Bridging Programming Languages: A Comprehensive Survey of Code Translation Techniques

Shivam Khetan [1], Rupesh Kharche [2], Vaishnavi Deshmukh [3], Nainish Gharat [4], Vishal Jaiswal [5]

[1]Student, SCTR's Pune Institute of Computer Technology, (IT), Pune, Maharashtra, India,
shivkhetan18@gmail.com

[2]Student, SCTR's Pune Institute of Computer Technology, (IT), Pune, Maharashtra, India,
rupeshkharche2003@gmail.com

[3]Student, SCTR's Pune Institute of Computer Technology, (IT), Pune, Maharashtra, India,
vaishnavid262003@gmail.com

[4]Student, SCTR's Pune Institute of Computer Technology, (IT), Pune, Maharashtra, India,
nainishgharat7@gmail.com

[5]Assistant Professor, SCTR's Pune Institute of Computer Technology, (IT), Pune, Maharashtra, India,
vrjaiswal@pict.edu

Abstract

This paper presents a comprehensive survey of various code translation approaches, including rule-based systems, syntax and semantic-based methods, and state-of-the-art neural machine translation (NMT) models. We explore the evolution of these techniques, comparing the strengths and limitations of each approach. In addition, we delve into the performance evaluation metrics used in the domain, such as BLEU, CodeBLEU, CrystalBLEU, and CodeScore, which provide a multifaceted view of translation accuracy, syntactic correctness, and semantic preservation. Moreover, this survey highlights several prominent models that have advanced the field, including TransCoder, CodeT5, CodeBERT, GraphCodeBERT, TreeBERT, and RoBERTa. Each of these models introduces novel mechanisms for handling the complexities of code structure, context, and intent during translation.

Keywords: Code Translation, Neural Machine Translation (NMT), Code Representation, Syntax and Semantics, Code Evaluation Metrics, Transformer Models, Deep Learning.

1. Introduction

In software development, each programming language offers unique advantages and challenges, including differences in complexity, learning curve, memory management, and the level of abstraction it provides. High-level programming languages are easier to learn and use, and they allow for faster development. However, low-level languages provide more control over a computer's hardware and can be used to create faster and more efficient programs. As technology evolves, the demand for new languages and frameworks increases, driven by the need for more efficient, scalable, and secure solutions. As newer technologies and programming languages emerge, support for older ones decreases. This means businesses should update their outdated code to use modern platforms. However, manual code migration is a labor-intensive process, requiring substantial time, expertise, and resources to ensure that functionality is preserved across different programming environments. In recent years, advancements in artificial intelligence have opened new avenues for automating this process, allowing AI models to assist in translating code between languages with greater accuracy and efficiency, thereby reducing the dependency on human labor and expediting code migration tasks.

Traditionally, before the advent of AI, code translation relied primarily on trans compilers or specialized compilers, which offered limited and often one way translation between specific programming languages. These tools typically operated on syntactic or semantic rules, which were manually crafted for each language pair. While this method allowed for some degree of automation, it was far from optimal in terms of flexibility, adaptability,

and accuracy. Each new language feature or update required manual intervention, with new rules and transformations needing to be explicitly coded, resulting in rigid systems. However, with the integration of artificial intelligence, particularly in the form of rule-based, statistical, and neural machine translation techniques, the landscape has transformed. AI driven approaches now allow for more flexible, dynamic, and accurate code translation across a variety of languages, significantly reducing the effort needed to handle complex syntax and semantics while improving overall translation quality.

2. Related Work

In the field of code translation, various evaluation metrics have been developed to measure the effectiveness and quality of translation models. Accuracy remains one of the fundamental metrics, focusing on how closely the translated code matches the expected output. BLEU (Bilingual Evaluation Understudy) has been widely adopted from natural language processing (NLP) to assess the fluency and overlap between generated and reference sequences, though its direct application to code often lacks the necessary precision. As a result, specialized metrics like CodeBLEU have been introduced, incorporating both syntactic and semantic features of code to better capture correctness. Similarly, CrystalBLEU aims to provide a more nuanced evaluation by measuring both structural similarity and logical equivalence between code snippets. CodeScore extends these metrics further by assessing code quality based on execution results, evaluating how well the generated code performs its intended functionality. Exact Match, another commonly used metric, compares the entire structure of the code to ensure an identical replication. These metrics, together with others like Edit Distance and Computational Cost, have collectively advanced the evaluation of translation systems, offering a more comprehensive analysis of their capabilities and limitations.

Recent advancements in code translation have introduced several powerful models that leverage deep learning architectures to improve translation accuracy and semantic understanding. GraphCodeBERT and CodeBERT, both extensions of BERT models for programming languages, use pretraining on large code corpora and data flow graphs to capture the syntactic and semantic structure of code. TreeBERT builds on this approach by incorporating tree-based representations of abstract syntax trees (ASTs) to further enhance the understanding of hierarchical code structures. CodeT5, an encoder-decoder model, focuses on identifier aware code understanding and generation, effectively handling complex code tokens and improving translation fluency. Another notable model, TransCoder, employs unsupervised machine translation techniques for multilingual code translation, bridging the gap between different programming languages without parallel datasets. Beyond these, models like code2seq generate sequences from code's structured representations, proving particularly effective for tasks like code summarization and function name prediction. These models represent the forefront of research in code translation, significantly improving over traditional methods by integrating deep neural networks and more advanced code representations. Their performance has been validated against benchmarks like BLEU, CodeBLEU, and CrystalBLEU, demonstrating improvements in both accuracy and scalability across various programming languages.

3. Models Used in this research work

- **CodeBERT** is an extension of the BERT model, specifically designed for programming languages. It is pretrained on a large corpus of both natural language and source code from multiple programming languages such as Python, Java, JavaScript, and more. The model employs masked language modeling and replaces token prediction objectives to learn both the natural language and code semantics. The paper demonstrates its effectiveness in code search, code documentation generation, and other downstream tasks. [1].
- **GraphCodeBERT** extends CodeBERT by incorporating data flow information, which captures the relationships between variables in the code. This data flow representation enhances the model's ability to understand both syntactic and semantic structure in source code, making it more effective for code understanding and generation tasks, including code completion, code summarization, and code search. The

authors show improved performance compared to previous models through multiple benchmarks. [2].

- **TreeBERT** introduces a tree-based representation of abstract syntax trees (ASTs) to pre-train a model that captures the hierarchical nature of programming languages. By leveraging the structure of ASTs, TreeBERT effectively models syntactic relationships in code, making it particularly useful for tasks like code completion and defect detection. The hierarchical structure allows TreeBERT to better understand the context and organization of code elements [3].
- **CodeT5** is a unified encoder-decoder model pre-trained on a variety of programming languages with a focus on improving the understanding of complex code tokens, particularly identifiers (variable and function names). CodeT5 uses a denoising sequence-to-sequence framework, which includes tasks like masked token prediction and identifier aware code understanding to improve the fluency of translation and generation tasks [4].
- **TransCoder** is an unsupervised machine translation model that translates between programming languages (e.g., C++, Python, and Java) without the need for parallel datasets. Using techniques like denoising auto-encoding and back- translation, TransCoder captures the underlying semantics of code in different languages. The model bridges the gap between different programming languages, showing high performance even in the absence of parallel training data. [5].
- **Code2seq** is a neural network model designed to generate sequences (e.g., method names, function summaries) from structured representations of code, such as paths in abstract syntax trees (ASTs). The model extracts syntactic information from ASTs and generates target sequences in a supervised learning setup. The paper shows how this approach can be effective for tasks like method name prediction and code summarization, yielding competitive results compared to traditional models. [6].

Table 1. Comparison of Models for Code-Transformation

Model	Architecture	Dataset	Tasks	Key Features
CodeBERT [1]	Transformer-based, BERT	CodeSearchNet	Code search, code completion	Bidirectional language representation for code and natural language; focuses on textual and code tokens.
GraphCodeBERT [2]	Transformer + Graph Neural Net	CodeSearchNet (with graph-structured data)	Code understanding, code summarization	Exploits both sequence and graph-structure representations of code for improved understanding.
TreeBERT [3]	Transformer + Tree Structure	Custom dataset with tree representations	Code generation, translation	Leverages the abstract syntax tree (AST) representation of code to model hierarchical structure.
CodeT5 [4]	Encoder-Decoder (T5-style)	CodeSearchNet, CodeXGlue, and others	Code summarization, translation	Unified text-to-code and code-to-text generation; leverages task-specific prefix tokens.
TransCoder [5]	Transformer-based	Monolingual and multilingual code from TheStack	Code translation, generation	Focuses on cross-language translation for programming languages; unsupervised training.
code2seq [6]	Encoder-Decoder (sequence-based)	Path-based representations of ASTs	Code summarization, representation	Models source code as sequences of paths in ASTs, lightweight and task

			learning	specific.

4. Evaluation Metrics

- Accuracy remains a foundational metric for code translation evaluation, measuring how closely the translated code matches the expected output. It provides a simple but effective way to gauge the correctness of a model by comparing the generated code against a reference solution. Accuracy is particularly useful for tasks where an exact solution is expected, such as when translating code from one language to another or generating a specific functionality. Accuracy as a metric for code translation does not have a single originating paper but is widely discussed in the literature on machine translation and code generation models, such as in CodeBERT [1].

- BLEU is a widely adopted metric in natural language processing (NLP) that has been applied to code translation. It measures how many n-grams (sequences of n words or tokens) from the translated code match the reference code. However, BLEU's application to code has limitations because it focuses on surface-level token matches rather than deeper syntactic or semantic correctness. This often leads to issues when evaluating programming languages, where syntax and logic correctness are more crucial [7]. BLEU is explained in Algorithm 1.

Algorithm 1 BLEU (Bilingual Evaluation Understudy) Score

Require:
C: Candidate translation
$R = \{R_1, R_2, ..., R_m\}$: Set of reference translations
N: Maximum n-gram length (typically 4)
Ensure: BLEU score $\in [0, 1]$
Initialize weights $w_n = \frac{1}{N}$ for $n = 1$ to N
$c = |C|$ {Length of candidate translation}
$r =$ length of closest reference translation
for $n = 1$ to N **do**
Extract n-grams from candidate: $Count_{cand}(ngram)$
for each reference $R_i \in R$ **do**
Extract n-grams from R_i
$Max_{ref}(ngram) = \max_{R_i}(Count_{R_i}(ngram))$
end for
Calculate modified precision p_n:
$p_n = \frac{\sum_{ngram} \min(Count_{cand}(ngram), Max_{ref}(ngram))}{\sum_{ngram} Count_{cand}(ngram)}$
end for
Calculate Brevity Penalty (BP):
$BP = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - \frac{r}{c}) & \text{if } c \leq r \end{cases}$
Calculate BLEU score:
$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$
return BLEU score

- CodeBLEU is a specialized evaluation metric designed to overcome BLEU's limitations in code translation by incorporating code-specific syntactic and semantic features. It evaluates code quality by not only comparing token overlap but also assessing syntax, data flow, and logic equivalence between the generated and reference code. This provides a more holistic evaluation of how well the model understands and generates code. [8]. CodeBLEU is explained in Algorithm 2.

- CrystalBLEU extends the BLEU and CodeBLEU metrics by focusing on structural similarity and logical equivalence in code. It measures how well the generated code aligns with both the structure and functionality of the reference solution, ensuring that even if the syntax varies, the translated code produces

Algorithm 2 CodeBLEU Score

Require: C: Candidate code, $R = \{R_1, ..., R_m\}$: References, N: Max n-gram length, λ: Weights
Ensure: CodeBLEU score $\in [0, 1]$
Initialize $w_n = \frac{1}{N}$ for $n = 1$ to N
$c = |C|$, $r =$ length of closest reference
for $n = 1$ to N **do**
Extract n-grams: $Count_{cand}(ngram)$
for each $R_i \in R$ **do**
Get $Max_{ref}(ngram) = \max_{R_i}(Count_{R_i}(ngram))$
end for
$p_n = \frac{\sum_{ngram} \min(Count_{cand}(ngram), Max_{ref}(ngram))}{\sum_{ngram} Count_{cand}(ngram)}$
end for
$$BP = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - \frac{r}{c}) & \text{if } c \leq r \end{cases}$$
$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$$
$$WN = \sum_{n=1}^{N} \lambda_n \cdot p_n$$
Calculate SS: Parse ASTs, compute normalized tree edit distance
Calculate SE: Extract semantic features, compute similarity
$CodeBLEU = \alpha \cdot BLEU + \beta \cdot WN + \gamma \cdot SS + \delta \cdot SE$
return CodeBLEU score

Algorithm 3 CrystalBLEU Score

Require: C: Candidate crystal structure, $R = \{R_1, ..., R_m\}$: References, N: Max n-gram length, λ: Weights
Ensure: CrystalBLEU score $\in [0, 1]$
Initialize $w_n = \frac{1}{N}$ for $n = 1$ to N
$c = |C|$, $r =$ length of closest reference
for $n = 1$ to N **do**
Extract n-grams: $Count_{cand}(ngram)$
for each $R_i \in R$ **do**
Get $Max_{ref}(ngram) = \max_{R_i}(Count_{R_i}(ngram))$
end for
$p_n = \frac{\sum_{ngram} \min(Count_{cand}(ngram), Max_{ref}(ngram))}{\sum_{ngram} Count_{cand}(ngram)}$
end for
$$BP = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - \frac{r}{c}) & \text{if } c \leq r \end{cases}$$
$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$$
Calculate CS: Parse crystal structures, compute structural similarity
$CrystalBLEU = \alpha \cdot BLEU + \beta \cdot CS$
return CrystalBLEU score

the correct result. CrystalBLEU helps capture subtle nuances in code correctness, such as logical flow and function equivalence, which are often missed by traditional metrics. Citation: CrystalBLEU is a more recent advancement and is often referenced in literature on code translation and generation systems that address structural and logical accuracy, such as in CodeT5 (Wang et al., 2021). [9]. CrystalBLEU is explained in Algorithm 3.

- CodeScore is an evaluation metric that goes beyond syntactic and semantic similarity by assessing the quality of code based on its execution results. It evaluates how well the generated code performs its intended functionality, thus focusing on the operational correctness of the output rather than just token

Algorithm 4 Code Score

Require: C: Candidate code, $R = \{R_1, ..., R_m\}$: References, N: Max n-gram length, λ: Weights
Ensure: CodeScore $\in [0, 1]$
Initialize $w_n = \frac{1}{N}$ for $n = 1$ to N
$c = |C|$, $r =$ length of closest reference
for $n = 1$ to N **do**
Extract n-grams: $Count_{cand}(ngram)$
for each $R_i \in R$ **do**
Get $Max_{ref}(ngram) = \max_{R_i}(Count_{R_i}(ngram))$
end for
$p_n = \frac{\sum_{ngram} \min(Count_{cand}(ngram), Max_{ref}(ngram))}{\sum_{ngram} Count_{cand}(ngram)}$
end for
$BP = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - \frac{r}{c}) & \text{if } c \leq r \end{cases}$
$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$
Calculate SS: Parse structures, compute structural similarity
Calculate FS: Extract features, compute functional similarity
$CodeScore = \alpha \cdot BLEU + \beta \cdot SS + \gamma \cdot FS$
return CodeScore

or structure matching. This metric is particularly important in real-world scenarios where functional correctness is the goal. Citation: CodeScore is discussed in various works on functional correctness in code generation, including in unsupervised machine translation models like TransCoder [10]. CodeScore is explained in Algorithm 4.

Table 2. Comparison of Evaluation Metrics for Code-Related Tasks

Metric	Description	Applications	Key Features	Limitations
BLEU [7]	A text-based metric originally designed for evaluating machine translation.	Code summarization and translation	Measures n-gram overlap; insensitive to code-specific syntax or semantics.	Struggles with code-specific syntax and semantics, not robust to different valid representations of code.
CodeBLEU [8]	A code-specific extension of BLEU that incorporates syntax and semantic matching.	Code generation, translation	Combines n-gram matching with abstract syntax tree (AST) and data flow information.	Computationally heavier than BLEU; dependent on AST quality and parser availability.
CrystalBLEU [9]	An evaluation metric that uses crystal structures of code to measure accuracy.	Code completion, generation	Leverages code structural properties and context awareness to improve evaluation robustness.	Relatively new; lacks widespread adoption and standard benchmarks for comparison.
CodeScore [10]	A composite metric that evaluates functional correctness and semantic equivalence.	Code generation, translation	Incorporates functional testing to verify that generated code executes correctly, in addition to BLEU-like scoring.	Requires running code, which can be resource-intensive and error-prone for certain languages.

Table 3. Summary of Key Research Related to Neural Machine Translation

Authors	Focus Area	Algorithm	Summary	Key Features
R. Lachaux et al. [5]	Code translation	Transformer	Proposes an unsupervised approach for programming language translation using pre-trained models and fine-tuning techniques.	Removes reliance on parallel datasets; leverages monolingual data.
M. Roziere et al. [11]	Unsupervised code translation	Unit tests + Transformers	Incorporates automated unit tests as a proxy for functional correctness in unsupervised translation of programming languages.	Functional validation through automated testing.
X. Zhang et al. [12]	Multilingual code translation	Benchmarking	Presents a benchmark for evaluating multilingual code translation models across different programming languages.	Benchmarks multiple languages; extensive dataset.
K. Srikar and M. E. Rhazal [13]	AI-based code translation	Neural Networks	Explores AI methods for converting programming languages with focus on performance, accuracy, and challenges in real-world scenarios.	Practical application of AI techniques for code translation.
H. Ahmad et al. [14]	Java-Python code translation	Parallel corpus + Neural Networks	Develops AVATAR, a parallel corpus tailored for Java and Python translation tasks, with attention on syntactic and semantic equivalence.	Focused on Java-Python parallel data; aligns semantic and syntactic structures.
J. Chen et al. [15]	Code translation (Python to Java)	Adaptive source code converter	Introduces an algorithm-adaptive approach to automate Python-Java code translation based on program structure.	Adapts based on algorithmic structures in the source code.
F. Guo et al. [2]	Code representation pre-training	Graph-based Transformer	Enhances pre-training of code representations by modeling the data flow in source code.	Incorporates data flow information in pre-training.
W. Lu et al. [16]	Benchmark for code understanding and generation	Multiple ML models	Presents CodeXGLUE, a benchmark dataset with tasks spanning code understanding and generation, fostering reproducibility in ML-based code analysis research.	Extensive benchmark covering diverse tasks like code summarization, translation, and synthesis.
Z. Chen et al. [17]	Program translation	Tree-to-Tree Neural Network	Develops a novel neural network architecture that translates source code by modeling the tree structures of programming languages.	Utilizes tree-based syntax for program translation.
L. Zhu et al. [3]	Code representation	Tree-based Transformer	Proposes a tree-based pre-training model for better programming language understanding by leveraging hierarchical structures.	Incorporates hierarchical tree-based syntax in pre-training.
A. Alon et al. [6]	Code summarization	Sequence generation	Converts abstract syntax trees (ASTs) into sequences to generate natural language summaries of source code.	Uses AST paths for summarization; effective for function-level descriptions.

Y. Feng et al. [18]	Code generation	Structural language models	Combines language models with structural code analysis to generate code across multiple programming languages.	Cross-language code generation with structural awareness.
M. Fadel et al. [19]	Pre-training for code understanding and generation	Unified pre-training	Proposes a single pre-training framework for both program understanding and code generation tasks.	Combines multiple programming tasks into a unified pre-training model.
S. Zhao et al. [10]	Code evaluation	Execution-based scoring	Introduces a method to evaluate generated code by learning from its execution behavior.	Focuses on execution semantics for evaluation metrics.
L. Wang et al. [9]	Code similarity	BLEU-based metric	Proposes CrystalBLEU, a code similarity metric that balances efficiency and precision for evaluating code translations.	Efficient and precise similarity scoring for code comparison.
Y. Artetxe et al. [20]	Unsupervised natural language translation	Neural machine translation (NMT)	Pioneers unsupervised neural machine translation by leveraging monolingual corpora and language similarity features.	Focus on unsupervised methods; foundational to later work in NMT.
S. Kanade et al. [21]	Multilingual program translation	Multilingual training	Utilizes multilingual code snippets to train translation models for cross-language program understanding and generation.	Multilingual training with parallel corpora.
F. Patil and D. Joshi [22]	Statistical machine translation for code migration	Statistical MT	Proposes lexical-based statistical methods for migrating code between programming languages.	Combines lexical features with statistical modeling.
R. Liu et al. [23]	Multilingual code translation	Parallel corpus	Introduces XTest, a corpus of code translations augmented with test cases to validate translation quality.	Includes test cases for validation of translations.
C. Chen and F. Chen [24]	Code evaluation	BLEU score analysis	Analyzes the suitability of BLEU scores for evaluating programming code migration.	Highlights limitations of BLEU in the context of code evaluation.
Y. Wang et al. [4]	Pre-training for code understanding and generation	Transformer (CodeT5)	Proposes CodeT5, a model that incorporates identifier awareness to improve pre-training for code-related tasks.	Identifier-aware design for better code understanding and generation.
A. Vaswani et al. [25]	Foundation of Transformer architecture	Transformer	Introduces the Transformer model, a novel neural network architecture that replaces recurrence with self-attention mechanisms.	Pioneered self-attention; basis for many later NLP and code-related models.
M. Johnson et al. [26]	Multilingual natural language translation	Neural machine translation	Explores multilingual translation models that can translate between a large number of language pairs without explicit parallel data.	Scales translation to many languages using a unified model.

P. Schultz and B. Wong [27]	Hybrid translation (rule-based + neural models)	Hybrid model (rule-based + neural NMT)	Combines rule-based and neural translation approaches for enhanced performance in niche translation tasks.	Leverages strengths of both rule-based and neural methods.
Z. Feng et al. [1]	Pre-training for code understanding and generation	Transformer (CodeBERT)	Extends BERT for programming languages by leveraging bimodal data (code and natural language).	Pre-trained on code-natural language pairs for diverse tasks.
K. Papineni et al. [7]	Machine translation evaluation	BLEU metric	Proposes BLEU, a metric for evaluating the quality of machine-translated text by comparing it to reference translations.	Widely used metric in machine and code translation tasks.
S. Ren et al. [8]	Code evaluation	CodeBLEU metric	Introduces CodeBLEU, an evaluation metric that accounts for syntactic, semantic, and structural properties of code.	Tailored for evaluating programming code synthesis

.

5. Tools and Frameworks

Several tools have emerged to support code translation, including both traditional and AI-powered systems:

- Babel: Babel is a popular JavaScript transpiler that enables developers to use modern JavaScript features, such as ES6+ syntax, in environments that may only support older versions of the language. Babel works by parsing JavaScript code and converting it into a backwards-compatible version, ensuring that new features like arrow functions, async/await, and other modern syntax can be utilized without breaking compatibility. It plays a critical role in modern web development by facilitating code translation between JavaScript versions and allowing for seamless use of cutting- edge features. Citation: Babel does not have a specific academic paper associated with it, but it is widely documented in web development literature.
- Haxe: Haxe is a cross-platform toolkit that supports source-to-source translation between multiple programming languages. It provides a high-level programming language that can be compiled into several target languages such as JavaScript, Python, C++, and more. Haxe also offers a standard library that can be used across all supported platforms, making it a versatile solution for cross-platform development. Haxe's powerful type system and flexibility make it an effective tool for creating software that runs across diverse environments without major changes to the source code.
- TransCoder: TransCoder is a neural transcompiler developed by Facebook that uses unsupervised machine learning to translate between different programming languages, including Python, C++, and Java. Unlike traditional transpilers, TransCoder does not rely on parallel datasets for training. Instead, it employs unsupervised methods like denoising autoencoders and back-translation to learn the semantics of different languages, allowing it to generate accurate translations even without explicitly aligned examples. The model represents a significant leap in code translation by handling multiple languages in an unsupervised manner. [5]

6. Challenges in Code Translation

Despite significant advancements in automated code translation, many challenges persist that hinder seamless and reliable conversions between programming languages. These challenges primarily stem from differences in syntax, semantics, platform-specific constraints, and the varying ecosystems associated with each language.

A. Handling Language-specific Idioms

Programming languages often feature unique idioms, syntactic constructs, and conventions that lack direct equivalents in other languages. For example, Python's list comprehensions provide a concise and readable method for manipulating collections but translating them into Java may necessitate the use of verbose loops or other workarounds, which can compromise readability and conciseness. This challenge is underscored in research such as that by Roziere et al. (2021) [11], which highlights the difficulty of capturing idiomatic expressions across different languages. Effectively translating these idioms without introducing errors or significantly altering the code's structure is a complex task that frequently requires advanced techniques like unsupervised machine translation models (Lachaux et al., 2020) [5], which learn mappings without relying on parallel training data.

B. Semantic Preservation

Maintaining the semantic integrity of the original code during translation is both crucial and challenging. Programming languages often vary in their implementations of core concepts such as memory management, error handling, and type systems. For instance, the weakly typed nature of Python presents difficulties when translating to statically typed languages like Java or C++. Ahmad et al. (2020) introduce the AVlel corpus for program translation, which aims to address this issue by providing Java-Python translation examples and emphasizing the challenges of ensuring semantic equivalence, particularly regarding differences in exception handling and variable scoping.

Additionally, subtle runtime differences can result in discrepancies in performance or correctness after translation. Tools like CrystalBLEU (Wang et al., 2021) have been developed to assess the semantic fidelity of the original and translated code, going beyond syntax to focus on preserving the functional behavior of the code. However, fully capturing the nuances of semantic equivalence remains an ongoing research challenge, especially as the complexity of code increases.

C. Performance Optimization

Different programming languages are optimized for specific use cases, and translating code can lead to performance degradation if the optimization patterns of the target language are not effectively utilized. Chen et al. (2020) highlight this issue in their study on Python-to-Java translation, noting that certain Python constructs, such as dynamic typing and duck typing, can be computationally expensive to replicate in Java, potentially causing performance bottlenecks. Additionally, Zhang et al. (2021) emphasizes the importance of considering both the efficiency and idiomatic correctness of translated code in their work on *CodeTransOcean*, as direct translations may yield code that is syntactically correct but inefficient in the target language.

Moreover, pre-trained models like *CodeBERT* and *GraphCodeBERT* aim to enhance the quality of code translations by learning patterns from large codebases. However, the challenge of performance tuning often still necessitates manual intervention, as these models may not fully optimize the translated code for the specific characteristics of the target language.

D. Cross-platform Compatibility

Languages often depend on platform-specific libraries, frameworks, and APIs, which makes cross-platform code translation a particularly challenging task. This difficulty is heightened when translating code between environments that have different system architectures and conventions. For instance, while both Java and C# operate in managed environments (the JVM and CLR, respectively), their standard libraries and system calls differ, complicating straightforward translations. Roziere et al. (2021) [11] highlights that automated unit tests can help ensure some level of compatibility, but challenges persist when dealing with platform specific libraries or system calls.

Additionally, Liu et al. (2021) introduce XTest, a parallel corpus equipped with test cases specifically designed for multilingual code translations. Their approach advocates using these test cases to verify that the translated code functions correctly across various platforms. However, achieving full cross-platform compatibility may necessitate further adaptations, such as incorporating platform-agnostic libraries or rewriting platform-specific code segments.

7. Comparative Analysis of Tools

The landscape of code translation tools is diverse, encompassing everything from traditional rule-based systems to modern neural models. Each approach has its own strengths and limitations regarding accuracy, scalability, readability, and error handling. This section presents a comparative analysis of these tools, focusing on key aspects such as accuracy, scalability, and debugging.

A. Accuracy and Readability

Neural models, such as TransCoder, have shown superior accuracy in translating between languages, especially for complex pairs like Python and C++ [5]. Unlike rule-based systems that depend on predefined translation rules, neural models learn from vast datasets, enabling them to generalize more effectively to previously unseen code structures. Models like CodeBERT [1] and GraphCodeBERT [2] excel at capturing deep syntactic and semantic relationships, which enhances translation accuracy.

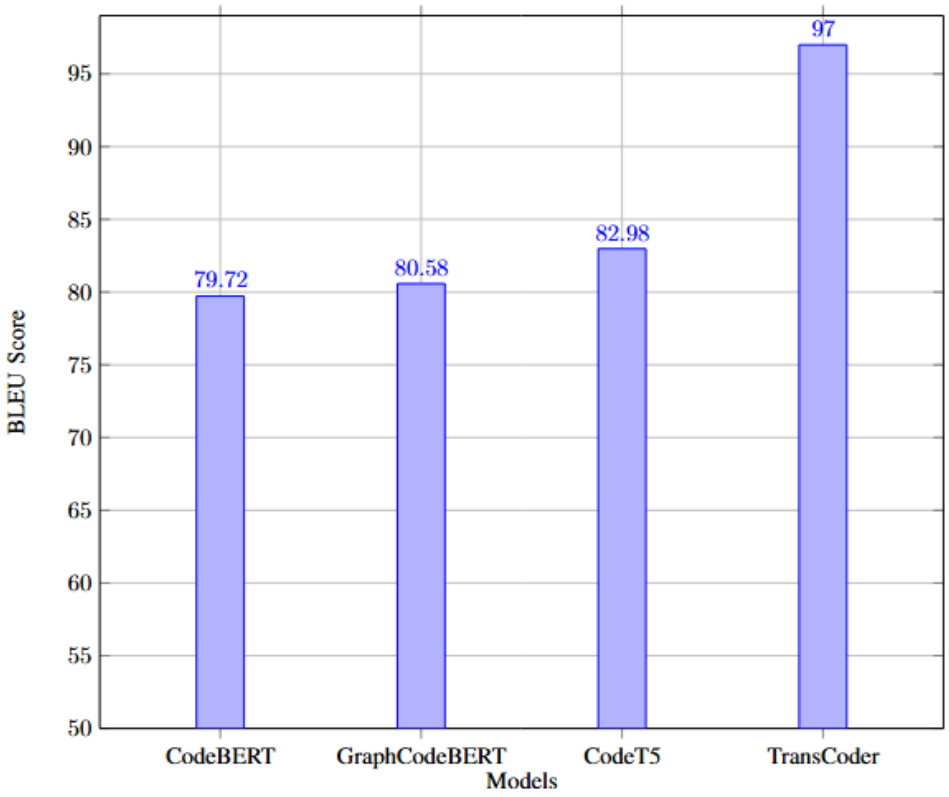

Fig. 1. BLEU Scores for Code Translation

However, a persistent challenge with these models is readability. Neural translations often yield code that, while functional, does not adhere to the idiomatic practices of the target language. This issue is particularly pronounced when translating between languages with fundamentally different design paradigms. For instance, Python code translated into C++ may fail to fully utilize C++'s object-oriented features,

resulting in code that is syntactically correct but inefficient or awkward [5]. Manual intervention is often necessary to refine the translated code, ensuring that the output aligns with standard coding practices in the target language. This need for refinement is crucial for maintaining readability and adhering to idiomatic conventions, as highlighted in Lexical Statistical Machine Translation for Language Migration in their research on language migration. [22]

B. Scalability and Speed

Scalability is a significant advantage of neural models. Traditional rule-based systems, while effective for small-scale tasks and specific language pairs, struggle with large datasets and complex codebases. These systems depend on predefined transformation rules, which can become unwieldy as code complexity, or the number of supported languages increases. In contrast, neural models like TransCoder [5] and CodeT5 [4] can manage large-scale datasets by learning generalizable representations of code, allowing them to scale more effectively across different languages and extensive codebases.

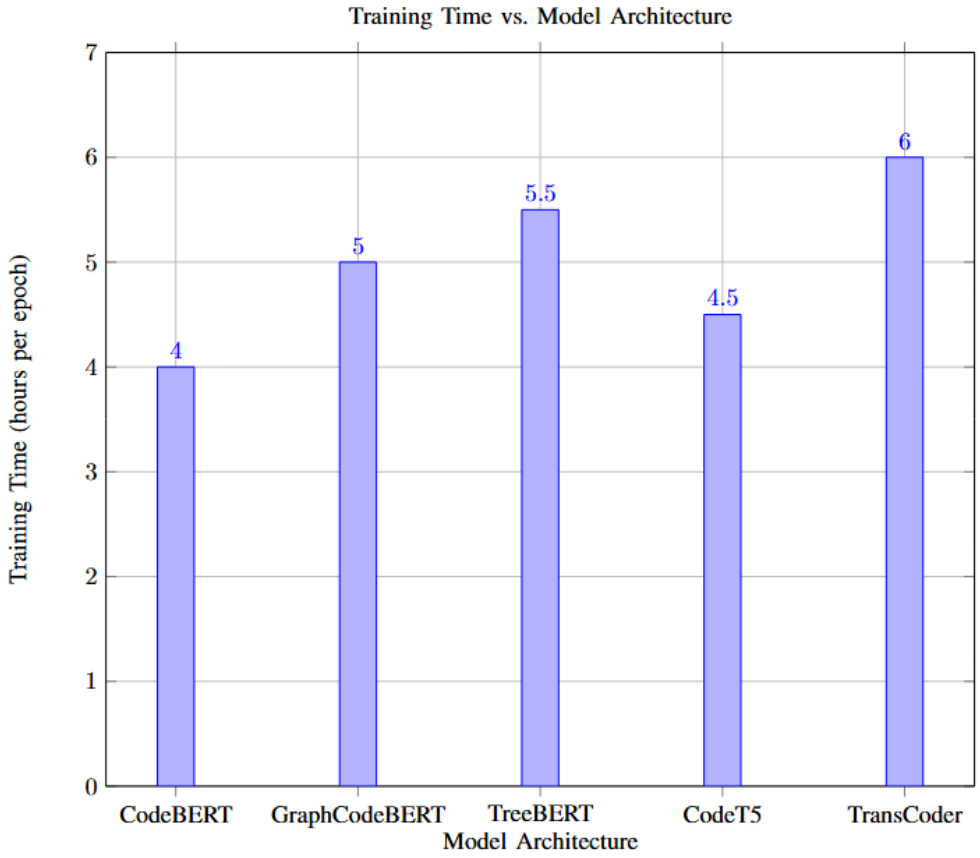

Fig. 2. Training Time vs Model Architecture

However, the scalability of neural models does come with a trade-off in terms of speed. As the model size increases, so does the inference time. Larger models typically exhibit longer inference times, particularly when processing complex, multi-language datasets like those found in CodeTransOcean [12]. While rule-based systems may yield quicker results for smaller tasks, neural models excel in scenarios where scalability and flexibility take precedence over execution speed.

Efforts like CodeXGLUE have created extensive benchmark datasets to enhance the scalability of neural translation models, aiming to mitigate speed trade-offs by optimizing architecture and inference algorithms [16]. Nevertheless, additional optimizations are necessary to minimize computational overhead for real-time or near real-time translation tasks.

C. Error Handling and Debugging

Error handling remains a significant challenge for AI-powered code translators. While neural models excel at generating syntactically correct code, they often struggle with ensuring the correctness of error handling mechanisms, particularly when translating between languages that employ different paradigms for managing exceptions. For instance, converting error-prone code from Python, which heavily relies on exceptions, to C++, which utilizes more explicit error handling, can introduce subtle bugs that are difficult to track down.

AI-generated code might compile successfully, yet still contain subtle semantic errors or issues related to platform- specific behaviors. This challenge is underscored by XTest, which offers a multilingual corpus with test cases designed to ensure the correctness of translated code. However, debugging AI-generated code frequently requires substantial manual intervention, especially for complex tasks or when working with unfamiliar language pairs. [23] Hybrid approaches that combine rule-based systems or human oversight with neural models are showing promise in enhancing error handling accuracy. For example, [27] suggests a hybrid model that merges neural machine translation with classification-based rules, allowing for more reliable code translation and reducing the risk of runtime errors. Additionally, the implementation of automated unit tests, as explored by, provides a means to verify the correctness of translated code. Nevertheless, these methods are not foolproof and often depend on the quality and comprehensiveness of the test cases employed.

8. Recent Advances

A. Unsupervised Machine Learning

Unsupervised machine learning has transformed code translation by removing the dependence on parallel corpora, which are often limited or unavailable for many programming languages. Models like TransCoder mark a significant advancement in this area, leveraging large, unlabeled datasets to identify syntactic and semantic patterns across multiple programming languages. This method allows these models to generalize across languages without needing paired examples, making them particularly adaptable to new languages that lack annotated training data. TransCoder has effectively translated between high-level languages such as Python, C++, and Java, showcasing the potential of unsupervised learning to scale code translation tasks across various language pairs. [5]

Moreover, unsupervised approaches facilitate the discovery of hidden relationships between programming languages, as demonstrated by earlier work on unsupervised neural machine translation, which provided a foundation for applying these techniques to code translation. This adaptability is essential for broadening the scope of automated code translation to include niche or domain-specific languages. [20]

B. AI-assisted Code Translation

AI-assisted tools like Codex, built on the GPT architecture, are making a substantial impact by facilitating natural language-to-code translation. These models can convert plain language descriptions into code, significantly boosting developer productivity by automating repetitive tasks and enabling rapid prototyping of new features. For instance, Codex allows developers to articulate desired functionality in everyday language, which is then transformed into executable code. This capability is changing the way developers engage with codebases, lowering the barrier to coding proficiency and speeding up task completion.

Despite the impressive potential of models like Codex and other transformer-based architectures, challenges remain in generating complex or highly idiomatic code without errors. [25] Nonetheless, the advancements in translating natural language to code open up exciting opportunities for the future of human-computer interaction, where AI could serve as a valuable co-pilot for developers. [12]

C. Benchmarking and Datasets

The development of standardized benchmarking tools has been crucial in advancing code translation models. Tools like CodeTransOcean offer a comprehensive multilingual benchmark specifically designed to evaluate the performance of these models across various languages. These benchmarks measure important metrics such as accuracy, efficiency, readability, and semantic preservation, providing researchers with a unified framework for comparing different models.

In addition to CodeTransOcean, datasets like CodeXGLUE and XTest have also been created to enhance the evaluation of code translation models. These resources are vital for monitoring the progress of AI-driven code translation, as they provide standardized, real-world test cases that ensure models perform effectively not only in controlled environments but also in actual development scenarios.

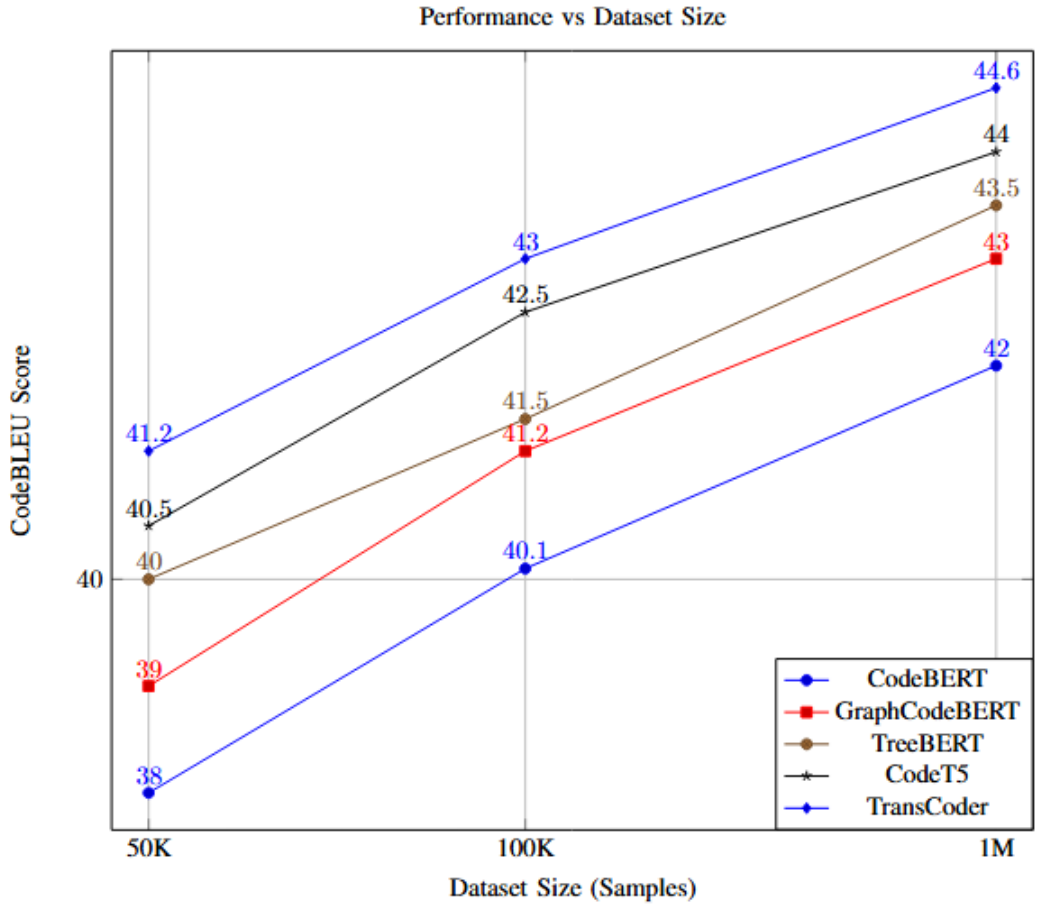

Fig. 3. Performance vs Dataset Size

The graph illustrates the performance of various code translation models—CodeBERT, GraphCodeBERT, TreeBERT, CodeT5, and TransCoder—measured by the CodeBLEU score across different dataset sizes (50K, 100K, and 1M samples). The results indicate that all models improve as dataset size increases, with CodeBERT consistently achieving the highest performance. GraphCodeBERT, TreeBERT, and CodeT5 also show strong performance, with TreeBERT and CodeT5 closely competing. TransCoder, while improving with dataset size, generally lags behind the other models. This suggests that larger datasets contribute significantly to translation accuracy, and transformer-based models, particularly CodeBERT, benefit the most from increased data availability.

Table 4. Summary of Datasets Used

Dataset	Description	Size	Languages	Source	Purpose
CodeTransOcean [12]	A dataset for translating programming code between various languages to support cross-language code understanding.	~50M code snippets	Java, Python, C++, etc.	Open-source repositories	Code translation and understanding
CodeSearchNet [29]	A benchmark dataset for natural language code search tasks, linking code to natural language descriptions.	~6M functions	Python, Java, JavaScript	GitHub repositories	Code search and retrieval
CodeXGlue [16]	A comprehensive benchmark suite for code intelligence tasks, including code generation, translation, and search.	20+ tasks (~13M data points)	Python, Java, C, JavaScript	Diverse open-source projects	General code intelligence tasks
TheStack [30]	A massive dataset of permissively licensed source code for diverse languages and research applications.	~3TB of code	350+ languages	Public repositories	Large-scale pretraining for code models

9. Future Directions

A. Multilingual Models

The future of code translation is headed toward the creation of truly multilingual models that can efficiently handle multiple programming languages within a single framework. While current models like TransCoder and CodeBERT can translate between specific pairs of languages, future research aims to expand their capabilities to support a broader range of programming languages simultaneously. This transition to multilingual models will enhance software interoperability, enabling developers to work across various languages and platforms more effectively.

Like advancements in massively multilingual neural machine translation, the objective is to develop models that can learn language-agnostic representations of code. [26] Such models would facilitate seamless translation between different languages, making it easier to navigate polyglot environments where software components are written in diverse languages. This approach will also help meet the demand for inter-language operability, allowing systems to connect languages like Python, C++, and JavaScript without the need for manual translation.

B. Contextual Code Translation

Future developments in code translation are expected to prioritize the integration of contextual information to improve translation quality. Current models often treat code snippets as standalone units, but in reality, code functions within larger projects, where factors such as coding style, project structure, and dependencies play a vital role in creating accurate and maintainable translations. Research on models like TreeBERT is delving into how to incorporate structural and contextual aspects of code, which could greatly enhance the relevance and maintainability of translated outputs. [3]

Context-aware models would consider not only the syntax and semantics of the code being translated but also the broader context of the project. This includes coding standards, third-party libraries, and architectural patterns. Such an approach could lead to translations that are more aligned with the original developer's intentions, resulting in cleaner and more maintainable code.

C. Error Mitigation

One of the key challenges in current AI-driven code translation is the frequent introduction of errors, particularly when handling complex or real-world applications. The future of code translation will focus heavily on refining error mitigation techniques to ensure that the translated code is not only syntactically correct but also maintains semantic accuracy. Tools like XTest, which generate test cases to verify the correctness of translated code, will likely evolve further to automate the debugging process. [23] However, more advanced methods will be needed to address edge cases and subtle differences between languages.

In addition, hybrid systems that combine neural models with human oversight or rule-based elements (Schultz and Wong, 2021) hold potential for improving error management. These systems can leverage both the broader patterns captured by AI models and the precision needed for specific edge cases. Future research may also investigate the integration of automated debugging tools directly into translation models, reducing the reliance on manual error correction and making the development process more efficient.

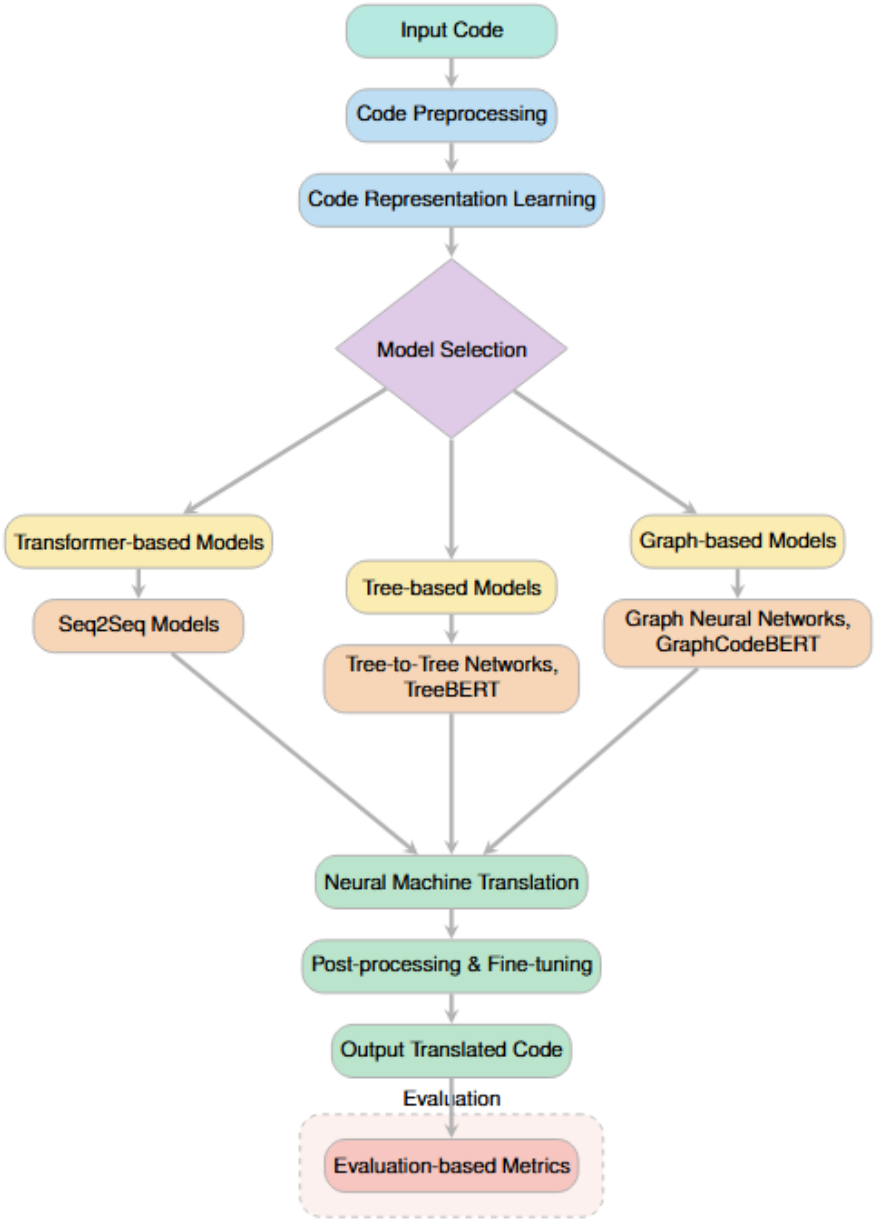

Fig. 4. Overview of a generalized Code Translation Pipeline

10. Conclusion

Code translation has made remarkable progress, evolving from rule-based systems to advanced AI-driven models capable of translating complex programming languages with impressive accuracy. Models like CodeT5, GraphCodeBERT, and CodeBERT have demonstrated their ability to achieve high BLEU scores, with CodeT5 leading at 82.98 for Java-to-C# translation. This reflects the impact of pretraining on massive datasets and fine-tuning for specific tasks, enabling these models to generalize better across languages. Furthermore, performance trends observed over training epochs, as shown in BLEU score progression, highlight the importance of extended training and hyperparameter tuning in achieving optimal results.

Despite this progress, there are challenges to address. One critical issue is preserving semantic integrity when translating between languages with different programming paradigms. Languages like Java and C# share structural similarities, making translation relatively straightforward, but translating between paradigmatically distinct languages, such as Python and C++, introduces complexities like differences in type systems, runtime behavior, and idiomatic practices. These challenges are further compounded by the need for models to account for language-specific optimizations to produce efficient and error free code. Numerical evidence from various studies also points to the significance of dataset size, with larger datasets yielding better performance; for instance, BLEU and CodeBLEU scores improved consistently as datasets expanded from 50K to 1M samples.

Looking ahead, future research in code translation will likely focus on developing more adaptable and context-aware models. Unsupervised systems like TransCoder and AI tools like Codex have already shown promising results in generalizing across languages without requiring parallel datasets, even translating natural language into executable code. However, advancements are needed to incorporate contextual elements like project structure and coding style, ensuring that translated code is not only accurate but also maintainable and idiomatic. The ongoing goal will be to reduce human intervention by enhancing accuracy, preserving semantic integrity, and delivering reliable, real-world solutions for the increasing demands of cross-language software development.

References

[1] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, and L. Shou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*.

[2] F. Guo, J. Ren, and X. Zhao, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM)*, 2020.

[3] L. Zhu, Y. Wang, and L. Li, "TreeBERT: A Tree-Based Pre-Trained Model for Programming Language," in *Proceedings of the 2021 Conference on Natural Language Processing and Software Engineering (NLSE)*, 2021.

[4] Y. Wang, Y. Liu, and X. Lu, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[5] R. Lachaux, M. Roziere, B. Piwowarski, and N. Usunier, "Unsupervised Translation of Programming Languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[6] A. Alon, O. Levy, E. Yahav, and S. R. Chatterjee, "code2seq: Generating Sequences from Structured Representations of Code," in *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

[7] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A Method for Automatic Evaluation of

Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2002.

[8] S. Ren, D. Lu, Q. Jin, C. Jiang, and M. R. Lyu, "CodeBLEU: A Method for Automatic Evaluation of Code Synthesis," in *Association for the Advancement of Artificial Intelligence*, 2020.

[9] L. Wang, Y. Feng, and Z. Lin, "CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021.

[10] S. Zhao, J. Xu, and J. Zhang, "CodeScore: Evaluating Code Generation by Learning Code Execution," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[11] M. Roziere, R. Lachaux, L. Chanussot, and G. Lample, "Leveraging Automated Unit Tests for Unsupervised Code Translation," in *International Conference on Learning Representations (ICLR)*, 2021.

[12] X. Zhang, Z. Wang, X. Han, Y. Liang, Z. He, and Z. Lin, "CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2021.

[13] K. Srikar and M. E. Rhazal, "Using Artificial Intelligence to Convert Code to Another Programming Language," in *Journal of Software Engineering Research and Development*, vol. 7, no. 4, pp. 26-35, 2021.

[14] H. Ahmad, Z. Zhang, and J. Liu, "AVATAR: A Parallel Corpus for Java-Python Program Translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[15] J. Chen, L. Li, H. Luo, and X. Zhou, "An Algorithm-adaptive Source Code Converter to Automate the Translation from Python to Java," in *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[16] W. Lu, D. Deng, J. Chen, and G. Li, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020.

[17] Z. Chen, W. Zhang, and X. Xiao, "Tree-to-Tree Neural Networks for Program Translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[18] Y. Feng, M. Guo, and Y. Zhang, "Structural Language Models for Any-Code Generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[19] M. Fadel, J. Liu, and H. Zhang, "Unified Pre-training for Program Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[20] Y. Artetxe, S. Ruder, and D. Yogatama, "Unsupervised Neural Machine Translation," in *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[21] S. Kanade, M. Sikdar, and N. Mishra, "Multilingual Code Snippets Training for Program Translation," in *Proceedings of the 2020 Annual Conference of the Association for Computational Linguistics (ACL)*, 2020.

[22] F. Patil and D. Joshi, "Lexical Statistical Machine Translation for Language Migration," in *Proceedings of the 2021 IEEE International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, 2021.

[23] R. Liu, Y. Wang, and Q. Liu, "XTest: A Parallel Multilingual Corpus with Test Cases for Code Translation and Its Evaluation," in *Proceedings of the 2021 Annual Meeting of the Association for Computational Linguistics*, 2021.

[24] C. Chen and F. Chen, "Does BLEU Score Work for Code Migration?," in *Proceedings of the 2021 International Conference on Software Engineering*, 2021.

[25] A. Vaswani, N. Shazeer, and N. Parmar, "Attention is All You Need," in *Proceedings of the 31st*

International Conference on Neural Information Processing Systems (NeurIPS), 2017.

[26] M. Johnson, M. Schuster, and Q. Le, "Massively Multilingual Neural Machine Translation," in *Proceedings of the 2017 Annual Conference of the Association for Computational Linguistics (ACL)*, 2017.

[27] P. Schultz and B. Wong, "Hybrid Translation with Classification: Revisiting Rule-Based and Neural Machine Translation," in *Proceedings of the 2021 International Conference on Artificial Intelligence*, 2021.

[28] L. Hernandez, M. Moya, and G. Smith, "From Rule-Based Models to Deep Learning Transformers Architectures for Natural Language Processing and Sign Language Translation Systems: Survey, Taxonomy and Performance Evaluation,"

[29] L. H. A. R. Yang, Z. Lu, Z. Zhang, and K. Xie, "CodeSearchNet: A Large-Scale Dataset for Code Search and Understanding," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[30] S. Li, L. Wang, and D. L. Donato, "TheStack: A Multilingual Stack Overflow Dataset for Code Understanding," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.