Polyglot: An Extensible Framework to Benchmark Code Translation with LLMs

Marco Vieira, Priyam Ashish Shah, Bhavain Shah, Rrezarta Krasniqi
College of Computing and Informatics
University of North Carolina at Charlotte, NC, USA
{marco.vieira, pshah75, bshah17, rrezarta.krasniqi}@charlotte.edu

***Abstract*—Large Language Models (LLMs) show great potential for automating code-related tasks. However, sound assessments are necessary to understand their true capabilities, particularly in code translation, where reliability is crucial. We introduce `Polyglot`, an automated, multi-language framework for evaluating the translation quality of LLMs between different programming languages. Leveraging the IBM CodeNet Project, an extensive collection of coding problems in multiple languages, we assess translation quality using syntactic correctness, execution reliability, semantic preservation, and static code metrics. Our evaluation focuses on translating C to Java, Python, and Rust, languages that follow distinct paradigms and represent alternatives to modernize C-based systems. We evaluate open-source LLMs using three prompting strategies to understand the impact on translation performance. Our findings highlight that while LLMs show promising results for simple code translation, their limitations regarding complex logic and distinct language paradigms require further analysis.**

***Index Terms*—Code Translation, LLM for Code, Code Quality**

I. Introduction

Software systems must often be migrated or modernized due to technological advances, maintainability concerns, or performance improvements. For example, a significant trend involves migrating legacy applications written in C into modern languages to enhance security and maintainability [1]. However, manual code translation is time-consuming, error-prone, and demands expertise in the source and target languages, making automated code translation a key area of research [2].

Large Language Models (LLMs) have recently demonstrated strong performance across a range of code-related tasks, including code completion [3], fault and vulnerability detection [4], [5], code patching [6], and automated code generation [7]–[9], among others [10]–[12]. Leveraging their emerging capabilities in natural language translation [13], [14], LLMs hold great potential for *automating code translation*, which can reduce migration effort and improve reliability. However, despite the growing interest in LLM-based code translation [15]–[17], two critical limitations persist: *i)* code reliability remains a key challenge, particularly with complex constructs and semantic gaps [18], and *ii)* existing evaluations are fragmented, relying on ad hoc setups that lack extensibility and reproducibility [8], [9], [19].

This paper presents `Polyglot`, an extensible and fully automated framework designed to rigorously overcome these limitations by providing a structured methodology comprising: (1) comprehensive evaluation metrics, (2) real-world programs of varying complexity, (3) systematic prompting strategies, and (4) a repeatable evaluation process. Our assessment targets four key aspects of translation quality: *i)* syntactic correctness (successful compilation), *ii)* execution reliability (absence of runtime errors), *iii)* semantic preservation (functional equivalence via test cases), and *iv)* static code metrics (analysis of complexity and size variation).

We instantiate the framework using the IBM CodeNet dataset [20], which provides a diverse set of verified solutions that we use to support the controlled evaluation of LLM-generated translations. To improve coverage, we augment the dataset with LLM-generated test cases and assess the impact of prompting strategies by comparing standard zero-shot (`S-ZS`), curated zero-shot (`C-ZS`), and Chain of Thought (`CoT`) approaches. `PolyglotWeb.site` is a public repository to support transparency and continuous assessment by publishing raw data and expanding evaluation results over time.

We assess the ability of seven open-source LLMs to translate C programs into Python, Java, and Rust. We focus on open-source models to ensure transparency, reproducibility, and cost-effectiveness. While not exhaustive, this setup showcases `Polyglot`'s utility and extensibility, while exposing key challenges in LLM-based code translation. We chose Python, Java, and Rust for their contrasting paradigms and relevance in modernizing C-based systems: Python emphasizes high-level abstraction and automatic memory management; Java introduces strong typing and object-oriented design; and Rust presents the most significant challenge due to its strict compile-time memory safety guarantees.

Our comparative analysis, beyond isolated language pairs and ad hoc evaluations, shows that LLM performance varies notably across target languages. While LLMs handle simple constructs well, complex features remain a problem. Java translations tend to be more accurate, whereas Rust presents significant challenges. We also found that LLM-generated code frequently introduces syntactic errors and semantic bugs, impacting reliability and maintainability. These findings highlight the need for robust evaluation frameworks such as `Polyglot` to support the development of more capable and trustworthy code translation systems.

To summarize, the key contributions of this work are:

- `Polyglot`, **a novel, fully automated, and extensible framework** for rigorous and reproducible assessment of LLM-based code translation. It defines all essential

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00195

components (evaluation metrics, real-world programs and test suites, prompting strategies, and a procedure) to overcome the limitations of ad hoc evaluations.
- **A comparative evaluation of open-source LLMs for translating C into Java, Python, and Rust**. Our structured analysis contrasts with fragmented prior work and reveals performance variations across distinct language paradigms.
- **Guidelines for integrating new workloads, advanced metrics, and diverse prompting strategies**. This design ensures the framework remains extensible and adaptable for evolving code translation scenarios.
- `PolyglotWeb.site`, **a public repository for publishing up-to-date evaluation results**, fostering collaboration in the LLM-based code translation community.

The remainder of the paper is as follows. Section II reviews related work. Section III introduces `Polyglot`. Sections IV and V describe the experimental setup and results. Section VI discusses key observations. Section VII outlines threats to validity. Section VIII concludes and puts forward future work.

II. Background and Related Work

LLMs are transforming software engineering by enabling code generation, bug detection, and documentation automation [21]. Open-source (e.g., LLaMA, DeepSeek) and commercial models (e.g., GPT-4, Gemini), trained on large code corpora, learn patterns to handle complex tasks. However, code translation remains challenging, as it requires adapting both syntax and semantics across diverse programming paradigms [18], [22]. Unlike traditional transpilers, which rely on explicit, rule-based mapping (e.g., AST transformations) and are designed specifically for translation [23], LLMs rely on pattern-based inference.

Code translation has traditionally relied on rule-based transpilers and neural machine translation (NMT) models [24], [25]. Rule-based systems map source to target constructs but are rigid and struggle with complex transformations [26], [27]. NMT models, adapted from natural language processing, offer improvements but often fail to capture deeper structural and semantic relationships essential for accurate translation. Hybrid approaches that combine static and dynamic analysis, AST transformations, and deep learning aim to improve translation accuracy, but still struggle with memory management, concurrency, and paradigm shifts [28], [29].

Pretrained models such as CodeBERT [7] and GraphCodeBERT [30] introduced multilingual code translation, though performance varied across language pairs. AI TransCoder [31] used unsupervised learning to translate between Python, Java, and C++, but struggled with fundamental structural transformations, especially for low-level C constructs. While LLMs have since shown implicit translation abilities, their reliability remains poorly understood, heavily influenced by prompt design, with persistent challenges in semantic preservation and adaptation across diverse programming paradigms [15]–[18].

Reference-based approaches evaluate generated code by comparing it to human-written references using metrics such as *BLEU* [32], *CodeBLEU* [33], and *Computational Accuracy (CA)* [34]. Syntactic correctness is often assessed via compilation success [35], while semantic preservation is typically measured by executing test cases. Similarity metrics, such as Levenshtein distance and AST-based comparisons (e.g., in CodeBLEU), measure alignment with the reference code. Cyclomatic complexity [36] captures structural overhead, and dependency-based metrics such as *DEP* [37] assess structural coherence. Embedding-based metrics such as *CodeBERTScore* [38], structure-aware *RUBY* [39], and the reference-free *ICE-Score* [40] provide deeper semantic evaluation. However, these sophisticated metrics can be difficult to reproduce or interpret, underscoring the importance of complementary, explainable metrics.

Several recent benchmarks have focused on evaluating LLM-based code translation. For example, *CodeXGLUE* [41] provides a broad collection of tasks and metrics, establishing a common ground for evaluation. Still, it relies primarily on reference-based measures (e.g., BLEU, CodeBLEU), supports only a limited number of language pairs, and offers little emphasis on test-based semantic validation or extensibility. *CodeAlpaca* [42] provides curated prompts and solutions that have been widely adopted in practice; however, it omits functional test cases and does not address translation-specific tasks, thereby limiting its relevance for code migration scenarios. *TransCoder* [31] was an influential early effort toward multilingual translation, demonstrating the potential of unsupervised learning, but it relied heavily on synthetic training data, and its evaluation lacks transparency and reproducibility.

Despite progress in LLM-based code translation, there is a lack of comprehensive and extensible evaluation frameworks [16], [21]. Many studies rely on narrow metrics or limited translation scenarios [18], [34], [43], which, while useful for large-scale automated evaluation, overlook aspects such as code complexity and semantic preservation. Inadequate language and dataset coverage further limit generalizability [15], [44], while inconsistent evaluation procedures hinder reproducibility and cross-study comparisons [8], [9].

In contrast to prior work, `Polyglot` introduces a multidimensional evaluation approach using real-world programs of varying complexity and functionality. Designed for scalability and adaptability, it supports consistent assessment across LLMs, prompting strategies, and target languages. Our evaluation, which translates C to Python, Java, and Rust, shows `Polyglot`'s applicability as a foundation for rigorous, automated, and reproducible comparisons that reveal the capabilities and limitations of LLM-based code translation.

III. The Polyglot Framework

`Polyglot` is a systematic framework for evaluating LLMs on code translation tasks, grounded in established benchmarking principles to ensure rigor and broad applicability. Its core goals are to be *representative* by using real-world problems, produce *repeatable* results, and remain *nonintrusive*, preserving natural model behavior. The framework supports workload *scalability*, offers partial *portability* across models

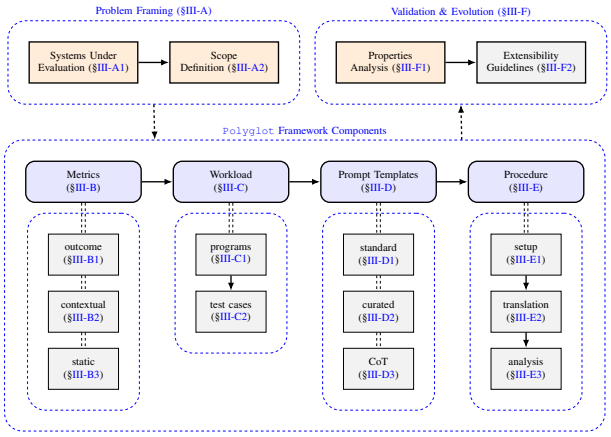

Fig. 1. Design of the Polyglot Framework

and tasks (acknowledging that complete portability, namely installability, and replaceability, are out of scope), and promotes *simplicity* through automation. It distinguishes itself with a clearly defined scope and built-in extensibility, enabling it to evolve alongside future research needs.

Polyglot design comprises three elements (Figure 1). **Problem Framing** defines the systems under evaluation and sets the framework's scope. The **Polyglot Framework Components** specify the key elements that support the assessment: *metrics* for evaluating translation quality, a *workload* of real-world programs and test cases, *prompt templates* to guide translation, and a *procedure* for experiment setup, execution, and analysis. Finally, **Validation and Evolution** ensures alignment with the core design principles and provides guidelines for extensibility, supporting new languages, metrics, workloads, and prompts.

Although our approach is intrinsically empirical, it is grounded in well-established assessment practices. Consistent with any evaluation framework, Polyglot abstracts practical use cases into a controlled setting, and direct generalization to other scenarios is neither intended nor supported. For instance, our evaluation focuses on translating C into Python, Java, and Rust, so we should interpret conclusions within the scope of these specific language pairs and the workload used.

A. Problem Framing

1) Systems Under Evaluation: Characterizing the systems under evaluation is a critical aspect of Polyglot's design. It defines the role of LLMs in code translation and ensures the framework captures the practical challenges posed across languages and paradigms. This characterization informs key operational assumptions and core evaluation requirements, enabling structured, repeatable comparisons and supporting adaptation to evolving LLM capabilities.

Our assumptions include evaluating LLMs in a closed-box setting and accessing them via APIs or prompts without internal modifications. This approach reflects common deployment scenarios and motivates a nonintrusive methodology. We assume model-agnostic conditions, avoiding reliance on vendor-specific features, training configurations, or architectures to ensure consistent, comparable assessment across LLMs.

Requirements for code translation include syntactically valid output, preservation of semantic behavior, and appropriate adaptation of language-specific constructs, even in the absence of direct mappings. These capabilities, shaped by factors such as model architecture, training data, tokenizer design, and pretraining objectives [16], constrain generalization across languages and abstraction levels. Inference-time variables, including sampling strategies, prompt design, and context limits, introduce variability that one must account for.

2) Scope Definition: Establishing a clear scope is essential for defining a practical evaluation framework. It determines the capabilities to assess, the nature of the translation tasks, and the success criteria. A well-defined scope ensures alignment between framework components and goals, maintains focus, and enables meaningful comparisons across models.

Polyglot evaluates LLMs on code translation tasks involving complete programs (not code snippets) while preserving functional behavior and structural integrity. This approach enables a multidimensional assessment across *syntactic correctness*, *execution reliability*, *semantic preservation*, and *code complexity*. To reflect real-world translation challenges, the framework includes code of varying complexity and constructs, such as loops, conditionals, memory operations, and exception handling.

The defined scope also sets specific evaluation conditions: Polyglot currently operates under zero-shot prompting without in-context examples, reflecting typical usage, and evaluates a single translation per program. While alternative strategies (e.g., multi-sample generation or re-ranking) may improve performance, they introduce additional variables beyond the current scope. Nonetheless, Polyglot is designed for extensibility, supporting new languages, metrics, prompt types, and evaluation procedures through clearly defined guidelines.

B. Metrics

Polyglot uses three groups of metrics: (1) *outcome-based metrics*, summarizing overall success (e.g., test pass rates); (2) *contextual metrics*, identifying failure points in the pipeline (e.g., compilation or runtime errors); and (3) *static metrics*, assessing structural preservation or degradation. We chose these types of metrics to balance syntactic and semantic insight with interpretability, avoiding reliance on reference translations or complex scoring. For fair comparison, metrics are computed independently per language pair and LLM. While aggregation can offer broader trends, one must use it cautiously to avoid masking model-specific behaviors or translation challenges.

1) Outcome-Based Metrics: We classify each translation into one of four mutually exclusive categories, capturing the full spectrum of success and failure modes: fails to compile (F_1); compiles but fails at runtime (e.g., crash or timeout) (F_2); compiles and runs but fails test cases (F_3); and passes all test cases (F_4). These categories form a complete partition: $F_1 + F_2 + F_3 + F_4 = 100\%$. Given a set of translations

T, we define the outcome-based metrics as the percentage of translations falling into each category F_j, computed as:

$$F_j = \frac{1}{|T|} \sum_{t \in T} \mathbb{I}_{F_j}(t) \times 100, \quad \text{for } j \in \{1, 2, 3, 4\}$$

where $\mathbb{I}_{F_j}(t)$ is an indicator function equal to 1 if translation t falls into category F_j, and 0 otherwise. These metrics allow precise identification of failure types, whether syntactic, runtime, or semantic, and enable comparative evaluation of performance across different stages of the translation pipeline.

2) Contextual Metrics: To complement the cumulative F_j categories with stage-level detail, we define a *generalized stage success rate* for each pipeline stage θ:

$$R_\theta = \frac{1}{|T_\theta|} \sum_{t \in T_\theta} \mathbb{I}_\theta(t) \times 100$$

where: $\theta \in \{\text{compile}, \text{run}, \text{pass}\}$ denotes the pipeline stage of interest; $T_\theta \subseteq T$ is the subset of translations for which stage θ is applicable; $\mathbb{I}_\theta(t) = 1$ if translation t successfully completes stage θ, and 0 otherwise; and T is the full set of translations. This formulation enables fine-grained tracking of LLM performance across the translation pipeline. Specifically, the compile rate reflects the percentage of translations that compile; the run rate considers only those that compile and execute without errors; and the pass rate measures correctness among those that compile, run, and pass all test cases. This stage-wise breakdown helps pinpoint strengths and weaknesses in the translation process.

We acknowledge some overlap between contextual and outcome-based metrics, but this is intentional to support a multi-perspective analysis. Metrics such as R_{compile} and F_1 are related yet distinct: contextual metrics (R_*) capture stage-specific success rates (e.g., conditional on compilation), while outcome-based metrics (F_*) provide an end-to-end view across the dataset. This dual perspective enables fine-grained diagnostics (e.g., isolating compilation or runtime failures) as well as macro-level comparisons across models, prompts, and languages, while giving benchmark users the flexibility to choose metrics aligned with their evaluation objectives.

3) Static Metrics: We assess structural properties that impact readability and maintainability, focusing on whether LLM-generated code preserves the characteristics of the source. These metrics are computed only for translations that compile, execute, and pass all test cases to ensure meaningful comparisons. We define two metrics: (1) **Cyclomatic Complexity Variation** ($\Delta CC_{\log}$) quantifies changes in control flow complexity between source and translated code, indicating whether LLMs introduce unnecessary complexity or preserve logical structure; and (2) **Source Lines of Code Variation** ($\Delta SLoC_{\log}$) captures verbosity or compactness of the translated output relative to the source. Both follow a unified formulation:

$$\Delta M_{\log} = \frac{1}{N_{tp}} \sum_{j=1}^{N_{tp}} \log\left(\frac{M_{t_j}}{M_{s_j}}\right)$$

where M denotes a metric (CC or $SLoC$); M_{s_j} and M_{t_j} are the metric values for the source and translated programs, respectively; and N_{tp} is the number of successfully translated programs (i.e., those that compile, run, and pass all tests). The logarithmic transformation ensures symmetric treatment of increases and decreases and scales differences proportionally.

C. Workload

A well-curated workload is critical for evaluating code translation across languages. It should meet key criteria [43], including: *parallel code samples* (equivalent implementations across languages) for cross-language comparison; *functional verification* via test cases to assess semantic preservation; and *diverse problem scopes* spanning algorithmic complexity and code structures. The workload should include languages with differing paradigms, typing systems, memory models, and abstraction levels to reflect real-world translation challenges. Following these principles, the `Polyglot` workload uses real-world programs with functional tests and varied structural properties, supporting robust and meaningful comparisons.

1) Real-World Programs: IBM CodeNet is a large-scale dataset for code-related machine learning tasks [20]. It contains 14 million **code samples** (referred to as "solutions") spanning 4,053 distinct problems across 55 programming languages. Although originating from competitive programming, this dataset includes diverse problems with varied structures, complexity, and constructs (e.g., I/O handling, memory management, control flow) that are common in code translation, particularly for modernizing legacy C code. While it differs from large-scale, domain-specific systems, it offers a controlled and diverse setting that effectively exposes LLM performance differences. Our results show clear variation across model specialization, prompt design, and problem complexity, demonstrating that even a limited yet well-structured benchmark can reveal the strengths and limitations of current translation capabilities.

Several adjustments were needed to adapt IBM CodeNet for integration into the `Polyglot` framework. From the 4,053 available problems, we excluded those lacking English descriptions (necessary for manual inspection), resulting in 2,667 problems, and applied additional filtering per language. For instance, after removing incorrect samples (as per CodeNet metadata), only 1,535 problems have at least six valid C solutions, totaling 276,014 samples. While the minimum sample threshold is configurable, multiple implementations per problem allow analysis of translation consistency, failure modes, and strategy diversity.

We extend the original metadata with metrics that characterize each code sample further. **For every solution**, we compute two standard metrics: Source Lines of Code (*SLoC*) and Cyclomatic Complexity (*CC*) [36]. To complement these, we introduce a Code Construct Feature Score (*CFS*), quantifying the presence and complexity of specific language constructs. While *SLoC* and *CC* capture overall size and control flow, *CFS* offers a finer-grained view by highlighting patterns, such as control structures, memory operations, and nested logic, that pose challenges for LLM-based cross-language translation. *CFS* is a weighted sum of construct occurrences:

$$CFS = w_C C + w_L L + w_R R + w_D D + w_M M + w_T T$$

where: C is the number of conditionals (`if`, `else`, `switch`); L is the number of loops (`for`, `while`, `do-while`); R is an indicator for recursion (1 if present, 0 otherwise); D is the count of unique data structures (e.g., arrays, lists, maps); M is the number of manual memory operations (`malloc`, `free`, `new`, `delete`); T are concurrency constructs (e.g., threads, locks, `async`); and $w_C, w_L, w_R, w_D, w_M, w_T$ are weights reflecting the complexity contribution of each feature.

Note that CFS is not intended to equate independent and nested loops, but to capture the presence of control-flow constructs that often challenge code translation. It is deliberately coarse-grained, supporting high-level complexity stratification without detailed per-problem analysis, which is unnecessary for our current sampling. Future work may extend CFS with depth-aware variants or related enhancements, enabling broader applications such as problem selection or fine-grained difficulty modeling beyond stratification.

At the problem level, we compute the average Source Lines of Code ($SLoC_{avg}$), average Cyclomatic Complexity (CC_{avg}), and average Code Construct Feature Score (CFS_{avg}) across all valid solutions for each programming language. These metrics highlight variations in problem difficulty, enabling the analysis of how LLMs perform on translation tasks with varying complexity. They also provide a basis for selecting or partitioning subsets of the dataset.

2) Test Cases: Although CodeNet does not include **test cases**, we retrieved them from the original platforms where the problems were published: `Aizu` and `AtCoder`. For `Aizu`, extraction was straightforward, as problem IDs matched those in CodeNet, and test cases were available via standardized URLs. For `AtCoder`, where IDs did not align, we applied a reverse-matching strategy: we matched each test set to a CodeNet problem by executing it against valid solutions.

We could not find test cases for 82 problems on `Aizu` and `AtCoder`. However, each problem includes at least one test case embedded in the natural language description. Many problems also contain too few test cases for reliable evaluation, often only one. We set a configurable minimum of 7 test cases per problem to ensure meaningful assessment. For the 2,667 English-described problems, we manually extracted embedded test cases for the 82 without any and used `LLaMA 3.1 (8B)` to generate additional test cases where necessary to meet the threshold. While this introduces synthetic elements, all additions were validated to minimize bias.

As our goal was to augment the dataset, not to evaluate LLMs for test case generation, we followed a simple strategy. For each problem needing more tests, we prompted `LLaMA 3.1 (8B)` with the natural language description and examples to generate inputs using three strategies: **similar cases**, **edge cases**, and **diverse variations**. We discarded invalid test cases and used three randomly selected C++ solutions to determine expected outputs (we used C++ to avoid data leakage with respect to our experimental evaluation). We retained only test cases where all three solutions produced matching outputs.

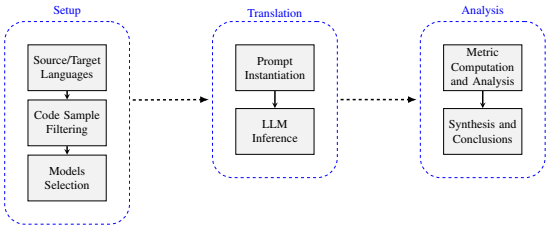

Fig. 2. Evaluation Procedure for LLM-based Code Translation

D. Prompt Templates

Prompting plays a critical role in the quality of LLM-generated translations [18]. For example, zero-shot uses only task instructions and the model's pre-trained knowledge; CoT (Chain of Thought) prompts encourage reasoning steps; and few-shot provides examples or fine-tuned guidance. `Polyglot` currently focuses on zero-shot prompting to isolate inherent translation capabilities without example-based influence, reflecting real-world scenarios where generalization is required (as mentioned, the framework includes guidelines for extending to other prompting strategies). Each prompt is built from a template and instantiated with the relevant source code and task details during execution (see `PolyglotWeb.site` for the prompt templates):

1) Standard Zero-Shot (S-ZS): provides direct instructions without examples or additional reasoning.

2) Curated Zero-Shot (C-ZS): we followed an iterative, instruction-oriented design process to build the `C-ZS` prompt template. Starting from the minimal `S-ZS` baseline, we incrementally added elements to clarify the task, constrain output format, and emphasize functional equivalence. Pilot evaluations and best practices in LLM prompt design guided this process. The resulting instruction provides structured, language-agnostic guidance that balances specificity and generality.

3) Chain-of-Thought (CoT): encourages the model to explain its reasoning before generating the translated code, but still in a zero-shot manner regarding external examples. Note that CoT in `Polyglot` is implemented in a *zero-shot* fashion: models are instructed to reason step by step before producing the final code, but without in-context demonstrations or examples. The distinction therefore lies in the *verbosity and reasoning style* encouraged by the prompt, not in the use of extended context.

E. Procedure

A procedure ensures a rigorous and reproducible evaluation. `Polyglot`'s execution process consists of three phases as depicted in Figure 2 and detailed next.

1) Setup: Involves defining the experimental setup, including the selection of languages, code samples, and LLMs:

1.1. *Source and Target Languages*: Chosen based on research objectives, accounting for syntactic and semantic differences, such as typing discipline, memory management, and execution model.

1.2. *Code Sample Filtering*: When complete evaluation is impractical, one may select a subset of samples based on problem- or solution-level characteristics. The selection criteria are flexible and context-dependent, e.g., filtering by average cyclomatic complexity or specific code constructs. In our evaluation (Section IV-B), we apply two-level stratified sampling based on selection scores.

1.3. *Model Selection*: Identify the LLMs to evaluate, along with configuration options such as decoding strategies (e.g., temperature), enabling exploration of translation behavior and the impact of determinism on code quality.

2) Translation: This step involves inserting the program's source code into the selected prompt templates (e.g., `S-ZS` or `C-ZS`), producing a ready-to-use prompt for LLM inference:

2.1. *Prompt Instantiation*: Code is integrated into each prompt, reflecting the different strategies (Section III-D).

2.2. *LLM Inference*: The selected LLM(s) generate translations for the given programs based on the constructed prompts and the selected decoding strategies.

3) Analysis: Consists of interpreting and analyzing the results to derive insights about LLM performance:

3.1. *Metric Computation and Analysis*: Compute the evaluation metrics (see Section III-B) and analyze the results across models, prompts, and code characteristics.

3.2. *Synthesis and Conclusions*: Conclude from the results, identifying trends, strengths, and limitations, informing the broader understanding of LLM performance.

F. Validation & Evolution

1) Properties Analysis: `Polyglot` is grounded in established benchmarking principles, embedding key properties directly into its design to ensure robust and fair evaluation.

To ensure *representativeness*, it targets translation tasks and code samples that, although simple, reflect real-world scenarios, spanning diverse problem complexities and language constructs. Although LLM outputs are not deterministic, `Polyglot`'s evaluation is *repeatable* through deterministic prompt generation, controlled runtime environments, and a standardized execution process. *Nonintrusiveness* is maintained by evaluating LLMs as-is, without modification, and using prompts that mirror realistic usage rather than synthetic input structures. The framework is *scalable* via automation and modularity, supporting large-scale experiments across languages and models. It promotes *portability* by avoiding model- or language-specific assumptions, and ensures *simplicity* through an automated, streamlined process.

2) Extensibility Guidelines: A core strength of `Polyglot` is its flexibility and adaptability without compromising the core benchmarking principles. Adding new **metrics**, such as those targeting maintainability or security, can be done by defining appropriate scoring functions. We can incorporate additional **code datasets** by computing size, complexity, and construct metrics using existing scripts. When test cases are missing, we can use LLMs to generate them (see Section III-C2), expanding dataset coverage and improving representativeness. Customizable templates enable the integration of new **prompting strategies** that incorporate additional context, examples, or reasoning. This approach maintains the framework's *nonintrusiveness* while enabling structured analysis of prompting effects. The framework can also support alternative **evaluation modes**, such as *multi-translation* and *in-context learning*, allowing researchers to assess output diversity and augment prompts with examples.

While adding new models and metrics may seem straightforward, the novelty of `Polyglot` lies in standardizing the entire evaluation pipeline (from workload integration to metric analysis and public result dissemination). This goes beyond ad hoc setups by providing a reproducible, extensible, and modular framework aligned with benchmarking principles.

Extending to additional languages depends mainly on dataset coverage. If the source language is already in CodeNet, experiments require only configuring and executing runs within the existing infrastructure. Otherwise, curating or integrating a parallel dataset with code samples, functional test cases, and metadata (e.g., SLoC, cyclomatic complexity, CFS) is necessary. Adding a new target language is simpler, typically involving adaptations to prompt templates across strategies to capture syntax, conventions, and I/O patterns. The extent of modification varies by prompt type, from simple formatting (`S-ZS`) to more structured refinements (`C-ZS`, `CoT`).

The benchmark also supports reverse translation with only minor adjustments. Since `Polyglot` is modular and language-agnostic, this mainly requires selecting a different source language from CodeNet and adapting its prompt templates. The evaluation pipeline remains fully applicable, as test cases are tied to problem specifications rather than implementation language.

`Polyglot` is written in Python and deployed using Docker containers. Its modular architecture simplifies extensions. `PolyglotWeb.site` is a companion platform for publishing results and comparisons, encouraging broader adoption.

IV. Experimental Setup

This section discusses the source and target languages, code sample filtering, and model selection.

A. Source and Target Languages

To demonstrate `Polyglot`, we evaluate **C-to-Python, C-to-Java, and C-to-Rust translations**, enabling a controlled comparative analysis across three diverse target languages. We chose these target languages for their structural diversity and relevance to real-world software modernization. They represent distinct paradigms, from dynamic to static typing, procedural to object-oriented, and manual to automatic memory management, capturing critical translation challenges. C's low-level features require careful adaptation to higher-level abstractions in the target languages. Note that the underlying CodeNet dataset includes implementations in many other languages. Extending to new source or target languages requires only configuring and executing additional experiments within the existing infrastructure.

TABLE I
C PROBLEMS AND SOLUTIONS USED IN THE EVALUATION

Selection		Value		
Problems (per complexity / total):				43 / 129
Solutions (per problem / per complexity / total):				6 / 300 / 900
		Avg	**Min**	**Max**
Simple	SLoC	18.1	3	264
	CC	5.7	1	86
Moderate	SLoC	46.0	7	400
	CC	13.7	2	191
Complex	SLoC	75.6	14	618
	CC	25.9	4	268

B. Code Sample Filtering

Due to practical constraints, including runtime limitations and computational resources, we were unable to evaluate all problems in the workload. Instead, we applied a principled selection process to construct a representative and diverse subset of problems and solutions. We will continuously publish more results on `PolyglotWeb.site`.

We employed a two-level sampling strategy to capture variation in problem complexity to ensure meaningful and context-aware evaluation. At the first level, we define a **Problem Selection Score (PSS)** to rank problems based on average code size, cyclomatic complexity, and construct richness, considering the solutions in the programming language at hand (C in our experimental evaluation). For each problem p, the score is computed as:

$$PSS_p = \frac{\bar{CS}_p - \mu_{CS}}{\sigma_{CS}} + \frac{\bar{CC}_p - \mu_{CC}}{\sigma_{CC}} + \frac{\bar{CFS}_p - \mu_{CFS}}{\sigma_{CFS}}$$

where $\bar{CS}_p, \bar{CC}_p, \bar{CFS}_p$ represent the average code size, cyclomatic complexity, and construct feature score (*CFS*) for problem p, respectively, and μ and σ denote the global mean and standard deviation for each metric. In our experiments, we assign equal weight to all components to maintain a neutral ranking, though we could adjust the weights to prioritize specific dimensions. Using PSS_p, we stratify problems into three complexity tiers (*simple*, *moderate*, and *complex*) based on percentile thresholds, and sample uniformly from each group. This ensures balanced coverage across the complexities.

For each selected problem, we compute a **Solution Selection Score (SSS)** to select a diverse subset of implementations, which is especially important for problems with many available solutions. The SSS quantifies how much each solution deviates from the problem-level averages:

$$SSS_s = \frac{|CS_s - \bar{CS}_p|}{\sigma^p_{CS}} + \frac{|CC_s - \bar{CC}_p|}{\sigma^p_{CC}} + \frac{|CFS_s - \bar{CFS}_p|}{\sigma^p_{CFS}}$$

where CS_s, CC_s, CFS_s denote the code size, cyclomatic complexity, and construct feature score of solution s; and $\bar{CS}_p, \bar{CC}_p, \bar{CFS}_p$, along with their corresponding standard deviations $\sigma^p_{CS}, \sigma^p_{CC}, \sigma^p_{CFS}$, are computed over all valid solutions for problem p. As with problem selection, solutions are stratified into three percentile-based groups and sampled uniformly across strata to ensure structural diversity.

In our experiments, we tested 150 problems, and for each, we considered six solutions of varying complexity (Table I). This corresponds to more than 56700 translations (150 problems $\times$ 6 solutions $\times$ 3 languages $\times$ 7 LLMs $\times$ 3 prompts).

C. Model Selection

Table II lists the LLMs evaluated. They span a diverse range of parameter counts (7B to 70B), training regimes (general-purpose vs. code-specialized), and sources (e.g., Meta, Alibaba, DeepSeek). This diversity enables a comparative analysis of how scale, specialization, and design affect multilingual code translation performance.

To ensure transparency, accessibility, and reproducibility, we focus on open-source LLMs and run all models using the `Ollama` framework with **default configurations**. We used the `ollama-python` library to interact with the models. This setup eliminates external variability, such as API instability or vendor-specific sampling behavior, ensuring consistent execution. While `Polyglot` supports proprietary models, our open-source focus provides a robust, reproducible foundation for analyzing performance across languages, prompt styles, and problem complexities, enabling low-overhead extensions.

V. RESULTS

This section discusses *outcome-based* and *static metrics*. Raw data and detailed results are available at `PolyglotWeb.site`. While *contextual metrics* offer a complementary view, they are omitted due to space constraints. However, these metrics reinforce the same trends observed. Examples of failure are presented at the end of the section.

A. Outcome-Based Metrics

1) Language vs Model: Table III summarizes the outcomes across target languages and models. The results reveal substantial variation in success and failure rates, shaped by model specialization, scale, and language characteristics.

Java: `qwen2.5-coder_32b` leads with a pass rate of 61.77%, followed by `qwen2.5_32b` (53.36%) and `deepseek-coder-v2_16b` (51.46%), `deepseek-coder_33b` (51.21%) and `llama3.1_32b` (50.24%). `llama3.1_8b` performs the worst, with 36.78% of outputs failing to compile and only 21.91% passing all tests. These results highlight the challenges that smaller, general-purpose models encounter due to Java's verbosity and type strictness.

Python: Compilation is rarely an issue, but runtime failures dominate. Although Python is not compiled, we simulate this step using its `compile()` function to detect syntax errors. `qwen2.5-coder_32b` leads with a 49.43% pass rate,

TABLE II
LLMS USED IN THE EVALUATION

Model	Size	Training Data	Source	Alias in the Text
LLaMA 3.1	8B	general (mixed)	Meta	llama3.1_8b
	70B	general (mixed)	Meta	llama3.1_70b
Qwen2.5	32B	general (mixed)	Alibaba Cloud	qwen2.5_32b
Qwen2.5-Coder	7B	code-specialized	Alibaba Cloud	qwen2.5-coder_7b
	32B	code-specialized	Alibaba Cloud	qwen2.5-coder_32b
DeepSeek-Coder	33B	code-specialized	DeepSeek-AI	deepseek-coder_33b
DeepSeek-Coder V2	16B	code-specialized	DeepSeek-AI	deepseek-coder-v2_16b

TABLE III
LLM TRANSLATION OUTCOMES BY LANGUAGE

Lang.	LLM	Fail Comp.	Fail Run.	Fail Tests	Pass Tests
Java	llama3.1_8b	36.78%	14.31%	26.99%	21.91%
	llama3.1_70b	17.02%	11.72%	21.02%	50.24%
	qwen2.5_32b	19.50%	11.12%	16.02%	53.36%
	qwen2.5-coder_7b	27.18%	12.27%	16.43%	44.12%
	qwen2.5-coder_32b	11.75%	10.90%	15.57%	61.77%
	deepseek-coder_33b	15.39%	13.83%	19.58%	51.21%
	deepseek-coder-v2_16b	21.13%	11.23%	16.17%	51.46%
Python	llama3.1_8b	5.78%	56.06%	17.98%	20.17%
	llama3.1_70b	2.82%	41.19%	17.54%	38.45%
	qwen2.5_32b	1.11%	39.75%	12.01%	47.13%
	qwen2.5-coder_7b	3.00%	54.76%	13.13%	29.11%
	qwen2.5-coder_32b	1.08%	35.93%	13.57%	49.43%
	deepseek-coder_33b	3.89%	43.79%	16.54%	35.78%
	deepseek-coder-v2_16b	4.63%	42.86%	14.28%	38.23%
Rust	llama3.1_8b	80.13%	9.75%	4.23%	5.90%
	llama3.1_70b	59.07%	16.57%	8.68%	15.68%
	qwen2.5_32b	51.35%	18.87%	10.72%	19.06%
	qwen2.5-coder_7b	67.33%	12.53%	8.97%	11.16%
	qwen2.5-coder_32b	35.22%	17.09%	15.39%	32.30%
	deepseek-coder_33b	74.45%	11.42%	5.04%	9.08%
	deepseek-coder-v2_16b	58.29%	15.31%	8.82%	17.58%

followed by `qwen2.5_32b` (47.13%). `deepseek-coder-v2` and `deepseek-coder_33b` follow at 38.23% and 35.78%, while `llama3.1` lags at 20.17%. These trends show LLMs struggle with Python's dynamic semantics, even when they handle syntax correctly.

Rust: Rust remains the most difficult target, with the highest compilation failure rates. `llama3.1` and `deepseek-coder_33b` exceed 70% in failed compilations. Only `qwen2.5-coder_32b` delivers a relatively acceptable pass rate (32.30%), clearly surpassing the next best model (`qwen2.5_32b` at 19.06%). These findings show the difficulty LLMs face with Rust's strict type and memory safety constraints. Sparse training data and Rust's niche usage may also contribute to the observed low performance.

Overall: Code-specialized models with larger capacities (`qwen2.5-coder_32b`, `deepseek-coder-v2_16b`) outperform general-purpose ones. On average, they achieve a *Pass Tests* rate above 40%, versus ≈25% for models like `llama3.1_8b` and `llama3.1_70b`. `qwen2.5-coder_32b` stands out for its strong, balanced results, highlighting the benefits of scale and task-specific training. Rust remains the hardest target due to strict compile-time checks, with *Fail Comp.* rates, in general, above 50%. Python is dominated by runtime failures, reflecting challenges with dynamic execution. Java presents mixed difficulty, with moderate compilation and runtime issues.

2) Language vs Model vs Prompt: Table IV summarizes the impact of prompting strategies for the three top-performing models. Rust is excluded due to consistently low pass rates, though similar prompt-related trends were observed.

Java: `qwen2.5-coder_32b` leads overall, with `S-ZS` achieving the highest pass rate (63.96%) and `C-ZS` close behind (62.74%), while also showing the lowest runtime failure rate (10.57%). `CoT` lags at 58.62% with moderately higher failure rates. `deepseek-coder-v2` follows a similar trend: best under `S-ZS` (53.06%), slightly lower with `C-ZS` (51.39%), and further down with `CoT` (49.94%). `llama3.1_70b` stays

TABLE IV
IMPACT OF PROMPTING STRATEGIES ON TRANSLATION OUTCOMES

Lang.	LLM	Prompt	Fail Comp.	Fail Run.	Fail Tests	Pass Tests
Java	llama3.1_70b	S-ZS	14.91%	11.35%	16.35%	57.40%
		C-ZS	14.79%	11.57%	17.35%	56.28%
		CoT	21.36%	12.24%	29.37%	37.04%
	qwen2.5-coder_32b	S-ZS	10.34%	11.01%	14.68%	63.96%
		C-ZS	13.46%	10.57%	13.24%	62.74%
		CoT	11.46%	11.12%	18.80%	58.62%
	deepseek-coder-v2_16b	S-ZS	21.13%	10.79%	15.02%	53.06%
		C-ZS	22.58%	10.23%	15.80%	51.39%
		CoT	19.69%	12.68%	17.69%	49.94%
Python	llama3.1_70b	S-ZS	1.22%	41.05%	15.35%	42.38%
		C-ZS	2.45%	36.82%	13.35%	47.39%
		CoT	4.78%	45.72%	23.92%	25.58%
	qwen2.5-coder_32b	S-ZS	0.78%	36.04%	12.90%	50.28%
		C-ZS	1.78%	34.71%	12.35%	51.17%
		CoT	0.67%	37.04%	15.46%	46.83%
	deepseek-coder-v2_16b	S-ZS	4.67%	43.83%	13.68%	37.82%
		C-ZS	5.23%	36.93%	16.35%	41.49%
		CoT	4.00%	47.83%	12.79%	35.37%

competitive under both `S-ZS` (57.40%) and `C-ZS` (56.28%), outperforming DeepSeek across prompts, but drops sharply with `CoT` to 37.04%, alongside the highest test failure rate (29.37%). These results indicate that `S-ZS` and `C-ZS` provide similar gains for Java translation, while verbose `CoT` prompts hinder performance, especially for the non-specialized model.

Python: Prompting differences are most evident in Python, where runtime instability prevails. `qwen2.5-coder_32b` excels under `C-ZS` (51.17%), remains strong with `S-ZS` (50.28%), and declines slightly under `CoT` (46.83%). `deepseek-coder-v2` follows this pattern, with the highest performance under `C-ZS` (41.49%) and the lowest under `CoT` (35.37%). `llama3.1_70b` again struggles most with `CoT`, dropping to 25.58% pass rate and showing peak runtime (45.72%) and test failure (23.92%) rates. Unlike in Java, where `S-ZS` often leads, Python results slightly favor `C-ZS`, suggesting that explicit instructions help manage the language's runtime unpredictability.

Overall: `S-ZS` and `C-ZS` outperform `CoT` across all metrics (compilation, runtime, and test success). In contrast, general-purpose models such as `llama3.1_70b` show high sensitivity to verbose prompting. Its pass rate drops from 57.40% (`S-ZS`) to 37.04% (`CoT`) in Java, and from 42.38% to 25.58% in Python. These results suggest that `CoT` introduces verbosity without the structured inductive bias needed for program synthesis. Prompt engineering thus acts as a critical tuning layer that complements model architecture and training.

3) Language vs Model vs Problem Complexity: To understand how model performance varies with problem complexity, problems are categorized into *simple*, *moderate*, and *complex* tiers using the **PSS** metric (Section IV-B). We focus on `llama3.1_70b`, `qwen2.5-coder_32b`, and `deepseek-coder-v2`, excluding Rust due to its consistently low pass rates. To isolate the impact of complexity, we limit this analysis to `S-ZS`, which previously showed strong performance (though the trends hold across prompt types). As shown in Table V, translation success declines consistently with increasing problem complexity.

Java: `qwen2.5-coder_32b` is the most reliable, with pass

TABLE V
IMPACT OF PROBLEM COMPLEXITY UNDER S-ZS PROMPTING

Lang.	LLM	Complex.	Fail	Comp. Fail	Run. Fail	Test Pass	Test
Java	llama3.1_70b	Simple	4.01%	7.69%	7.69%	80.60%	
		Moderate	12.24%	9.86%	22.11%	55.78%	
		Complex	28.10%	16.34%	19.28%	36.27%	
	qwen2.5-coder_32b	Simple	2.34%	6.35%	10.03%	81.27%	
		Moderate	10.54%	9.18%	14.29%	65.99%	
		Complex	17.97%	17.32%	19.61%	45.10%	
	deepseek-coder-v2_16b	Simple	6.02%	8.03%	9.70%	76.25%	
		Moderate	19.05%	8.84%	18.71%	53.40%	
		Complex	37.91%	15.36%	16.67%	30.07%	
Python	llama3.1_70b	Simple	0.33%	23.08%	10.03%	66.56%	
		Moderate	0.68%	44.22%	18.71%	36.39%	
		Complex	2.61%	55.56%	17.32%	24.51%	
	qwen2.5-coder_32b	Simple	0.00%	25.42%	4.35%	70.23%	
		Moderate	1.02%	33.33%	14.97%	50.68%	
		Complex	1.31%	49.02%	19.28%	30.39%	
	deepseek-coder-v2_16b	Simple	1.34%	31.44%	5.69%	61.54%	
		Moderate	3.40%	43.54%	19.73%	33.33%	
		Complex	9.15%	56.21%	15.69%	18.95%	

rates of 81.27% (simple), 65.99% (moderate), and 45.10% (complex), and moderate rises in compilation and runtime failures. `deepseek-coder-v2` degrades more steeply (from 76.25% to 30.07%) driven by an increase in compilation failures (6.02% to 37.91%). `llama3.1_70b` starts strong at 80.60% but drops to 55.78% and 36.27%, following similar trends and highlighting challenges in complex scenarios.

Python: Both `qwen2.5-coder_32b` and `llama3.1_70b` handle simple Python tasks well (70.23% and 66.56%, respectively), but performance deteriorates on complex inputs. `qwen2.5-coder_32b` drops to 50.68% (moderate) and 30.39% (complex), while `deepseek-coder-v2` falls to 33.33% and 18.95%. The main cause of degradation is rising runtime failures, e.g., `deepseek-coder-v2` from 31.44% (simple) to 56.21% (complex), indicating challenges with execution and control flow in dynamically typed code.

Overall: While code-focused LLMs excel at simple tasks, reliability declines as problem complexity increases. On average, *Pass Tests* rates drop from ≈72% (simple) to ≈31% (complex), with failure types reflecting language-specific challenges (compilation issues in Java and runtime errors in Python). These trends highlight the need to evaluate LLMs across difficulty levels and to develop prompts and models that generalize better to complex scenarios.

B. Static Metrics

We now analyze the structural transformations introduced by LLM-generated code across languages, models, and prompting strategies (Tables VI and VII). This analysis helps reveal how LLMs alter the underlying structure of source code and how such changes relate to functional correctness and maintainability. While we omit complexity-specific breakdowns due to space constraints, the observed trends are consistent, with detailed results available at `PolyglotWeb.site`.

Java: All LLMs show minimal structural change under all prompts (in general, small average decreases in complexity and small average increases in size). While rare outliers do occur (such as `llama3.1_70b` reaching +12 in CC under `CoT`, and `deepseek-coder-v2` hitting +22) there

TABLE VI
CYCLOMATIC COMPLEXITY (ΔCC) VARIATION

Lang.	LLM	S-ZS $\Delta CC_{\log}$	S-ZS Min	S-ZS Max	C-ZS $\Delta CC_{\log}$	C-ZS Min	C-ZS Max	CoT $\Delta CC_{\log}$	CoT Min	CoT Max
Java	llamA3.1_70b	-0.04	-173	5	-0.06	-90	10	-0.09	-92	12
	qwen2.5-coder_32b	-0.04	-89	11	-0.07	-184	12	-0.02	-105	14
	deepseek-coder-v2_16b	-0.05	-59	5	-0.11	-92	11	-0.05	-44	22
Python	llamA3.1_70b	0.17	-37	25	0.22	-56	38	0.59	-57	35
	qwen2.5-coder_32b	0.24	-102	47	0.17	-102	33	0.43	-101	57
	deepseek-coder-v2_16b	0.25	-179	29	0.10	-91	17	0.41	-91	19
Rust	llamA3.1_70b	-0.09	-8	5	-0.13	-97	26	-0.08	-78	6
	qwen2.5-coder_32b	-0.12	-106	10	-0.18	-105	53	0.00	-66	12
	deepseek-coder-v2_16b	-0.08	-55	7	-0.24	-92	5	0.03	-73	13

are also notable reductions. For instance, `llama3.1_70b` and `qwen2.5-coder_32b` reduce SLoC by up to 369 lines, demonstrating abstraction capabilities in boilerplate-heavy code. However, SLoC spikes such as `llama3.1_70b`'s +210 under `C-ZS` and `qwen2.5-coder_32b`'s +317 under `CoT` suggest occasional loss of structural control.

Python: Python translations generally increase cyclomatic complexity while reducing SLoC, particularly under `CoT`. `llama3.1_70b` peaks at $\Delta CC_{\log} = 0.59$ under `CoT`, while `deepseek-coder-v2` reaches $\Delta \text{SLoC}_{\log} = -0.30$ under `S-ZS`. The SLoC range for `qwen2.5-coder_32b` spans from -331 to $+203$, suggesting significant variability in code expansion and condensation depending on prompt style and problem structure. While cyclomatic complexity tends to rise, SLoC increases remain rare and within acceptable bounds. The models' ability to reduce size, and occasionally even simplify control flow, reflects their potential for structural abstraction.

Rust: Rust exhibits the most erratic structural behavior. `qwen2.5-coder_32b` shows moderate complexity shifts ($\Delta CC_{\log} = -0.12$ under `S-ZS`) but wide SLoC variation (up to +332). `deepseek-coder-v2` follows suit with similarly large SLoC changes. These fluctuations reflect challenges in generating structurally and semantically valid Rust due to its strict typing and ownership semantics. Compilation failures align with these structural instabilities.

Overall: LLMs tend to increase cyclomatic complexity and reduce source lines in permissive languages such as Python, maintain structural stability in statically typed Java, and show erratic shifts in strict, low-level Rust. Lower structural deviation correlates with higher functional pass rates, especially in Java. Model behavior varies by language and prompt; `qwen2.5-coder_32b` and `deepseek-coder-v2` generally maintain moderate deviation but exhibit significant outliers under `CoT`, underscoring prompt-induced volatility. Structural consistency is higher under `S-ZS` and `C-ZS`, which produce more stable transformations. These findings underscore the

TABLE VII
SOURCE LINES OF CODE (ΔSLOC) VARIATION

Lang.	LLM	S-ZS $\Delta\text{SLoC}_{\log}$	S-ZS Min	S-ZS Max	C-ZS $\Delta\text{SLoC}_{\log}$	C-ZS Min	C-ZS Max	CoT $\Delta\text{SLoC}_{\log}$	CoT Min	CoT Max
Java	llamA3.1_70b	0.08	-354	127	0.07	-369	210	0.06	-369	88
	qwen2.5-coder_32b	0.06	-163	57	0.07	-354	219	0.03	-369	317
	deepseek-coder-v2_16b	0.09	-163	149	0.10	-104	148	0.08	-163	141
Python	llamA3.1_70b	-0.24	-386	99	-0.28	-386	76	-0.26	-386	144
	qwen2.5-coder_32b	-0.29	-331	80	-0.29	-330	203	-0.29	-330	125
	deepseek-coder-v2_16b	-0.30	-386	197	-0.25	-386	147	-0.29	-386	202
Rust	llamA3.1_70b	0.05	-65	23	-0.05	-379	14	0.01	-118	106
	qwen2.5-coder_32b	0.04	-94	332	0.02	-379	231	0.00	-120	69
	deepseek-coder-v2_16b	0.05	-65	142	0.05	-199	204	0.03	-63	207

importance of structure-aware evaluation and the need to align prompts with language-specific constraints.

C. Qualitative Insights

Beyond quantitative metrics, it is essential to contextualize translation outcomes with concrete examples of failure, which highlight why syntactic validity alone does not ensure functional correctness. Next, we present some example cases, grouped into compilation, runtime, and semantic/test failures.

Java and Rust translations frequently failed during **compilation** due to type mismatches, undefined symbols, and method misuse. For example, `Llama3.1` in Java often produced "incompatible types" errors (e.g., long to int) and "cannot find symbol" issues. In Rust, `Qwen2.5-coder` and `DeepSeek-v2` commonly triggered borrow checker violations such as `E0499` (double mutable borrow) and `E0384` (immutable mutation), typical when translating from memory-unsafe languages.

Python translations were especially prone to semantic drift. For instance, `Qwen2.5-coder` produced **runtime errors** such as `ValueError` ("invalid literal for int()"), `UnboundLocalError`, and `IndexError`. In Java, frequent failures included `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `InputMismatchException`. Rust outputs often panicked on `unwrap(None)` or list out-of-bounds access.

A recurring source of **semantic/test** failures was misinterpreting I/O formats, particularly space-separated or multiline inputs. Models often hardcoded values or misapplied parsing logic, resulting in programs that compile and run but fail correctness checks. Another common failure involved the misuse of language-specific constructs, such as translating C loops into invalid Python list comprehensions or recursion, which sometimes led to errors like `RecursionError`.

Nine problems could not be translated by any model: three simple, two moderate, and four complex. Failures in the simple cases stemmed mainly from input handling, as the original C programs required multi-line, space-separated inputs that models misrepresented. No problem was solved in all scenarios; the best-performing one reached 300 correct translations out of 378. This simple task (CC 3–7, SLoC 9–18) was consistently solved across all prompt types only by three configurations (`deepseek-coder_33b`, `qwen2.5_32b`, `qwen2.5-coder_32b`), showing that even simple problems can defy robust handling. Conversely, the most complex successfully translated problem (CC 191) was solved in Java (`llama3.1_70b`, `qwen2.5-coder_32b`), Python (`deepseek-coder-v2`), and Rust (`qwen2.5-coder_32b`), where models abstracted utility code, reducing effective complexity to 7–18. The largest program (over 400 SLoC) was translated into all three languages by multiple model–prompt combinations, with outputs condensed to about 21 SLoC.

VI. Discussion

Here are the key observations from our results. Note that these findings are valid only within the context of our experiments (see Section IV) and are not intended to be generalized.

Impact of Target Language: Our study shows that the target language impacts translation outcomes, underscoring the importance of multilingual evaluation. Rust's strict type and memory safety lead to high compilation failure rates, often accompanied by erratic structural changes. In contrast, despite syntactic correctness, Python's dynamic semantics result in frequent runtime failures, revealing LLMs' difficulty with logical accuracy and implicit typing. Java's verbosity and strict typing cause compile and runtime issues, but its regular structure leads to more conservative and stable transformations.

Code-Specialized vs. General-Purpose Models: Code-specialized models such as `qwen2.5-coder_32b` and `deepseek-coder-v2_16b` consistently outperform general-purpose models, even those with larger parameter counts. Trained on structured, code-centric corpora and optimized for syntax-aware generation, they show good performance across languages and greater robustness to type constraints, runtime edge cases, and structural variability. In contrast, general-purpose models such as `llama3.1_70b` struggle with structurally complex translations and low-level features such as memory management and ownership in Rust.

Specialization vs. Complexity: `qwen2.5-coder_32b` shows the highest syntactic validity and performance across problem complexity tiers, still losing performance in all three languages as task complexity increases. Although having a lower performance, `deepseek-coder-v2_16b` shows a similar pattern for Java and Python, but exhibits a steeper decline in performance in Rust. As shown in Table V, its pass rate in Java drops from 76.25% (simple) to 30.07% (complex), and in Python from 61.54% to 18.95%. This indicates that while specialization improves average performance, robustness under complexity remains a challenge.

Scale vs. Specialization: The translation outcomes across Java, Python, and Rust reveal that model scale alone does not guarantee superior performance. For example, the general-purpose model `llama3.1_70b` (70B parameters) achieves a moderate pass rate of 50.24% in Java and 38.45% in Python, but struggles significantly with Rust (15.68%). In contrast, code-specialized models with fewer parameters, such as `qwen2.5-coder_32b` and `deepseek-coder-v2_16b`, outperform it in Java (61.77% and 51.46%, respectively) and maintain more robust results in Rust (32.30% and 17.58%). This suggests that architectural specialization and training on code contribute more to translation success than model size.

Limitations of General-Purpose Models: General-purpose LLMs like `llama3.1_70b` and `llama3.1_8b` show clear limitations in code translation tasks, particularly as complexity increases. These models frequently produce outputs with high compilation and runtime failure rates. For example, `llama3.1_8b` exhibits an 80.13% compilation failure rate in Rust, and only 5.90% of its translations pass all tests. Similarly, `llama3.1_70b` achieves just a 15.68% pass rate in Rust, highlighting its struggle with strict type and memory safety requirements. Even in Python, a more permissive language, general-purpose models underperform code-specialized ones.

Prompt Sensitivity and Reasoning Stability: Prompt-

ing strategies highlight differences in model stability. Code-specialized LLMs perform consistently across `S-ZS` and `C-ZS` prompts and degrade more gracefully under `CoT`. In contrast, general-purpose models are more volatile, with `CoT` often reducing compilation and test success rates. This instability is amplified in high-complexity tasks involving longer control flows. These findings support the hypothesis that training on enriched data, such as ASTs, type annotations, or compiler traces, can improve translation accuracy.

Generalizable and Complexity-Aware Translation: Our results suggest that effective multilingual code translation depends on a combination of scale, architectural priors, training alignment, and complexity sensitivity. Code-specialized models consistently outperform general-purpose ones in both average-case accuracy and robustness under difficult conditions. However, even top performers degrade with increasing complexity, exposing limitations in current LLM generalization. To address this, we advocate hybrid approaches that integrate LLMs with static analysis, compiler feedback, and symbolic reasoning to bridge the gap between fluent generation and semantic correctness.

Significance of Differences: All models were evaluated under a fixed inference configuration (temperature = 0.7) with one decoding trial per model–prompt setup, reflecting standard zero-shot usage. Prompt instantiation is deterministic, and over 56,000 translation attempts help smooth out fluctuations and support consistent trend observation. To verify robustness, we applied Wilcoxon signed-rank tests across prompt strategies and model comparisons. All yielded statistically significant results (e.g., $p < 0.01$ for pairwise prompt comparisons with `deepseek-coder-v2` in Java; $p = 0.03125$ for `qwen2.5-coder_32b` vs. `deepseek-coder-v2` under S-ZS), providing confidence that observed trends are not artifacts of stochastic variation.

VII. Threats to Validity

Internal Validity: Training data leakage remains a concern, as LLMs may memorize content seen during pretraining. However, since translations of our dataset are not publicly available, this risk is limited. While we use datasets with clear provenance, leakage cannot be fully ruled out. Additional noise may stem from inconsistencies in human-written code or the inherent stochasticity of LLM inference. To mitigate these issues, we filter out non-compiling or non-executing solutions from the CodeNet dataset. We apply standardized inference settings to reduce variability stemming from the inherent stochasticity of LLM inference. Selection bias may occur from filtering criteria (e.g., requiring multiple valid solutions), potentially reducing diversity. Broader datasets could improve generalizability. Lastly, despite relying on objective metrics, human interpretation bias remains a risk.

External Validity: Generalizing our findings across LLMs is challenging. This study focuses on open-source models, whose performance may not extend to proprietary systems (e.g., GPT-4) due to differences in architecture and training data. Reproducibility is another limitation, as LLMs can produce different outputs across runs, even with identical inputs (a result of their inherent stochasticity). While we use standardized inference settings to reduce variability, achieving full reproducibility is challenging due to the models' considerable internal complexity. Computational constraints also limit the scope of our evaluation. Although we use a representative subset from the CodeNet dataset, scaling to more problems, solutions, and models would improve robustness.

Construct Validity: While we evaluate translations using metrics for syntactic correctness (via compilation), runtime reliability (through execution and test cases), semantic preservation (by comparing outputs), and static complexity (via code metrics), these dimensions do not fully capture translation quality. For instance, successful execution does not guarantee that code is maintainable, idiomatic for the target language, or performance-optimized. LLM performance is also sensitive to prompt formulation, making it challenging to disentangle true model capability from the effectiveness of the prompt strategy. To mitigate this, we use standardized templates and controlled instantiation, e.g., fixed prompt structures with only the C code varying, though some input sensitivity remains. Future work should expand evaluation with metrics for maintainability, idiomaticity, and coding standard adherence, as well as explore prompt variants. Our framework supports these enhancements through systematic and reproducible extension guidelines.

VIII. Conclusion and Future Work

In this paper, we introduced `Polyglot`, a novel and flexible framework designed to evaluate LLM-based code translation rigorously. Instantiated using the IBM CodeNet Project, `Polyglot` measures code translation quality across four dimensions: syntactic correctness, execution reliability, semantic preservation, and static code metrics. We evaluated seven open-source models, including both general-purpose and code-specialized LLMs, on the task of translating C into Python, Java, and Rust.

Our evaluation shows that LLMs handle simpler translation tasks reliably, with `qwen2.5-coder_32b` achieving more than 80% semantic correctness for C-to-Java conversions of basic functions. However, performance deteriorates on structurally complex problems, particularly when translating low-level constructs into Rust (where runtime reliability can fall below 20%). Code-specialized models demonstrate superior resilience in these scenarios compared to general-purpose models such as `llama3.1_70b`, which are prone to logic errors and structural volatility. A key finding is the effectiveness of structured prompting: `S-ZS` and `C-ZS` strategies outperform `CoT` in runtime reliability, reinforcing the benefit of concise, instruction-based guidance.

Future work will expand support to additional language pairs (e.g., C++ to Go) and include proprietary models (e.g., GPT-5),. We will extend metrics to capture maintainability and idiomaticity, and explore hybrid solutions such as enhancing test generation, refining outputs via post-processing, and integrating LLMs with rule-based or fine-tuned components.

REFERENCES

[1] P. Larsen, " Migrating C to Rust for Memory Safety ," *IEEE Security & Privacy*, vol. 22, no. 04, pp. 22–29, Jul. 2024. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSEC.2024.3385357

[2] W. K. G. Assunção, L. Marchezan, L. Arkoh, A. Egyed, and R. Ramler, "Contemporary software modernization: Strategies, driving forces, and research opportunities," *ACM Trans. Softw. Eng. Methodol.*, Dec. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3708527

[3] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639138

[4] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 47–51. [Online]. Available: https://doi.org/10.1145/3639476.3639762

[5] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623342

[6] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. Bissyande, C. Le Goues, and S. Jin, "Morepair: Teaching llms to repair code via multi-objective fine-tuning," *ACM Trans. Softw. Eng. Methodol.*, May 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3735129

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[8] S. Wang, M. Geng, B. Lin, Z. Sun, M. Wen, Y. Liu, L. Li, T. F. Bissyandé, and X. Mao, "Natural language to code: How far are we?" in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 375–387. [Online]. Available: https://doi.org/10.1145/3611643.3616323

[9] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3672456

[10] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani, "Core: Resolving code quality issues using llms," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 789–811, 2024.

[11] Y. Zhang, "Detecting code comment inconsistencies using llm and program analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 683–685.

[12] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using llms and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[14] S. Zhu, Supryadi, S. Xu, H. Sun, L. Pan, M. Cui, J. Du, R. Jin, A. Branco, and D. Xiong, "Multilingual large language models: A systematic survey," 2024. [Online]. Available: https://arxiv.org/abs/2411.11072

[15] 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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 695–707. [Online]. Available: https://doi.org/10.1145/3611643.3616350

[16] 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," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660778

[17] Y. Luo, R. Yu, F. Zhang, L. Liang, and Y. Xiong, "Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2448–2449. [Online]. Available: https://doi.org/10.1145/3691620.3695322

[18] 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 *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639226

[19] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[20] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *Proceedings of the NeurIPS 2021 Datasets and Benchmarks Track*, 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf

[21] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3695988

[22] 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," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660778

[23] R. Li, D. Fu, C. Shi, Z. Huang, and G. Lu, "Efficient llms training and inference: An introduction," *IEEE Access*, 2024.

[24] B. Wang, A. Kolluri, I. Nikolić, T. Baluta, and P. Saxena, "User-customizable transpilation of scripting languages," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586034

[25] Y. Akinobu, M. Obara, T. Kajiura, S. Takano, M. Tamura, M. Tomioka, and K. Kuramitsu, "Is neural machine translation approach accurate enough for coding assistance?" in *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code*, ser. BCNC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 23–28. [Online]. Available: https://doi.org/10.1145/3486949.3486966

[26] E. Visser, *Program Transformation with Stratego/XT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 216–238. [Online]. Available: https://doi.org/10.1007/978-3-540-25935-0_13

[27] Immunant, "C2Rust: Migrate C code to Rust," https://github.com/immunant/c2rust, 2023, accessed: 2025-05-18.

[28] T. Zhou, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran, "Llm-driven multi-step translation from c to rust using static analysis," 2025. [Online]. Available: https://arxiv.org/abs/2503.12511

[29] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, "Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis," 2024. [Online]. Available: https://arxiv.org/abs/2412.14234

[30] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow." in *ICLR*. OpenReview.net, 2021. [Online]. Available: http://dblp.uni-trier.de/db/conf/iclr/iclr2021.html#GuoRLFT0ZDSFTDC21

[31] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[32] 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.

[33] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[34] B. Rozière, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems*, 2020.

[35] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, "Scalable, validated code translation of entire projects using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2412.08035

[36] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[37] P. Xue, L. Wu, Z. Yang, C. Wang, X. Li, Y. Zhang, J. Li, R. Jin, Y. Pei, Z. Shen, X. Lyu, and J. W. Keung, "Classeval-t: Evaluating large language models in class-level code translation," 2025. [Online]. Available: https://arxiv.org/abs/2411.06145

[38] S. e. a. Zhou, "Codebertscore: Evaluating code generation with pretrained models of code," in *EMNLP*, 2023.

[39] N. e. a. Tran, "Does bleu score work for code migration?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 165–176.

[40] T. Y. Zhuo, "ICE-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*. St. Julian's, Malta: Association for Computational Linguistics, 2024, pp. 2232–2242. [Online]. Available: https://aclanthology.org/2024.findings-eacl.148/

[41] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. [Online]. Available: https://openreview.net/forum?id=6lE4dQXaUcb

[42] I. Chaudhary *et al.*, "Code alpaca: An instruction-following code generation dataset," https://github.com/sahil280114/codealpaca, 2023, accessed: 2025-10-01.

[43] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, "Codetransocean: A comprehensive multilingual benchmark for code translation," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 4590–4605.

[44] G. Ou, M. Liu, Y. Chen, X. Peng, and Z. Zheng, "Repository-level code translation benchmark targeting rust," *arXiv preprint arXiv:2411.13990*, 2024.