.8% higher than GPT-4, 33.3% higher than DeepSeekCoderV2-16B, and 36.8% higher than Llama-3.1-8B on the *Pass@1* metric. However, even DeepSeek-R1 achieved only a 51.5% *Pass@1* score, emphasizing the difficulty of RustRepoTrans and the challenges inherent in translating code with complex dependencies.

Self-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 [23], [35]. The self-debugging performance rankings revealed DeepSeek-R1 leading at 62.1%, followed by DeepSeek-V3 at 58.7%. Despite the strong performance of DeepSeek-R1, 37.9% of its cases remained unresolved, highlighting the inherent challenges of self-correction in intricate translation tasks. In terms of relative improvement, Claude-3.5 exhibited the highest percentage gain at 29.9%, indicating its strong capabilities of self-debugging in repository-level context code translation. Overall, even after self-debugging, DeepSeek-R1 maintained its leading position, emphasizing its robust capabilities in **incremental code translation scenario** to Rust.

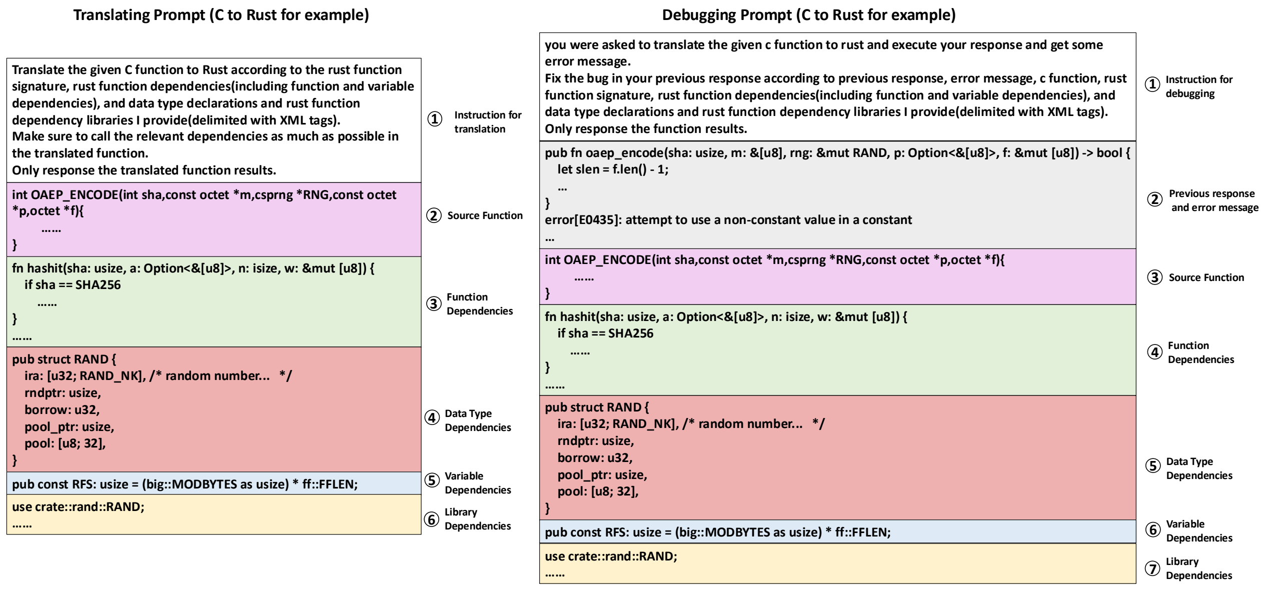
3) Summary: Results reveal that RustRepoTrans effectively challenges LLMs in complex code translation, as even the best model, DeepSeek-R1, only achieves a *Pass@1* of 51.5%, and *DSR@1* of 62.1%, highlighting the limitations of LLMs.

C. RQ2: RustRepoTrans Effectiveness

To assess our benchmark's effectiveness, a comparative analysis was conducted based on prior literature.

1) Design: We evaluated performance on RustRepoTrans and a prior Rust translation benchmark without repository-level context, demonstrating RustRepoTrans's effectiveness in assessing models for complex, real-world translation scenarios. Our analysis is based on CodeTransOcean [23], which includes translation tasks across 45 programming languages, including Rust, without repository-level dependencies. To ensure a fair and efficient comparison, we constructed a subset of CodeTransOcean by selecting translation pairs where Rust is the target language and C, Java, or Python is the source language, removing those with non-compilable Rust code. From the remaining pairs, we randomly sampled 300 (100 per language) to match our dataset's scale for comparative experiments.

We comprehensively evaluated top-performing models in RustRepoTrans (DeepSeek-R1, DeepSeek-V3, Claude-3.5, Qwen-2.5-coder-32B) by comparing their average *Pass@1* and *DSR@1* on a CodeTransOcean subset with RQ1 results. The


Translating Prompt (C to Rust for example)
① Instruction for translation
② Source Function
③ Function Dependencies
④ Data Type Dependencies
⑤ Variable Dependencies
⑥ Library Dependencies
Debugging Prompt (C to Rust for example)
① Instruction for debugging
② Previous response and error message
③ Source Function
④ Function Dependencies
⑤ Data Type Dependencies
⑥ Variable Dependencies
⑦ Library Dependencies

Translate the given C function to Rust according to the rust function signature, rust function dependencies(including function and variable dependencies), and data type declarations and rust function dependency libraries I provide(delimited with XML tags).
Make sure to call the relevant dependencies as much as possible in the translated function.
Only response the translated function results.

```
int OAEP_ENCODE(int sha,const octet *m,csprng *RNG,const octet
*p,octet *f){
    ……
}
```

```
fn hashit(sha: usize, a: Option<&[u8]>, n: isize, w: &mut [u8]) {
   if sha == SHA256
      ……
}
……
```

```
pub struct RAND {
   ira: [u32; RAND_NK], /* random number...   */
   rndptr: usize,
   borrow: u32,
   pool_ptr: usize,
   pool: [u8; 32],
}
```

```
pub const RFS: usize = (big::MODBYTES as usize) * ff::FFLEN;
```

```
use crate::rand::RAND;
……
```

you were asked to translate the given c function to rust and execute your response and get some error message.
Fix the bug in your previous response according to previous response, error message, c function, rust function signature, rust function dependencies(including function and variable dependencies), and data type declarations and rust function dependency libraries I provide(delimited with XML tags).
Only response the function results.

```
pub fn oaep_encode(sha: usize, m: &[u8], rng: &mut RAND, p: Option<&[u8]>, f: &mut [u8]) -> bool {
   let slen = f.len() - 1;
   …
}
error[E0435]: attempt to use a non-constant value in a constant
…
```

```
int OAEP_ENCODE(int sha,const octet *m,csprng *RNG,const octet *p,octet *f){
      ……
}
```

```
fn hashit(sha: usize, a: Option<&[u8]>, n: isize, w: &mut [u8]) {
   if sha == SHA256
      ……
}
……
```

```
pub struct RAND {
   ira: [u32; RAND_NK], /* random number...   */
   rndptr: usize,
   borrow: u32,
   pool_ptr: usize,
   pool: [u8; 32],
}
```

```
pub const RFS: usize = (big::MODBYTES as usize) * ff::FFLEN;
```

```
use crate::rand::RAND;
……
```

Fig. 4. Translating Prompt and Debugging Prompt

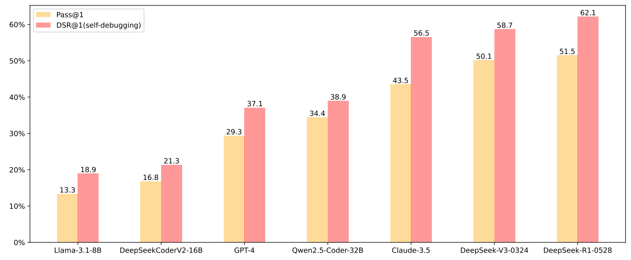


Fig. 5. Pass@1, DSR@1(self-debugging) on RustRepoTrans

prompts (Fig. 4) excluded dependency instructions, and the hyperparameters remained consistent with RQ1.

2) Results: The results presented in Table III reveal significant differences in the *Pass@1* and *DSR@1* scores of LLMs across various datasets. Notably, while LLMs achieve high accuracy rates—with DeepSeek-R1 being the top performer, attaining a *Pass@1* of 73.7% and a *DSR@1* of 89.0%—on established benchmarks, their performance drastically drops when evaluated on RustRepoTrans.

Specifically, the *Pass@1* scores for RustRepoTrans are significantly lower, with a decline of approximately 16.2% to 30.8% compared to the higher scores observed on CodeTransOcean. Moreover, after self-debugging, the accuracy gap further widens, with a decline of 26.3% to 38.4% in *DSR@1*. These notable decreases highlight the increased complexity and unique challenges posed by RustRepoTrans, reinforcing the need for models to effectively manage repository-level dependencies and navigate intricate code structures in real-world scenarios. Furthermore, these complexities and challenges are less easily corrected through simple error message feedback.

It is also important to note that Claude-3.5 and DeepSeek-R1, which exhibit a clear performance gap on RustRepoTrans, demonstrate very similar results on CodeTransOcean. This

TABLE III
PASS@1 AND DSR@1 COMPARISON OF LLMS' PERFORMANCE ON CODETRANSOCEAN AND RUSTREPOTRANS

Model	Dataset	Pass@1	DSR@1
DeepSeek-R1	CodeTransOcean	73.7%	89.0%
	RustRepoTrans	**51.5%(↓22.2%)**	**62.1%(↓26.9%)**
DeepSeek-V3	CodeTransOcean	66.3%	85.0%
	RustRepoTrans	**50.1%(↓16.2%)**	**58.7%(↓26.3%)**
Claude-3.5	CodeTransOcean	74.3%	87.7%
	RustRepoTrans	**43.5%(↓30.8%)**	**56.5%(↓31.2%)**
Qwen-2.5-coder-32B	CodeTransOcean	56.7%	77.3%
	RustRepoTrans	**34.4%(↓22.3%)**	**38.9%(↓38.4%)**

indicates that existing stand-alone, function-level datasets can only evaluate a model's basic code translation capability and are insufficient for effectively distinguishing between models with stronger code abilities. Therefore, a more challenging benchmark like RustRepoTrans, which incorporates repository-level context and targets incremental translation scenarios, is necessary to provide a finer-grained and multifaceted evaluation of the models' code translation capabilities.

3) Summary: Our benchmark's repository-level context poses greater challenges for LLMs, reducing their *Pass@1* and *DSR@1* scores by 16.2% to 30.8% and 26.3% to 38.4%, respectively, while simultaneously providing greater differentiation between models that perform similarly on existing datasets. This underscores the benchmark's effectiveness in evaluating real-world code translation capabilities and highlights the need for models to better handle project dependencies.

D. RQ3: 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.

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 IV-B), resulting in 1,748 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. [16], 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 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 (1,614 code samples), which accounted for the vast majority, 92.3%, of the failures as shown in Table IV (see Section IV-D2). Using an open coding approach [58], 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.

2) Results: **Error Outcome Distribution.** Table IV shows the error outcome distribution from RQ1 on RustRepoTrans, alongside data from Pan et al. [16] for translations to other programming languages (C, C++, Python, Java).

In RustRepoTrans, compilation errors account for 92.3% 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.

In unsuccessful translations, the number of compilation errors ranged from 1 to 193 (average: 7.7, median: 3), complicating 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 [58]. Table V presents the top 10 compilation errors encountered. 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 [59], [60].

Error Cause Taxonomy. We conduct open coding [61] with an iterative consensus on error messages to annotate error categories. During each iteration, two annotators with over 6 years of programming experience independently labeled errors by fully considering the error description of the error message and the context of the failed code translation and then resolved disagreements through discussion. After three rounds,

TABLE IV
COMPARISON OF THE PROPORTIONS OF ERROR TYPES BETWEEN RUSTREPOTRANS AND [16].CE: COMPILATION ERRORS, RE: RUNTIME ERRORS, FE: FUNCTIONAL ERRORS, NTE: NON-TERMINATING EXECUTION

	Source	Target	CE	RE	FE	NTE
Pan et al. [16]	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**	**92.3%**	**0%**	**7.7%**	**0%**

TABLE V
TOP 10 COMPILATION ERRORS BY RUST COMPILER ON RUSTREPOTRANS

Error Code	Frequency	Description
E0599	4,079	This error occurs when a method is used on a type which doesn't implement it.
E0425	2,095	An unresolved name was used.
E0308	1,035	Expected type did not match the received type.
E0277	976	You tried to use a type which doesn't implement some trait in a place which expected that trait.
E0609	906	Attempted to access a nonexistent field in a struct.
E0433	639	An undeclared crate, module, or type was used.
E0061	343	An invalid number of arguments was passed when calling a function.
E0252	290	Two items of the same name cannot be imported without rebinding one of the items under a new local name.
E0432	287	An import was unresolved.
E0616	187	Attempted to access a private field on a struct.

the categories stabilized, ensuring consistent and reliable results. We randomly selected 50 error cases and asked another author to independently label them. The Cohen's kappa [62] reached 0.885, which indicates almost perfect agreement, representing the validity and reproducibility of the error categories obtained.

Fig. 6 shows the ten final error cause categories for failed code translations with compilation errors in RustRepoTrans, further grouped 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 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 detailed 10 error causes from the types are 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

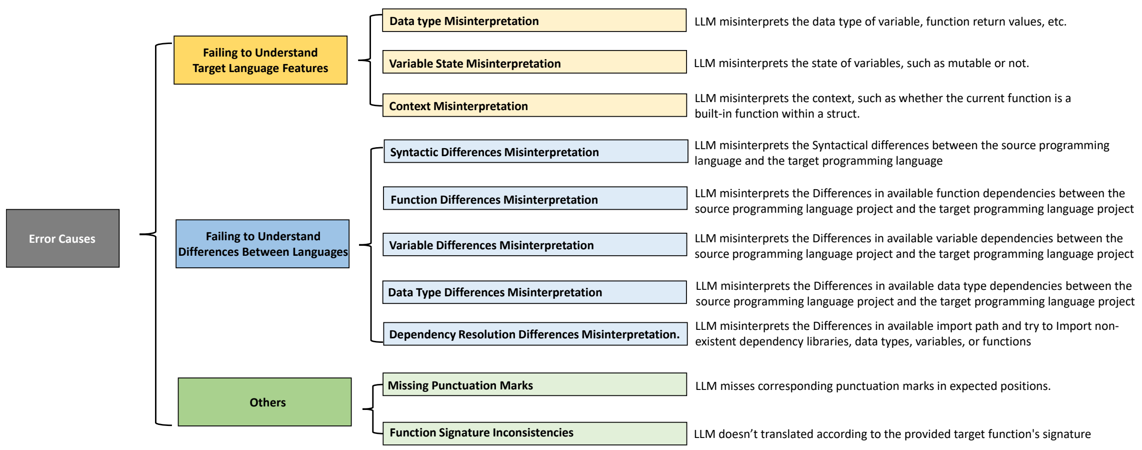

Error Causes
Failing to Understand Target Language Features
Data type Misinterpretation
LLM misinterprets the data type of variable, function return values, etc.
Variable State Misinterpretation
LLM misinterprets the state of variables, such as mutable or not.
Context Misinterpretation
LLM misinterprets the context, such as whether the current function is a built-in function within a struct.
Failing to Understand Differences Between Languages
Syntactic Differences Misinterpretation
LLM misinterprets the Syntactical differences between the source programming language and the target programming language
Function Differences Misinterpretation
LLM misinterprets the Differences in available function dependencies between the source programming language project and the target programming language project
Variable Differences Misinterpretation
LLM misinterprets the Differences in available variable dependencies between the source programming language project and the target programming language project
Data Type Differences Misinterpretation
LLM misinterprets the Differences in available data type dependencies between the source programming language project and the target programming language project
Dependency Resolution Differences Misinterpretation.
LLM misinterprets the Differences in available import path and try to Import non-existent dependency libraries, data types, variables, or functions
Others
Missing Punctuation Marks
LLM misses corresponding punctuation marks in expected positions.
Function Signature Inconsistencies
LLM doesn't translated according to the provided target function's signature

Fig. 6. Error Cause Taxonomy for Failed Code Translations on RustRepoTrans in LLMs

Rust's strict typing exposes LLM's 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 LLMs' 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.

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

Context Misinterpretation. This error cause arises from the LLM's misunderstanding of context. It includes issues such as failing 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())` with using the keyword `self`, which is unnecessary 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 keyword 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), attempting to use the `goto` keyword from C, or employing an unsupported 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 with identical function calling way. In the example below, the first line results in an error because the LLM fails to recognize that get should be called within the `Self` prefix.

```
resort(&mut items_vec);        # error
Self::resort(&mut items_vec);        # 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 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.

Overall Distribution of Error Causes. As shown in Fig. 7, 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 73.9% of the errors. This is followed by *Failing to Understand Target Language Features* at 22.4%, 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* (38.6%), *Variable Differences Misinterpretation* (24.9%), and *Data Type Misinterpretation* (16.1%). 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. 8 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 publicly. This results in comparable understanding of the target language across models. In contrast, *Failing to Understand Differences Between Languages* and *Other* errors are more common in LLMs with weaker translation capabilities. The former evaluates the model's understanding of language differences, while the latter reflects its ability to follow instructions, both positively correlating with translation performance.

Notably, DeepSeek-R1, the top-performing model on RustRepoTrans, exhibits the lowest rate of *Syntactic, Function and Variable Differences Misinterpretation*, representing its exceptional comprehension capabilities in capturing cross-lingual discrepancies. Furthermore, Claude-3.5 made no errors on simpler error types such as *Missing Punctuation Marks*

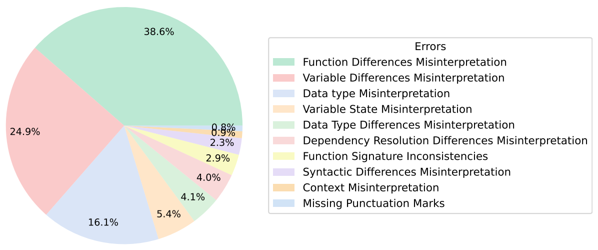


Fig. 7. Overall Distribution of Error Causes

and *Function Signature Inconsistencies*, indicating superior understanding and adherence to instructions compared to others.

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.

E. RQ4: Key Capabilities

Beyond Pass@1 in RQ1, we analyze LLMs' translation capabilities in noise robustness, syntactical difference identification, and code simplicity.

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 Section IV-B 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 [63]) 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%. 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. 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 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

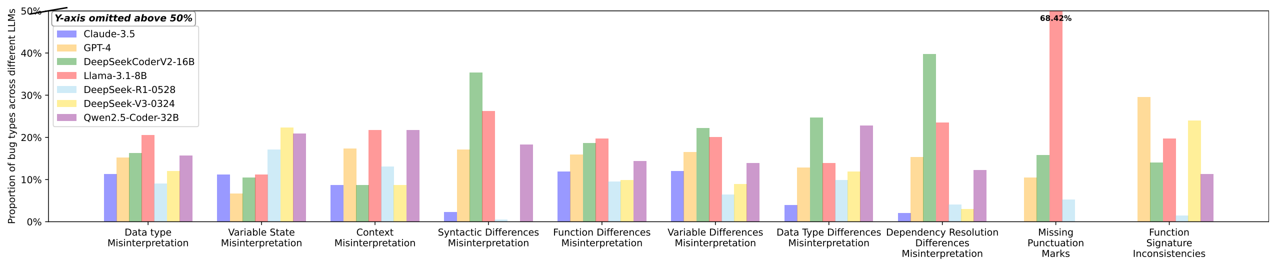
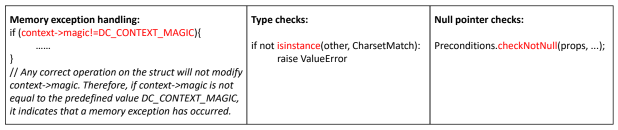


Fig. 8. Comparative Distribution of Error Causes in LLMs



Memory exception handling:	**Type checks:**	**Null pointer checks:**
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.*	if not isinstance(other, CharsetMatch): raise ValueError	Preconditions.checkNotNull(props, ...);

Fig. 9. Examples of checks from other languages that are not needed in Rust

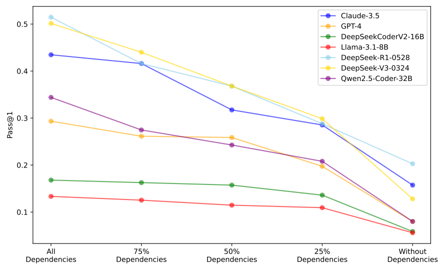


Fig. 10. The *Pass@1* of LLMs under different proportions of dependencies

pointer checks (in C and Java) shown in Fig. 9. We selected 89 function pairs from RustRepoTrans that include these checks in the source language and evaluated whether the LLM correctly identifies and omits in the target language. We introduce a novel metric, Syntactical Differences Identification Rate (*SDIR@K*), which measures the LLM's success in identifying these syntactical differences 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 [40] and measuring cyclomatic complexity [64], where higher values indicate greater complexity. We apply rust-code-analysis [65] to the reference and translated code that passed the test cases. A higher ratio suggests that the translated code is more concise and closely 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 simpler generated code and is more aligned with the original.

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, DeepSeek-R1 and DeepSeek-V3, the two LLMs that ranked top-2 in terms of Pass@1 and DSR@1, exhibited larger fluctuations, with RIR values of 92.7% and 86.2%, highlighting potential areas for improvement in their noise robustness regarding redundant dependencies. From the perspective of incomplete dependencies, as shown in Fig. 10, 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. This is because 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. In contrast, other LLMs faced significant drops at nearly every stage, likely because their successfully translated tasks include those with multiple dependencies. As shown in Fig. 11, the IIR ranking among models are almost the same as the Pass@1 under All Dependencies in IV-B, DeepSeek-R1 demonstrated the best performance with IIR values of 35.8%.

Syntactical Differences Identification. As shown in Fig. 11, DeepSeek-V3, DeepSeek-R1 and Claude-3.5 achieve high *SDIR@1* at 97.8%, 96.6% and 95.5% separately, demonstrating a strong ability to identify syntactical differences between programming languages. This finding is consistent with the results in Section IV-D regarding the distribution of error causes across different LLMs (see Fig. 8). In contrast, DeepSeekCoderV2-16B has the lowest *SDIR@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.

Code Simplicity. Fig. 11 shows that among the successfully translated tasks, the Token Rate for code generated by DeepSeek-R1 and DeepSeek-V3 reached 100.9% and 99.5%, 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 and DeepSeek-R1 excel with a CC Rate of 103.7% and 103.2% separately, while Llama-3.1-8B is notably lower at 53.6%. This difference arises because Claude-3.5 and DeepSeek-R1 often uses concise built-in functions in place of loops. For instance, as illustrated in Fig. 12, when tasked with finding the first element in an array that meets a condition, they employ the target language's `any` method from the iterator library, whereas other LLMs rely on traditional loops. Furthermore, the CC Rate of Claude-3.5 and DeepSeek-R1 exceeding 1 indicates that, in terms of

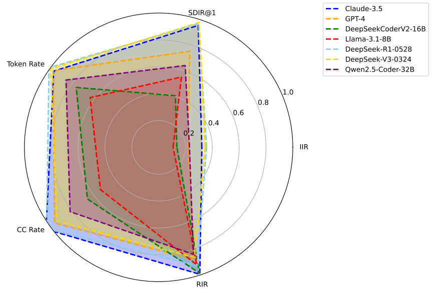
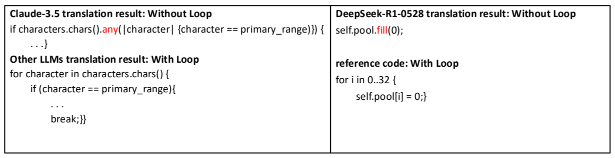


Fig. 11. The performance of LLMs on Key Abilities of code translation.



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;}}
```
DeepSeek-R1-0528 translation result: Without Loop
```
self.pool.fill(0);
```
reference code: With Loop
```
for i in 0..32 {
    self.pool[i] = 0;}
```

Fig. 12. Claude-3.5 and DeepSeek-R1 translation result compared to Other LLMs and reference code

cyclomatic complexity, their generated code is simpler than the reference code. Fig. 12 shows an example with C as the source language. This simplification likely results from Rust ground truth developers using the C2Rust migration tool [66], which translates code through one-to-one pattern matching. It is also important to note that although DeepSeek-V3 outperforms Claude-3.5 in terms of pass@1, its CC Rate is 93.5%, which is significantly lower than that of Claude-3.5. This suggests that a model with a lower pass@1 score does not necessarily indicate weaker code capabilities, and the quality of the code it generates may actually be higher.

These findings suggest that DeepSeek-R1 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.

3) Conclusion: The complexity of code generated by LLMs reflects their translation capabilities: models with stronger abilities produce code with lower complexity for equivalent functionality. DeepSeek-R1, which excels on RustRepoTrans, also shows superior performance in Noise Robustness, Syntactical Differences Identification, and Code Simplicity. Meanwhile, although Claude-3.5 does not achieve the highest pass@1, it demonstrates exceptional performance in Code Simplicity, highlighting the high quality of the code it generates.

V. Discussion

In this section, we further discuss the intended audience and the limitation of RustRepoTrans.

Intended Audience. The RustRepoTrans is designed to provide value to both researchers and practitioners in the field of code translation. Not only do the error causes revealed by RustRepoTrans for different LLMs, along with the associated analysis, offer fine-grained insights for researchers in future work, but the relative performance ranking of various LLMs can also guide developers in choosing suitable base models when constructing their translation tools in real-world scenarios.

Limitations. In our evaluation, the provided information contains comprehensive and non-redundant contextual information. We did not conduct in-depth assessments of two specific aspects: the migration of global variables, type definitions, or function signatures, and the integration of LLMs with static/dynamic analysis. This is because the primary focus of this paper is to achieve a more authentic, practical, and fine-grained evaluation for the precise diagnosis and targeted improvement of LLMs' code translation capabilities under an incremental translation scenario. However, we provide the complete target codebase to enable any related future work incorporating such analyses.

VI. 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. Lastly, the reliability and completeness of the error causes taxonomy pose another potential threat. We employed open coding to systematically identify and categorize the error causes, adhering to established open coding practices to ensure thoroughness and accuracy. Concurrently, we constructed an iterative refining process, reducing individual biases and ensuring more objective assessments. Finally, we engaged a non-participant to annotate the same error messages based on the obtained error cause categories, achieving a Cohen's kappa of 0.885, indicating almost perfect agreement.

VII. Conclusion

This work makes the first attempt to evaluate LLMs on repository-level context code translation targeting Rust. We first manually construct the first repository-level context code translation benchmark RustRepoTrans and evaluate seven representative LLMs. We find that existing LLMs show much worse performance on incremental repository-level context code translation compared to standalone code translation and exhibit more fine-grained deficiencies compared to end-to-end full repository translation. 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. Meanwhile, we propose a set of more fine-grained evaluation metrics and an enhanced evaluation framework, enabling a more comprehensive analysis of LLMs' performance in repository-level context code translation tasks to provide fine-grained insights that can effectively inform the development of code translation techniques.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62032025, 62402113), CCF - Sangfor 'Yuanwang' Research Fund.

References

[1] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Understanding the effectiveness of large language models in code translation," *CoRR*, 2023.

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

[3] "Supporting linux kernel development in rust," https://lwn.net/Articles/829858/, 2020.

[4] "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/ 2020.

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

[6] "Github's journey from monolith to microservices," https://www.infoq.com/articles/github-monolith-microservices/, 2020.

[7] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

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

[9] "Cangjie," https://developer.huawei.com/consumer/cn/cangjie/, 2025.

[10] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos, "Code migration through transformations: An experience report," in *CASCON First Decade High Impact Papers*, 2010, pp. 201–213.

[11] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, *CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution*. IOS Press, Oct. 2024. [Online]. Available: http://dx.doi.org/10.3233/FAIA240968

[12] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 695–707.

[13] Copilot, "Using github copilot to migrate a project to another programming language," 2024. [Online]. Available: https://docs.github.com/en/copilot/tutorials/migrate-a-project

[14] B. G. Mateus, M. Martinez, and C. Kolski, "Learning migration models for supporting incremental language migrations of software applications," *Information and Software Technology*, vol. 153, p. 107082, 2023.

[15] J. Brant and D. Roberts, "The smacc transformation engine: how to convert your entire code base into a different programming language," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 809–810. [Online]. Available: https://doi.org/10.1145/1639950.1640026

[16] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 995–1007.

[17] R. Security, "Cisa's 2026 memory safety deadline: What ot leaders need to know now," 2024. [Online]. Available: https://thenewstack.io/feds-critical-software-must-drop-c-c-by-2026-or-face-risk/

[18] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in neural information processing systems*, vol. 33, pp. 20 601–20 611, 2020.

[19] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[20] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x," *arXiv preprint arXiv:2303.17568*, 2023.

[21] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1529–1541.

[22] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[23] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, "Codetransocean: A comprehensive multilingual benchmark for code translation," *arXiv preprint arXiv:2310.04951*, 2023.

[24] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, "Xlcost: A benchmark dataset for cross-lingual code intelligence," *arXiv preprint arXiv:2206.08474*, 2022.

[25] Y. Wang, Y. Wang, S. Wang, D. Guo, J. Chen, J. Grundy, X. Liu, Y. Ma, M. Mao, H. Zhang *et al.*, "Repotransbench: A real-world benchmark for repository-level code translation," *arXiv preprint arXiv:2412.17744*, 2024.

[26] X. Zhang, J. Wen, F. Yang, P. Zhao, Y. Kang, J. Wang, M. Wang, Y. Huang, E. Nallipogu, Q. Lin *et al.*, "Skeleton-guided-translation: A , benchmarking framework for code repository translation with fine-grained quality evaluation," *arXiv preprint arXiv:2501.16050*, 2025.

[27] "Rustrepotrans," https://github.com/SYSUSELab/RustRepoTrans/, 2024.

[28] "Rustrepotrans," https://huggingface.co/datasets/SYSUSELab/RustRepoTrans, 2024.

[29] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 651–654.

[30] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in neural information processing systems*, vol. 31, 2018.

[31] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016.

[32] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[33] ——, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," *arXiv preprint arXiv:1810.02720*, 2018.

[34] I. J. Rithy, H. Hossain Shakil, N. Mondal, F. Sultana, and F. M. Shah, "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)*, 2022, pp. 623–628.

[35] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[36] M. Macedo, Y. Tian, F. R. Cogo, and B. Adams, "Exploring the impact of the output format on the evaluation of large language models for code translation," in *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge) Conference Acronym:*, 2024, pp. 57–68.

[37] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray, "C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques," *arXiv preprint arXiv:2501.14257*, 2025.

[38] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, "Scalable, validated code translation of entire projects using large language models," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1616–1641, 2025.

[39] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 801–824, 2025.

[40] tree sitter, 2023. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[41] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 195–204.

[42] A. Trotman, A. Puurula, and B. Burgess, "Improvements to bm25 and language models examined," in *Proceedings of the 19th Australasian Document Computing Symposium*, 2014, pp. 58–65.

[43] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1184–1196.

[44] S. Chen, R. Varma, A. Sandryhaila, and J. Kovačević, "Discrete signal processing on graphs: Sampling theory," *IEEE Transactions on Signal Processing*, vol. 63, no. 24, pp. 6510–6523, 2015.

[45] OpenAI. (2024) Openai api interface. [Online]. Available: https://platform.openai.com/docs/api-reference/introduction

[46] Claude. (2024) Claude api interface. [Online]. Available: https://docs.anthropic.com/en/api/getting-started

[47] DeepSeek, “Deepseek r1 0528 release,” 2025. [Online]. Available: https://api-docs.deepseek.com/news/news250528

[48] ——, “Deepseek-v3-0324 release,” 2025. [Online]. Available: https://api-docs.deepseek.com/news/news250325

[49] Aliyun, “Qwen coder api,” 2025. [Online]. Available: https://help.aliyun.com/zh/model-studio/qwen-coder#844e7528bbs7w

[50] M. Macedo, Y. Tian, F. R. Cogo, and B. Adams, “Exploring the impact of the output format on the evaluation of large language models for code translation,” in *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge) Conference Acronym:*, 2024, pp. 57–68.

[51] M. A. M. Khan, M. S. Bari, D. Long, W. Wang, M. R. Parvez, and S. Joty, “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)*, 2024, pp. 6766–6805.

[52] Anthropic, “Claude 3.5 sonnet,” https://www.anthropic.com/news/claude-3-5-sonnet, 2024.

[53] OpenAI, “Gpt-4,” https://openai.com/index/gpt-4-research/, 2023.

[54] Meta, “Llama-3.1-8b-instruct,” https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, 2024.

[55] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, “Qwen2. 5-coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.

[56] DeepSeek, “Deepseek-coder-v2-lite-instruct,” https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct, 2024.

[57] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[58] R. error codes, 2024. [Online]. Available: https://doc.rust-lang.org/error_codes/error-index.html

[59] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang *et al.*, “Repoagent: An llm-powered open-source framework for repository-level code documentation generation,” *arXiv preprint arXiv:2402.16667*, 2024.

[60] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, “Teaching code llms to use autocompletion tools in repository-level code generation,” *arXiv preprint arXiv:2401.06391*, 2024.

[61] S. H. Khandkar, “Open coding,” *University of Calgary*, vol. 23, no. 2009, p. 2009, 2009.

[62] M. L. McHugh, “Interrater reliability: the kappa statistic,” *Biochemia Medica*, vol. 22, pp. 276 – 282, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:5421278

[63] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[64] G. Gill and C. Kemerer, “Cyclomatic complexity density and software maintenance productivity,” *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[65] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, “rust-code-analysis: A rust library to analyze and extract maintainability information from source codes,” *SoftwareX*, vol. 12, p. 100635, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711020303484

[66] C2Rust, 2024. [Online]. Available: https://github.com/immunant/c2rust