Department of Electrical and Computer Engineering
North South University

Directed Research
Pre-trained Transformers for Java-Python Code Translation

Sajan Kumer Sarker	ID# 2111131042
Mahzabeen Rahman Meem	ID# 2021300042

Faculty Advisor:

Dr. Mohammad Ashrafuzzaman Khan (AZK)

Associate Professor

ECE Department

Spring, 2025

APPROVAL

Sajan Kumer Sarker (ID # 2111131642) and Mahzabeen Rahman Meem (ID # 2021300042) from Electrical and Computer Engineering Department of North South University, have worked on the Directed Research Project titled "**Pre-trained Transformers for Java-Python Code Translation**" under the supervision of Dr. Mohammad Ashrafuzzaman Khan partial fulfillment of the requirement for the degree of Bachelors of Science in Engineering and has been accepted as satisfactory.

Supervisor's Signature

……………………………………

Dr. Mohammad Ashrafuzzaman Khan

Associate Professor

Department of Electrical and Computer Engineering

North South University

Dhaka, Bangladesh.

Chairman's Signature

……………………………………

Dr. Mohammad Abdul Matin

Professor & Chair

Department of Electrical and Computer Engineering

North South University

Dhaka, Bangladesh.

DECLARATION

This is to declare that this project/directed research is our original work. No part of this work has been submitted elsewhere partially or fully for the award of any other degree or diploma. All project related information will remain confidential and shall not be disclosed without the formal consent of the project supervisor. Relevant previous works presented in this report have been properly acknowledged and cited. The plagiarism policy, as stated by the supervisor, has been maintained.

Students' names & Signatures

1. Sajan Kumer Sarker

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

2. Mahzabeen Rahman Meem

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

ACKNOWLEDGEMENTS

The authors would like to express their heartfelt gratitude towards their project and research supervisor, Dr. Mohammad Ashrafuzzaman Khan, Associate Professor, Department of Electrical and Computer Engineering, North South University, Bangladesh, for his invaluable support, precise guidance and advice pertaining to the experiments, research and theoretical studies carried out during the course of the current project and also in the preparation of the current report.

Furthermore, the authors would like to thank the Department of Electrical and Computer Engineering, North South University, Bangladesh for facilitating the research. We would also like to thank my friends for helping us in this project. The authors would also like to thank their loved ones for their countless sacrifices and continual support.

ABSTRACT

Pre-trained Transformer for Java-Python Code Translation

As the software system evolves, there is an increasing need to convert code written in one programming language into another–especially when moving from older technologies to more modern or flexible ones. This study focuses on automatically translating Java code into Python code using the transformer-based encoder-decoder model. We utilized the AVATAR-TC custom research dataset, developed using the same approach as the AVATAR dataset, which contains matched pairs of Java and Python code, to train, validate, and test models such as CodeT5 and PLBART. These models are based on transformer architecture–a robust framework designed initially for natural language processing tasks, now widely used in code understanding, summarizing, or translation task. We propose a two-step approach; in the first step, raw Java code is translated into Python; in the second step, the translated Python code is refined by correcting syntactic or semantic errors and minimizing the code where possible, thus providing users with two versions to choose from. For the first step, we employed the CodeT5 Base model, and in the second step, we used the PLBART Base model, and each model was pretrained with Python code snippets. We evaluated the models using the BLEU, CodeBLEU, n-gram match, weighted n-gram match, syntax match, and dataflow match metrics. The CodeT5 model achieved a BLEU score of 51.72 and a CodeBLEU score of 0.4913, while the PLBART model achieved 28.92 BLEU, and 0.3327 CodeBLEU scores.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Background and Motivation

The rapid progress of programming languages, coupled with growing complexity in software systems, has necessitated an urgent need for tools to make it possible to port code between different programming languages [1]. As companies move to improve their technological platforms and adopt new ones, there is a pressing need to move legacy code to today's programming languages without losing any functionality and accuracy. The manual process of porting code is typically time-consuming, error-prone, and most of the time requires a deep knowledge of both target and source programming languages. The recent advancement in artificial intelligence (AI), especially in areas of Machine Learning (ML), Deep Learning (DL), and Natural Language Processing (NLP), have shown it is possible to build models to comprehend and produce human language [2]. Most importantly, these skills have been transferred to programming languages, so models are capable of interpreting code semantics and code syntax [3].

The development of transformer-based architecture, as seen in T5 (Text-to-Text Transfer Transformer) and CodeT5, is one of the current leaps in code intelligence [4] [5]. By using a self-attention mechanism, these architectures are able to model complex dependencies not just within the elements of a sequence, but also between sequences of elements, whether these are natural language phrases or symbols used in program code [2]. Through the framing of code translation as a sequence-to-sequence problem, transformer-based models have the ability to "understand" code in a source language and "generate" its equivalent form in a target language, very much like how neural machine translation is carried out in natural language processing [1]. While T5, for example, recasts a wide variety of linguistic tasks into a text-to-text format, CodeT5 extends this formulation to include source code, in the process supporting functions such as code summarization, code generation, and—of specific concern to this research—code translation between programming languages like Java and Python [5].

In this context, the project focuses on leveraging pretrained transformer models to automate and enhance the quality of Java-to-Python code translation. The core idea is to utilize a model that has already learned rich representations of code structure and semantics from large-scale code corpora [3] [6]. The model initially encodes the input java code into a latent representation, which captures its logic, grammar, and semantics. Then, based on this representation, it creates Python code that keep the original functionality while following Pythonic rules. Such a system can be train or fine-tune on parallel datasets of Java and Python code. This lets it learn idioms that are peculiar to each language, deal with changes in syntax, and handle small semantic variances [7].

The translation pipeline typically involves several key steps. First, the Java source code is tokenized and passed through the transformer encoder, which produces contextual embeddings for each token. The decoder then processes these embeddings to generate Python code token by token, using its multi-head attention to reference both the source and previously generated tokens the model learns during training by comparing its output to accurate examples and trying to get the two as close as possible [2]. This process frequently employs a technique known as teacher forcing, wherein the model is directed by the real reference outputs, with a kind of error measurement referred to as cross-entropy loss. When it's time for the model to generate on its own, techniques like beam search can help it reduce more fluent and accurate results, especially for tasks like generating code [8].

By leveraging the power of pretrained transformer models, this project aims to reduce manual effort, minimize the risk of introducing errors during the migration process, and make code modernization more accessible for organizations of all sizes. Ultimately, this approach holds promise not only for Java to Python translation but also as a general framework for multilingual code translation and automation in software engineering [2].

1.2 Purpose and Goal of the Project

The primary objective of this project is to develop a two-stage transformer-based solution to translate java to python code. This comes in the wake of the increase in demand of automated, trusted and efficient migration tools in coding software development. This is in response to the rising need for automated, reliable, and efficient code migration tools in modern software development. As organizations seek to modernize their technology stacks, the ability to move legacy Java code to Python, one of the most popular and versatile programming languages, can reduce redevelopment costs, lower human error, and speed u absorbing new platforms.

The research uses cutting-edge pretrained models like CodeT5 and PLBART, which have shown outstanding performance in jobs involving both code and natural language, to do this [5] [3]. The research intends to allow the models to learn the underlying syntactic structures, semantic subtleties, and idiomatic patterns specific to each language by pretraining them on a sizable sample of Python and Java code. Using concurrent Java-Python code pairings, this fundamental knowledge is further improved through a targeted fine-tuning process to increase translation accuracy and generalizability.

The project's main objective is to capture deeper program logic, control structures, and language-specific conventions in order to go beyond straightforward surface-level code conversion. Training procedure in two stages that involved careful pretraining and fine-tuning ensures that the final models preserve the functionality, purpose, and maintainability of the original programs besides ensuring correct code translation. This method addresses the common problems of code translation such as data types differences, replacement of libraries, how to handle exceptional conditions, and code which makes use of idiomatic expressions, which may not have equivalents in other languages.

The novelty of this work is the combination of state-of-the-art transformer architectures with a novel training regimen designed to accommodate the complexities in code translation. With a thorough analysis of the performance of these models across a variety of code samples and practical use cases, this project will offer valuable recommendations on best practices, inherent limits, and avenues for further enhancement for automatic code migration. In addition, the project will provide open-source resources—such as datasets, training scripts, and pretrained models—to facilitate future research endeavors and real-world applications by the community.

In short, the anticipated success of this initiative will likely enable painless transfer of software, promote code reuse, and speed up the introduction of contemporary programming languages into legacy environments. It has the potential to empower developers to modernize extensive codebases with more confidence, lessen technical debt, and improve innovation by simplifying the integration of future technology and programming models.

1.3 Organization of the Report

- Chapter 1 represents Introduction.
- Chapter 2 represents Research Literature Review.
- Chapter 3 represents Methodology.
- Chapter 4 represents Investigation/Experiment, Result, Analysis and Discussion.
- Chapter 5 represents Conclusion

Chapter 2 Research Literature Review

2.1 Existing Research and Limitations

Automatic code reading and translation between programming languages has advanced dramatically in recent years thanks to developments in artificial intelligence (AI) and natural language processing (NLP). Specifically, the development of pretrained transformer models has made it possible to map and generate code across a variety of languages, including Python and Java, more efficiently. The potential of these models to enhance code translation quality, decrease manual labor, and facilitate software maintenance in multilingual development settings has been shown in numerous research. Not with standing these developments, a number of obstacles and restrictions still exist, which spur further study in the area.

For instance, Raffel et al. [9] developed the Text-to-Text Transfer Transformer (T5), a unified transfer learning framework that reformulates all-natural language processing (NLP) tasks into a text-to-text format. The authors introduced the Colossal Clean Crawled Corpus (C4), a large-scale, cleaned English dataset compiled from web data, to pre-train their models. They systematically compared various pre-training objectives, Transformer-based architectures, and transfer strategies on several benchmarks, including GLUE, SuperGLUE, SQuAD, CNN/Daily Mail, and WMT translation tasks. Among their results, the largest T5 model, trained on over 1 trillion tokens and containing up to 11 billion parameters, achieved state-of-the-art performance across multiple benchmarks, notably matching or surpassing human-level performance on the SuperGLUE benchmark.

In a study, researchers developed CodeT5 [5], a unified pre-trained encoder-decoder Transformer model designed to enhance both code understanding and generation by leveraging developer-assigned identifiers and code comments. They pre-trained CodeT5 on the CodeSearchNet dataset, augmented with C/C# data from GitHub, using novel identifier-aware tasks like Masked Snap Prediction, Identifier-aware tasks like Masked Span Prediction, Identifier Tagging, and Masked Identifier Prediction, alongside a bimodal dual generation task to align natural language (NL) and programming language (PL). The model was fine-tuned on the CodeXGLUE benchmark, tackling tasks like code summarization, translation, refinement, defect detection, and clone detection, with a multi-task learning approach to boost generalization. The findings revealed that CodeT5 outperformed prior models like CodeBERT and PLBART across 14 CodeXGLUE sub-tasks, excelling in capturing code semantics due to its identifier-aware pre-training and achieving significant improvements in BLEU-4 and exact match scores. Notably, the bimodal dual generation enhanced NL-PL tasks, while multi-task learning boosted code summarization and refinement.

In another study, Sun et al. [10] introduced TransCoder, a unified transferable fine-tuning strategy for code representation learning that addresses challenges in adapting large code pre-trained models (CodePTMs) to various software intelligence tasks. The authors proposed a tunable prefix encoder to capture cross-task and cross-language transferable knowledge, enabling CodePTMs to generalize better, especially for tasks or languages with limited data. Their approach consists of a two-stage process: first, a universal knowledge prefix is learned through continual learning on multiple source tasks and languages; then, this prefix is prepended to new models for downstream adaptation. Extensive experiments on code understanding (e.g., clone and defect detection) and code generation (e.g., code summarization and translation) tasks using CodeT5 and PLBART backbones showed that TransCoder consistently outperformed standard fine-tuning, particularly in low-resource scenarios. For instance, on the code summarization task across six programming languages, TransCoder achieved higher BLEU scores than fine-tuning, with the most significant improvements observed in languages with smaller training sets, such as Ruby and JavaScript.

Wasim et al. introduce the PLBART model [3], a sequence-to-sequence model to master programing language understanding and generation task, pre-trained on a vast collection of Java to Python functions from several online sources. They fine-tuned PLBART on the CodeXGLUE benchmark [11], tackling tasks like code summarization, generation, translation, program repair, clone detection, and vulnerability detection with a Transformer-based architecture. PLBART surpassed or matched models like CodeBERT, achieving notable gains in BLEU and CodeBLEU scores, particularly in Ruby summarization (16% improvement) and Java-to-C# translation (9.5% improvement). It excelled at grasping program syntax and logic flow, even with minimal annotations. However, PLBART faced hurdles with PHP due to syntax mismatches and lacked pre-training on C/C++, potentially limiting its vulnerability detection capabilities. Ethical concerns include biases in GitHub data that might embed stereotypes and the risk of automation bias from over-relying on AI-generated code.

Feng et al. introduced CodeBERT [12], the first large bimodal pre-trained model designed for both natural language (NL) and programming language (PL) understanding. CodeBERT leverages a Transformer-based architecture and is trained on a hybrid objective that combines masked language modeling (MLM) and replaced token detection (RTD) using over 2 million function-level NL-PL pairs and more than 6 million unimodal code samples from six programming languages. The model was evaluated on downstream tasks such as natural language code search and code documentation generation, where it achieved state-of-the-art performance. For example, on the CodeSearchNet benchmark, CodeBERT (trained with both MLM and RTD, initialized from RoBERTa) achieved a mean reciprocal rank (MRR) of 0.7603, outperforming previous baselines. Additionally, in NL-PL probing tasks and code-to-documentation generation, CodeBERT consistently outperformed RoBERTa and models trained only on code. The research also demonstrated CodeBERT's ability to generalize to languages not seen during pre-training, showing competitive BLEU scores on C# code summarization.

Bahdanau et al. [13] proposed a novel neural machine translation (NMT) architecture that jointly learns to align and translate, introducing the now widely-used attention mechanism. Unlike the traditional encoder-decoder models that encode the entire source sentence into a fixed-length vector, their model allows the decoder to automatically (soft-)search for parts of the source sentence most relevant to generating each target word, thus mitigating the information bottleneck for long sentences. The approach uses a bidirectional RNN encoder to produce a sequence of annotation vectors for the input, and a decoder that, at each step, computes a context vector as a weighted sum of these annotations, where the weights are learned alignment scores. Experiments on English-to-French translation with the WMT'14 dataset showed that their model (RNNsearch) significantly outperformed conventional encoder-decoder models, achieving a BLEU score of 26.75 on all sentences and 34.16 on sentences without unknown words, comparable to the state-of-the-art phrase-based system Moses. Qualitative analysis demonstrated that the learned (soft) alignments were linguistically plausible, and the model was especially robust for longer sentences, making this a landmark contribution in neural machine translation.

In a study, Prithwish et al. [14] present an LLM-based code translation method and an associated tool called CoTran. That uses a Large language model (LLM) to translate entire programs between Java and Python, tackling the challenge of ensuing translated code compiles and functions like the original. They fine-tuned the CodeT5-base model using reinforcement learning, incorporating feedback from compilers and symbolic execution to check for compilation errors and functional equivalence. The training process involved two back-to-back LLMs–one translating from source to target language and another back to the source–using over 57,000 Java-Python code pairs from the AVATAR-TC dataset. They ensured robust translations by leveraging a keyword-based tokenizer and tools like Symflower to generate unit tests. CoTran outperformed 14 other tools, achieving up to 76.98% compilation accuracy and 48.68% functional equivalence for Python-to-Java translations, significantly improving over baselines like PLBART. However, it struggles with Python-to-Java translations due to challenges in inferring variable types from Python's dynamic typing. The approach is limited to function-level translations, making whole-program translations less reliable, as seen with low functional equivalence in some cases (e.g., 0.46% for Java-to-Python with TransCoder-based tools). Future work aims to extend CoTran to translate legacy code to modern languages, addressing these scalability issues.

A thorough review of existing literature reveals numerous lacunae and open questions in state-of-the-art studies. Most studies show a bias towards particular model architectures or a limited set of transformer variations, with a resultant dearth of comparative studies regarding preprocessing or fine-tuning strategies specifically targeted for code translation [5] [14]. Most studies also depend on large benchmark data sets whose nature may fail to capture actual codebase diversity and complexity in general and Java-Python translation in particular, leading to considerably large generalization gaps [6] [7]. Most importantly, there exists a severe lacuna in thorough investigations on dealing with distinctive code-related linguistic, syntactic, and semantic characteristics such as idiomatic phrases, library calls or invocations, and exceptions in the course of translating code [3] [12]. They are critical to real-world applications. Few studies thoroughly test low-resource languages or domains with limited parallel data availability to test model flexibility across different situations. Finally, though popular measures such as BLEU or MRR are widely adopted to test a model, they may fall short in gauging functional accuracy or maintainability of translated code and point towards a need for stringent testing paradigms [15]. The above findings drive this work in narrowing gaps through thorough investigation and improvement on state-of-the-art transformer-based approaches to Java-Python code translation, focusing particularly on model robustness, cross-lingual generalizability, and real-world adoptions.

Chapter 3 Methodology

3.1 System Design

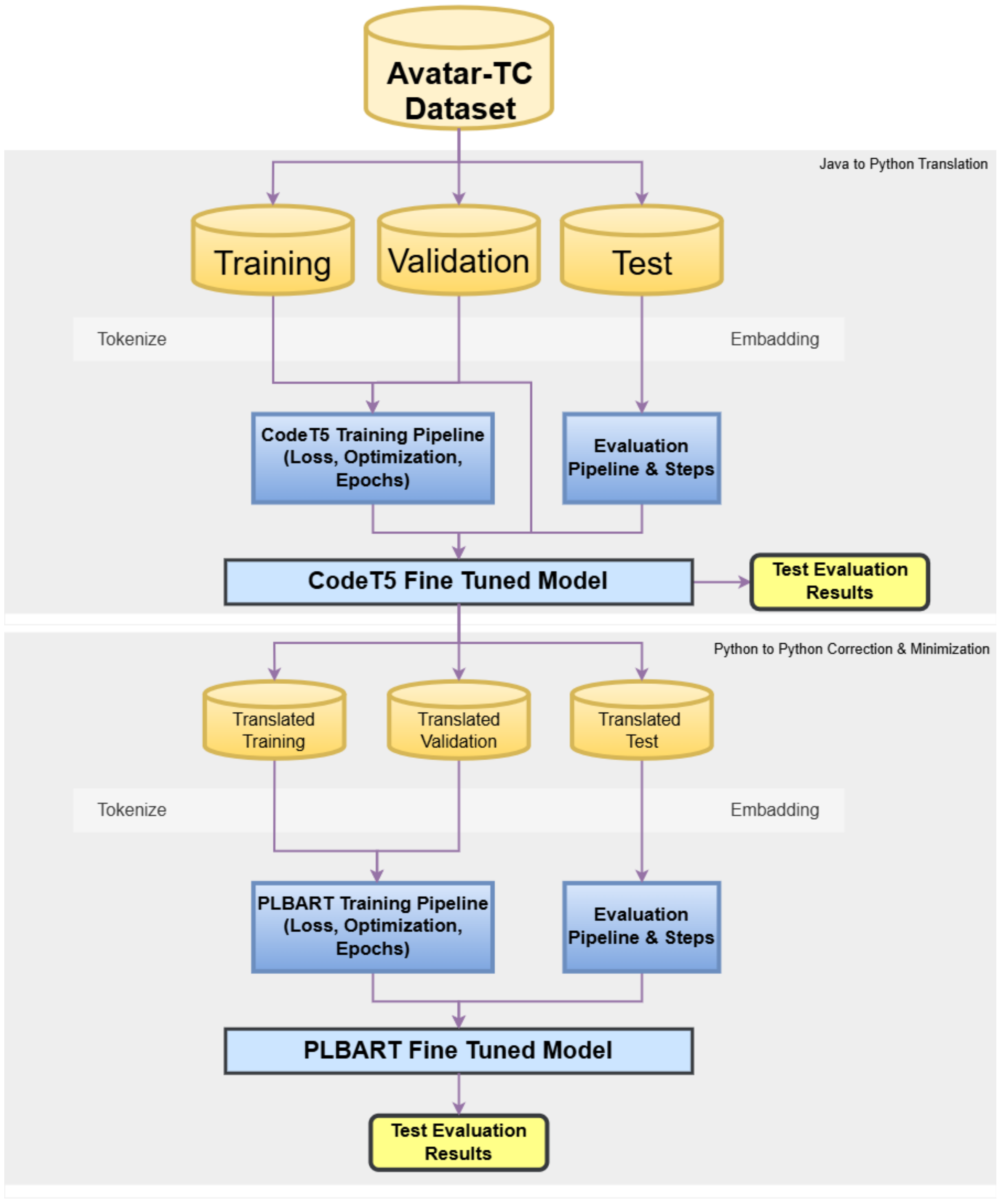

Avatar-TC Dataset
Java to Python Translation
Training
Validation
Test
Tokenize
Embadding
CodeT5 Training Pipeline (Loss, Optimization, Epochs)
Evaluation Pipeline & Steps
CodeT5 Fine Tuned Model
Test Evaluation Results
Python to Python Correction & Minimization
Translated Training
Translated Validation
Translated Test
Tokenize
Embadding
PLBART Training Pipeline (Loss, Optimization, Epochs)
Evaluation Pipeline & Steps
PLBART Fine Tuned Model
Test Evaluation Results

Figure 1: System Design Process Diagram

The system architecture for Java-to-Python code translation is illustrated in Figure 1. The proposed methodology is structured into the following components:

3.1.1 Dataset Collection

This work, we utilize the AVATAR-TC dataset originally created and published by a third-party research team [14]. The dataset was specially designed for code-to-code translation between Java and Python. It contains functionally equivalent code solutions across both languages, primarily sourced from competitive programming platforms such as LeetCode, Codeforces, GeeksforGeeks, AtCoder, Google Code Jam, and Project Euler. These platforms were chosen for their abundance of algorithmic problems and the availability of user-submitted solutions in multiple programming languages. The dataset authors employed web scripting techniques for sites like GeeksforGeeks and Codeforces, while the rest were collected from public repositories. Preprocessing involved tokenizing the code using model-specific tokenizers for both models. Additionally, all comments and docstrings were removed to focus on executable code. To maintain diversity and reduce redundancy, the authors selected up to five solutions per problem that were most dissimilar from one another, utilizing the dfflib library for Python and an equivalent open-source library for Java. The preprocessing and structure of this dataset closely follow the methodology outlined in the AVATAR dataset [16], ensuring its suitability for training and evaluating models in the code translation domain. Table I illustrates the statics distribution of the dataset used in this study.

TABLE I. STATICS OF THE AVATAR-TC DATASET

Sub-dataset	Pairs of Java-Python Codes		
	Train	Valid	Test
Aizu	14,019	41	190
AtCoder	13,558	19	97
Codeforces	28,311	96	401
Google CodeJam	347	1	4
LeetCode	81	7	18
GeeksForGeeks	3,753	268	95
ProjectEuler	110	11	41
Total	55,179	443	1,746

3.1.2 Tokenization and Alignment

The data preprocessing stage is carried out before the fine-tuning of the models because the data ought to be clean, consistent, and amenable to proper analysis. A tokenizer relating to the pre-trained model is fed each code snippet. The tokenization procedure means splitting the code into sub word units and inserting special tokens in case of need (e.g., [CLS], [SEP], and [PAD]). Code snippets are padded or cut off to no more than a desired length to have a uniform input size in the data. The source and target code lines are both post-processed to ensure they fit under the maximum lengths established, and this is useful in preserving the efficiency of computing the model during training. This is done to ensure that every pair of the Java and Python code is aligned appropriately, where each pair of Java code snippets is aligned directly with a Python code snippet and vice versa, and this pair is retained with each training instance.

3.1.3 Model Selection

The primary models used for this task are Transformer-based architectures: CodeT5 small and base variants and PLBART base; these models are specifically designed to handle code translation, summarization, and generation tasks and were pre-trained on a large corpus of code to understand the syntax and semantics of different programming languages. The models chosen for this research as follows:

CodeT5: CodeT5 is an extension of T5 designed specifically for software-related task [9]. It uses the same encoder-decoder architecture as T5 but is pre-trained on large-scale code corpora from CodeSearchNet, converting multiple programming languages, including Java and Python. CodeT5 introduces task-specific prefixes and supports tasks like code summarization, generation, and translation. Its training objectives include masked span prediction and indenter prediction, helping it learn structure-aware code representation [5]. In our work, we use the CodeT5 model to translate the initial Java codes into Python code.

PLBART: PLBART is a pre-trained encoder-decoder model adapted from BART that is tailored for software engineering tasks [17] [3]. It is trained on parallel source and target code from multiple languages and is suitable for code-to-code translation tasks (e.g., Java

→ Python). PLBART performs denoising pre-training with corrupted inputs to learn robust sequence generation. It is capable of handling both understanding and generation tasks and benefits from the dual-encoder-decoder structures. This structure makes it especially effective for tasks that involve translating or transforming code between programming languages. In our work, we use the PLBART model to correct the synthetic and semantic errors from the translated Python code and minimize the Python code.

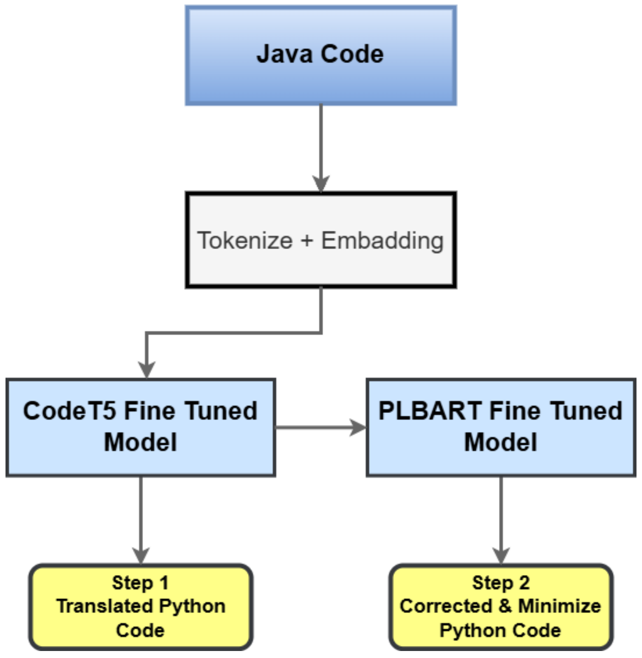


Figure 2: Code Translation Workflow

3.1.4 Model Training

The training process involves fine-tuning the pre-trained models on the Java-to-Python code pairs. Figure 2 illustrates the code translation process for code translation tasks using the fine-tuned model. For each model, the following steps are performed:

- **Tokenization:** Initially, Java code is tokenized using the corresponding tokenizer for the CodeT5 model, and the Translated Python code is tokenized using the model-specific tokenizer for the PLBART model (e.g., RobertaTokenizer for CodeT5 and

PLBartTokenizer for PLBART). These tokenizers convert the Java code and Python code into numerical representations that can be fed into the models.

- **Loss Calculation:** During training, the models were optimized by minimizing the loss function. The loss is computed as the difference between the predicted output and the actual target. The most commonly used loss function in this case is cross-entropy loss.
- **Optimization:** The AdamW optimizer is used with a learning rate of 3e-4, weight decay of 0.05, epsilon of 1e-8 for CodeT5 base model and CodeT5 small variant model, and learning rate of 3e-5, weight decay of 0.01 for PLBART base model. The learning rate is tuned to ensure efficient convergence without overfitting. Gradients are computed and backpropagated to update the model's weights, and improve the translation accuracy over time.
- **Batching:** Training is performed in batches, with a batch size of 8 for both steps training. This helps in improve computational efficiency by leveraging the parallelism in GPUs and reducing training time.
- **Epochs:** The models are trained for a fixed number of epochs (10 epochs), during which the model parameters are continually updated to improve translation performance. After each epoch, the model is evaluated on a validation set to monitor its progress and adjust hyperparameters if necessary.

3.1.5 Evaluation Metrics

We evaluated our models using BLEU and CodeBLEU evaluation metrics, considering only the top translation with the highest log probability. The BLEU (Bilingual Evaluation Understudy) [18] score is a metric commonly used for evaluating sequence-to-sequence generation tasks. It measures the precision of n-grams in the generated output compared to the reference output. We calculated BLEU using a smoothing function to handle cases where no exact match exists for higher-order n-grams. While BLEU is effective for natural language tasks, it falls short in evaluating programming code because it does not account for code structure or semantics. CodeBLEU [19] extends BLEU by adding four features: standard n-gram matching, weighted n-gram matching, syntax matching, and semantic matching. CodeBLEU uses various methods to understand better the correctness, structure, and purpose of code. This makes it a more accurate and valid assessment than regular BLEU for assessment than regular BLEU for programming-related tasks.

3.2 Hardware and/or Software Components

Several software tools and technologies were employed to facilitate data processing, model training, and evaluation to implement the Java-to-Python code translation project. The entire project was developed using Python due to its extensive support for machine learning and natural language processing libraries. The core models were implemented using the PyTorch deep learning framework, offering flexibility and compatibility with advanced transformer-based architectures. HuggingFace Transformers library was utilized to access and fine-tune pre-trained models such as CodeT5-small, CodeT5-base, and PLBART-base each designed for understanding and generating programming languages. CodeT5 was the primary model for its specialized training on code-related tasks, particularly code translation. The custom dataset, created explicitly for source code translation tasks, was used and split into training, validation, and testing sets. Tokenization was handled using model-specific tokenizers from HuggingFace to ensure consistent and accurate input formatting. Model training involved a series of loss functions, optimizers, and multiple epochs to maximize performance. Translation quality was evaluated using BLEU and CodeBLEU scores, which were imported from Microsoft CodeXGLUE GitHub repo. Development and testing were carried out in Kaggle Notebook and Google Colab with GPU acceleration and interactive environments for testing. This combination of tools provided a robust setup for experimenting with multiple models and delivering efficient code translation from Java to Python.

TABLE II. A SAMPLE SOFTWARE/HARDWARE TOOLS TABLE

Tool	Functions	Other similar Tools (if any)	Why selected this tool
Python3	Main programming language used for model training, preprocessing, and evaluation	R, Java	Widely used in data science and ML, Deep Learning, Natural Language Processing,

			large ecosystem of libraries, easy syntax.
PyTorch	Deep learning framework for building and training transformer-based models	TensorFlow	Flexible and dynamic computation graph, widely used in academia, research, easier debugging and customization, seamless integration with HuggingFace Transformers
HuggingFa ce Transforme rs	Pre-trained models and tokenizers (CodeT5, PLBART, etc.)		Easy access to state-of-the-art transformer models with simple APIs
Tokenizer (from HuggingFa ce)	Converts code to token sequences as input to models	Byte-Pari Encoding	Model-specific tokenization ensures compatibility and better performance
Kaggle Notebook	Development and Testing environment	VS Code, Google Colab	Easy setup, GPU access (P100), interactive for model development and debugging
CodeT5	Performs code translation from java to python	TransCoder, CodeTrans, CodeBERT	Pretrained for code tasks, supports prefix-tuning, open-sourced, and low resource model
PLBART	Performs code correction and minimization tasks	CodeBERT, CodeT5	Specifically, pretrained trained java and python code for generation and translation tasks and low resource model
CodeBLEU	Evaluation metric designed for programming language translation	BLEU, Exact Match, CodeBLEU+	Evaluates n-gram, syntax and dataflow match– better suited for code than BLEU

3.3 Hardware and/or Software Implementation

The implementation of this project was entirely software-based, focusing on developing a two-stage approach for automatic Java-to-Python code translation using Transformer-based models. The dataset, consisting of aligned Java and Python code pairs, was preprocessed to prepare it for model training. Tokenization was performed using model-specific tokenizers from the HuggingFace Transformers library to ensure compatibility and alignment. The PyTorch framework was used to load and fine-tune pre-trained models, including CodeT5 and PLBART. The training pipeline involved defining loss functions, applying the AdamW optimizer, and running multi-epoch training with validation evaluations to monitor performance. Google Colab and Kaggle Notebooks were the primary development environments, offering GPU acceleration and interactive code execution. After training, the models were evaluated using BLEU and CodeBLEU metrics to assess translation quality and manual inspection for semantic correctness. The implementation integrated several tools and libraries, such as Python, PyTorch, and HuggingFace Transformers, to build a robust and scalable code translation system.

Chapter 4 Investigation/Experiment, Result, Analysis and Discussion

The evaluation and inference phases of the Java-to-Python translation pipeline, utilizing the CodeT5 and PLBART models, demonstrate varying degrees of success in translating Java Code to Python and refining the output. The pipeline was tested on various Java code samples, ranging from simple "Hello World" outputs to more complex recursive functions and data structure manipulations. The results indicate that the initial and subsequent correction steps produce functional Python code, but in some cases, some inconsistencies and limitations affect the overall accuracy and reliability.

Our model is designed to accept the raw Java code as input without adding any special tokens. Before being fed into the model, this code is tokenized and preprocessed as plain Java syntax. The model then generates the corresponding Python code in a flattened sentence format enriched with special structural tokens such as NEW_LINE and INDENT. These tokens represent line breaks and indention levels within a single sequence, allowing the model to encode Python's strict formatting rules during generation. We use a unique post-processing software to turn this output into executable Python code. This script reads and eliminates these tokens, then combines the necessary indentation and line structure to make a clean, well-formatted Python program. This ensures that while the model operates in a format suitable for sequence learning, the final outputs remain fully usable and syntactically correct. Below are examples of the Java input and the corresponding Python output.

Sample Java Input:

```
import java . util . * ; class TreeNode { public int val ; public TreeNode left ; public TreeNode
right ; public TreeNode ( int rootData ) { this . val = rootData ; this . left = null ; this . right =
null ; } } public class Solution { public TreeNode convertBST ( TreeNode root ) { int sum = 0
; TreeNode node = root ; Stack < TreeNode > stack = new Stack < TreeNode > ( ) ; while ( !
stack . isEmpty ( ) || node != null ) { while ( node != null ) { stack . add ( node ) ; node = node .
right ; } node = stack . pop ( ) ; sum += node . val ; node . val = sum ; node = node . left ; }
return root ; } public List < Integer > printTree_vLR ( TreeNode node , List < Integer > arr ) {
if ( node != null ) { arr . add ( node . val ) ; arr = printTree_vLR ( node . left , arr ) ; arr =
printTree_vLR ( node . right , arr ) ; } return arr ; } public static void main ( String [ ] args ) {
```

```
Solution sObj = new Solution ( ) ; TreeNode root = new TreeNode ( 4 ) ; root . left = new TreeNode ( 1 ) ; root . right = new TreeNode ( 6 ) ; root . left . left = new TreeNode ( 0 ) ; root . left . right = new TreeNode ( 2 ) ; root . right . left = new TreeNode ( 5 ) ; root . right . right = new TreeNode ( 7 ) ; root . left . right . right = new TreeNode ( 3 ) ; root . right . right . right = new TreeNode ( 8 ) ; TreeNode out = sObj . convertBST ( root ) ; List < Integer > arr = sObj . printTree_vLR ( out , new ArrayList < Integer > ( ) ) ; System . out . println ( arr ) ; } }
```

Sample Python Output:

```
class TreeNode : NEW_LINE INDENT def __init__ ( self , rootData ) : NEW_LINE INDENT self . val = rootData NEW_LINE self . left = None NEW_LINE self . right = None NEW_LINE DEDENT DEDENT class Solution ( object ) : NEW_LINE INDENT def convertBST ( self , root ) : NEW_LINE INDENT total = 0 NEW_LINE node = root NEW_LINE stack = [ ] NEW_LINE while stack or node is not None : NEW_LINE INDENT while node is not None : NEW_LINE INDENT stack . append ( node ) NEW_LINE node = node . right NEW_LINE DEDENT node = stack . pop ( ) NEW_LINE total += node . val NEW_LINE node . val = total NEW_LINE node = node . left NEW_LINE DEDENT return root NEW_LINE DEDENT def printTree_vLR ( self , node , arr ) : NEW_LINE INDENT if node != None : NEW_LINE INDENT arr . append ( node . val ) NEW_LINE arr = self . printTree_vLR ( node . left , arr ) NEW_LINE arr = self . printTree_vLR ( node . right , arr ) NEW_LINE DEDENT return arr NEW_LINE DEDENT DEDENT if __name__ == " _ _ main _ _ " : NEW_LINE INDENT sObj = Solution ( ) NEW_LINE root1 = TreeNode ( 4 ) NEW_LINE root1 . left = TreeNode ( 1 ) NEW_LINE root1 . right = TreeNode ( 6 ) NEW_LINE root1 . left . left = TreeNode ( 0 ) NEW_LINE root1 . left . right = TreeNode ( 2 ) NEW_LINE root1 . right . left = TreeNode ( 5 ) NEW_LINE root1 . right . right = TreeNode ( 7 ) NEW_LINE root1 . left . right . right = TreeNode ( 3 ) NEW_LINE root1 . right . right . right = TreeNode ( 8 ) NEW_LINE out = sObj . convertBST ( root1 ) NEW_LINE arr = sObj . printTree_vLR ( out , [ ] ) NEW_LINE print ( arr ) NEW_LINE DEDENT
```

In second step we finetune the PLBART model with the translated train, validation, test Python code with the corresponding original python code. This model accepts the translated Python code as input with special tokens NEW_LINE and INDENT. Here also before fed into the model, this code is tokenized and processed as plain Python syntax. The model then generates the corresponding Python code in a flattened sentence with the same special structural tokens. The PLBART model was pretrained on python code so it provides simplified and more accurate outputs compare to the original Python codes. Below are examples of the translated Python input and the corresponding Python output.

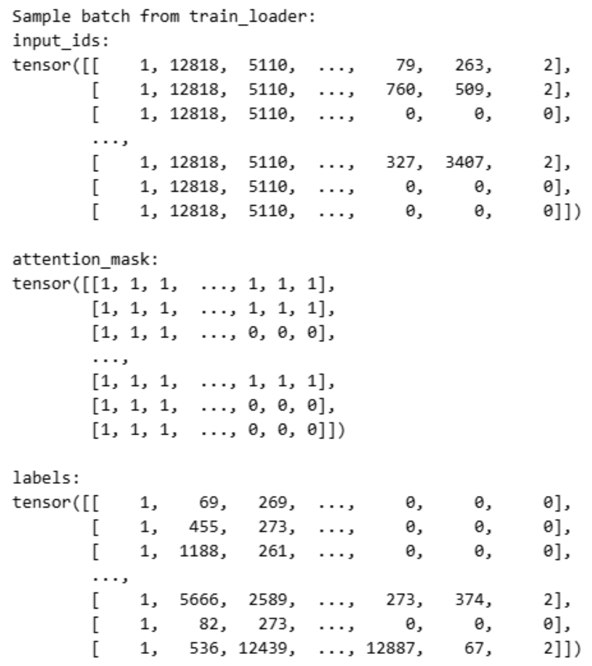
Translated Python Input:

```
class TreeNode : NEW_LINE INDENT def __init__ ( self , rootData ) : NEW_LINE INDENT self . val = rootData NEW_LINE self . left = None NEW_LINE self . right = None NEW_LINE DEDENT DEDENT class Solution ( object ) : NEW_LINE INDENT def convertBST ( self , root ) : NEW_LINE INDENT sum = 0 NEW_LINE node = root NEW_LINE stack = [ ] NEW_LINE while ( len ( stack ) != 0 or node != None ) : NEW_LINE INDENT while ( node != None ) : NEW_LINE INDENT stack . append ( node ) NEW_LINE node = node . right NEW_LINE DEDENT node = stack . pop ( ) NEW_LINE sum += node . val NEW_LINE node . val = sum NEW_LINE node = node . left NEW_LINE DEDENT return root NEW_LINE DEDENT def printTree_vLR ( self , node , arr ) : NEW_LINE INDENT if ( node != None ) : NEW_LINE INDENT arr . append ( node . val ) NEW_LINE arr = printTree_vLR ( node . left , arr ) NEW_LINE arr = printTree_vLR ( node . right , arr)
```

Sample Python Output:

```
class TreeNode : NEW_LINE INDENT def __init__ ( self , rootData ) : NEW_LINE INDENT self . val = rootData NEW_LINE self . left = None NEW_LINE self . right = None NEW_LINE DEDENT DEDENT class Solution ( object ) : NEW_LINE INDENT def convertBST ( self , root ) : NEW_LINE INDENT total = 0 NEW_LINE node = root NEW_LINE stack = [ ] NEW_LINE while stack or node is not None : NEW_LINE INDENT while node is not None : NEW_LINE INDENT stack . append ( node ) NEW_LINE node = node . right NEW_LINE DEDENT node = stack . pop ( ) NEW_LINE total += node . val NEW_LINE node . val = total NEW_LINE node = node . left NEW_LINE DEDENT return root NEW_LINE DEDENT def printTree_vLR ( self , node , arr ) : NEW_LINE INDENT if node != None : NEW_LINE INDENT arr . append ( node . val ) NEW_LINE arr = self . printTree_vLR ( node . left , arr ) NEW_LINE arr = self . printTree_vLR ( node . right , arr ) NEW_LINE DEDENT return arr NEW_LINE DEDENT DEDENT if __name__ == " _ _ main _ _ " : NEW_LINE INDENT sObj = Solution ( ) NEW_LINE root1 = TreeNode ( 4 ) NEW_LINE root1 . left = TreeNode ( 1 ) NEW_LINE root1 . right = TreeNode ( 6 ) NEW_LINE root1 . left . left = TreeNode ( 0 ) NEW_LINE root1 . left . right = TreeNode ( 2 ) NEW_LINE root1 . right . left = TreeNode ( 5 ) NEW_LINE root1 . right . right = TreeNode ( 7 ) NEW_LINE root1 . left . right . right = TreeNode ( 3 ) NEW_LINE root1 . right . right . right = TreeNode ( 8 ) NEW_LINE out = sObj . convertBST ( root1 ) NEW_LINE arr = sObj . printTree_vLR ( out , [ ] ) NEW_LINE print ( arr ) NEW_LINE DEDENT
```

During the training and translation process we tokenize the flatten Java code and translated Python code in both steps. For the CodeT5 model we use the model specific RobertaTokenizer, below in Figure 3 illustrate the sample tokenize batch of the initial Java code as input ids, attention mask, and corresponding python code as labels. For the PLBART model PLBartTokenizer, below in Figure 4 illustrate the sample tokenize batch of the translated Python code as input ids, attention mask, and corresponding python code as labels.

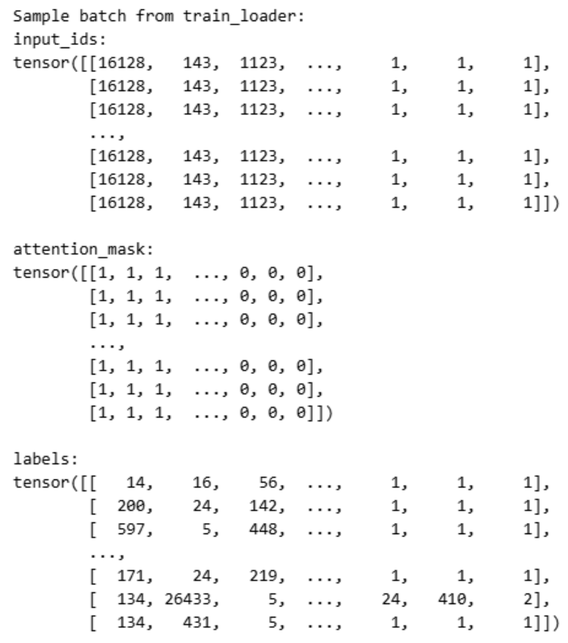


```
Sample batch from train_loader:
input_ids:
tensor([[    1, 12818,  5110,  ...,    79,   263,     2],
        [    1, 12818,  5110,  ...,   760,   509,     2],
        [    1, 12818,  5110,  ...,     0,     0,     0],
        ...,
        [    1, 12818,  5110,  ...,   327,  3407,     2],
        [    1, 12818,  5110,  ...,     0,     0,     0],
        [    1, 12818,  5110,  ...,     0,     0,     0]])

attention_mask:
tensor([[1, 1, 1,  ..., 1, 1, 1],
        [1, 1, 1,  ..., 1, 1, 1],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 1, 1, 1],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])

labels:
tensor([[    1,    69,   269,  ...,     0,     0,     0],
        [    1,   455,   273,  ...,     0,     0,     0],
        [    1,  1188,   261,  ...,     0,     0,     0],
        ...,
        [    1,  5666,  2589,  ...,   273,   374,     2],
        [    1,    82,   273,  ...,     0,     0,     0],
        [    1,   536, 12439,  ..., 12887,    67,     2]])
```

Figure 3: Sample batch for CodeT5 model



```
Sample batch from train_loader:
input_ids:
tensor([[16128,   143,  1123,  ...,     1,     1,     1],
        [16128,   143,  1123,  ...,     1,     1,     1],
        [16128,   143,  1123,  ...,     1,     1,     1],
        ...,
        [16128,   143,  1123,  ...,     1,     1,     1],
        [16128,   143,  1123,  ...,     1,     1,     1],
        [16128,   143,  1123,  ...,     1,     1,     1]])

attention_mask:
tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])

labels:
tensor([[   14,    16,    56,  ...,     1,     1,     1],
        [  200,    24,   142,  ...,     1,     1,     1],
        [  597,     5,   448,  ...,     1,     1,     1],
        ...,
        [  171,    24,   219,  ...,     1,     1,     1],
        [  134, 26433,     5,  ...,    24,   410,     2],
        [  134,   431,     5,  ...,     1,     1,     1]])
```

Figure 4: Sample batch for PLBART model

4.1 Model Configurations:

CodeT5 is a transformer-based model based on the T5 architecture, designed for code-related tasks such as code generation, summarization, and translation. PLBART is a transformer-based model designed for code-related tasks, particularly for code refinement and correction. CodeT5 and PLBART models configurations of our study are represented below in Table III. Each model has different training configurations, different parameter size, input length, output length, and finetuning task.

TABLE III. MODEL TRAINING CONFIGURATIONS

Aspect	CodeT5-base	PLBART-base
Model Type	T5ForConditionalGeneration	PLBartForConditionalGeneration
Architecture	Encoder-Decoder Transformer (T5-based)	Encoder-Decoder Transformer (BART-based)
Tokenizer	RobertaTokenizer	PLBartTokenizer
Vocabulary Size	32,000 tokens	50,005 tokens
Embedding Dimension	768	768
Encoder Layers	12	6
Decoder Layers	12	6
Input Length	350	256
Output Length	256	256
Input Format	“Translate Java to Python:” + Java code	Translated Python Code from CodeT5
Output	Initial Translated Python Code	Corrected/minimized Python Code
Fine-Tuning Task	Java-to-Python Translation	Python-Python Code Correction/Minimization
Generation Parameters	Beam size = 10, batch size = 8	Beam size = 10, batch size = 32
Model Size	220M parameters	139M parameters

4.2 Evaluation Metrics and Scores:

The evaluation phase assessed the model’s translation accuracy using metrics: BLEU score and CodeBLEU score, n-gram match, weighted n-gram match, syntax match, and dataflow match.

TABLE VI. MODEL EVALUATION RESULTS

Step	Model	BLEU	CodeBLEU	n-gram match	Weighted n-gram match	Syntax match	Dataflow match
1	CodeT5-base	51.72	0.4913	0.5094	0.5274	0.4945	0.4338
2	CoodeT5-base + PLBART-base	28.92	0.3327	0.3290	0.3476	0.3432	0.3109
1	CodeT5-small	51.32	0.4855	0.5067	0.5258	0.4947	0.4150
2	CodeT5-small + PLBART-base	41.68	0.4418	0.4168	0.4382	0.4947	0.4175

According to the evaluation results given in Table VI, the model based on CodeT5, fine-tuned on Java-to-Python translation, shows the best results compared to other configurations. It produced the best BLEU score of 51.72 and the best CodeBLEU score of 0.4913, which means that the quality of its translated outputs is better and closer to the reference code that human beings have written. The model also showed the best score in terms of n-gram match (0.5094), which indicates a strong overlap in token sequences in generated code compared to reference code. Furthermore, the weighted n-gram match value of 0.5274 indicates a strong contextual matching where the significant tokens and longer strings obtain more weight. The syntax match score is 0.4945, so the generated Python code is structurally correct at all times, and the dataflow match score is 0.4338, the highest possible dataflow match score, thus certifying that the control and data dependencies in the Python output code most closely follows that in the reference, preserving logical correctness.

In contrast, the second-stage model (CodeT5-base + PLBART-base), which uses PLBART to refine the initial translations, yielded lower scores across all metrics — including a BLEU score of 28.92 and a CodeBLEU score of 0.3327. This decline is expected, as PLBART is applied not to re-translate but to correct and optimize the output from the first step, potentially removing redundant or verbose code. As a result, the translated Python code becomes more concise and possibly cleaner, but the surface-level similarity to the reference (measured by BLEU) may decrease.

Similarly, the CodeT5-small model demonstrated competitive performance in the first step, with a BLEU score of 51.32 and CodeBLEU of 0.4855, slightly below the base version. It got the most excellent syntax match score of 0.4947, which shows that it is good at keeping the code structure. However, it lagged in dataflow and weighted n-gram matches, suggesting slightly weaker semantic and contextual accuracy.

When combined with PLBART (step 2), the CodeT5-small + PLBART-base model improved upon the base + PLBART setup, achieving a BLEU score of 41.68 and CodeBLEU of 0.4418, along with strong syntax (0.4947) and dataflow (0.4175) match scores. PLBART works better with the smaller CodeT5 variant, successfully correcting structural or semantic issues while retaining key elements of the original translation. However, it did not significantly reduce code redundancy, as seen in the relatively high BLEU and CodeBLEU scores compared to the CodeT5-base + PLBART combination.

4.3 Final Translation from Java to Python

TABLE V. JAVA TO PYTHON CODE TRANSLATION RESULTS

Problem	Java Input Code	Initial Step 1 Translation	Step 2 Correction or Minimization
Hello	`public class HelloWorld {`	`import sys`	`print(" Hello, World!")`
World	`public static void main(String[] args) {`	`print(" Hello, World!")`	
	`System.out.println("Hello, World!");`		
	`}`		
	`}`		
Recursive	`public class Main {`	`def sum ( start , end ) :`	`def sum (start, end):`
Sum	`public static void main(String[] args) {`	`if end - start > 1 :`	`if end - start > 1:`
	`int result = sum(5, 10);`	`return end + sum ( start , end -`	`return end + sum (start, end`
	`System.out.println(result);`	`1 )`	`- 1)`
	`}`	`else :`	`else:`
	`public static int sum(int start, int end) {`	`return end`	`return end`
	`if (end > start) {`	`print ( sum ( 5 , 10 ) )`	`print(sum( 5, 10 ))`
	`return end + sum(start, end - 1);`		
	`} else {`		
	`return end;`		
	`}`		
	`}`		
	`}`		
HashSet	`import java.util.HashSet;`	`import sys`	`import sys`
Example		`def solve ( ) :`	`def solve():`
	`public class Main {`	`c = set ( )`	`c = set ()`
	`public static void main(String[] args) {`	`for i in sys . stdin :`	`for line in sys.stdin:`
	`HashSet<String> cars = new`	`if i . strip ( ) == ' \n ' :`	`if line.strip() =='\n':`
	`HashSet<String>();`	`c . add ( ' Volvo ' )`	`c.add('Volvo')`
	`cars.add("Volvo");`	`elif i . strip ( ) == ' BMW ' :`	`elif line.strip() =='BMW':`
	`cars.add("BMW");`	`c . add ( ' Ford ' )`	`c.add('Ford')`
	`cars.add("Ford");`	`elif i . strip ( ) == ' BMW ' :`	`elif line.strip() =='BMW':`
	`cars.add("BMW");`	`c . add ( ' Mazda ' )`	`c.add('Mazda')`
	`cars.add("Mazda");`	`else :`	`else:`
	`System.out.println(cars);`	`c . add ( i )`	`c.add( line )`
	`}`	`print ( c )`	`print(".join(c))`
	`}`	`if __name__ == '__main__':`	`if __name__ == '__main__':`
		`solve ( )`	`solve()`

While Loop	**public class WhileLoopExample** {
	  public static void main(String[] args) {
	    int count = **1**;
	
	    while (count <= **5**) {
	      System.out.println("Count is: " + count);
	      count++;
	    }
	  }
	}
		def whileLoop ( ) :
		  count = **1**
		  while count <= **5** :
		    print ("Count is : " , count)
		    count += **1**
		if __name__ == '__main__':
		  while True :
		    print (" WhileLoopExample ")
			def whileLoop():
			  count = **1**
			  while count <= **5**:
			    print("Count is :", count)
			    count += **1**
			if __name__ == '__main__':
			  while True:
			    print("WhileLoopExample")

Count Fives	**public class CountFives** {
	  public static void main(String[] args) {
	    int[] numbers = {**5**, **1**, **5**, **3**, **5**, **7**, **2**, **5**};
	    int count = **0**;
	
	    for (**int** num : numbers) {
	      if (num == **5**) {
	        count++;
	      }
	    }
	
	    System.out.println("Number of times 5 appears: " + count);
	  }
	}
		def CountFives (n) :
		  count = **0**
		  for i **in** range (n) :
		    if i == **5** :
		      count += **1**
		  print (" Number of times 5 appears " , count)
		n = [**5** , **1** , **5** , **3** , **5** , **7** , **2** , **5**]
			CountFives (n) :
			  count = **0**
			  for i **in** range (n) :
			    if i == **5** :
			      count += **1**
			  print (" Number of times 5 appears ", count)
			n = [**5**, **1**, **5**, **3**, **5**, **7**, **2**, **5**]

Table V demonstrates how our models translate various Java code into Python in two stages: initial translation and post-correction/minimization. Each row represents a different problem type, such as printing “Hello World”, performing recursive sum, working with sets, using loops, and counting specific elements in a list. In the first column, the original Java code is shown exactly as written. The second column shows the model’s initial Python translation, and the third column represents the refined output after applying a correction/minimization model, where redundant expressions are cleaned and formatting is improved. For instance, in the recursive sum example, the function call becomes more compact and readable. Overall, the results confirms that the proposed two-stage model effectively translates and refines Java code into accurate and syntactically correct Python code.

Chapter 5 Conclusions

8.1 Summary

The main aim of this research was to propose a two-stage method for Java-to-Python code transformation based on pre-trained transformer models. There is an increasing need for these kinds of translational programming solutions since there is always a need to rewrite old programs in a new programming environment. The traditional ways of translations are not error-free, time-consuming and only those with a high degree of source and target language programming can perform that. We therefore proposed a two-headed model driven by two recent transformer-based models, CodeT5 and PLBART, particularly designed to handle code comprehension and code generation.

During the first phase, unprocessed Java code is translated to Python using the CodeT5 model. This model was carefully fine-tuned on the AVATAR-TC dataset composed of a high-quality selection of Java-Python code pairs harvested from competitive programming platforms like Codeforces, GeeksforGeeks, and LeetCode. Then, during the second phase, the translated code is refined using the PLBART model. The PLBART model is optimized to reduce syntactic and semantic mismatches and enhance code generation closer to Pythonic guidelines.

The method was evaluated through both CodeBLEU and BLEU measures, and CodeT5-base achieved a BLEU and CodeBLEU measure value of 51.72 and 0.4913, respectively, which are good indicators of improved translation quality. In addition, complementary methods such as n-gram matching, syntactic matching, and dataflow matching offer further evidence supporting the suggested methodology. For example, a syntactic match value of 0.4945 helps validate that the code generated conforms with the conventional structure through Python programming. This two-stage set-up was found to satisfactorily bridge the difference between static Java forms and dynamic semantics of Python.

It helps modernize older software through automated code translation, with results comparable to those achieved by human developers, thus enabling better interoperability between different programming languages and reducing technical debt in large codebases.

8.2 Limitations

While the system shows promising efficacy, there are certain constraints present. Awareness of these limitations is vital for understanding the set parameters and possible direction of this study:

Inadequate Diversity in Dataset: Although the AVATAR-TC dataset is well-structured, its primary focus is algorithmic. It lacks the inclusion of code from practical applications that usually encompass external libraries, complex class hierarchies, graphical user interfaces, and multithreading capabilities. The resulting models could, therefore, have limited generalizability when transferred to software projects at an industrial scale.

Syntactic Errors in Edge Cases: Although the CodeT5 and PLBART models were very good on routine tasks, some edge cases—namely those involving nested control structures or idiomatic expressions characteristic of a given language—sometimes generated syntactically incorrect or wordy translations.

Insufficiency of Runtime Verification: The generated code was not tested at runtime for functional correctness. Although syntactic correctness was ensured through the use of structure tokens like NEW_LINE and INDENT, logical errors can still occur without dynamic run-time examination.

Surface-Level Similarity Loss following Refinement: The second phase with PLBART correction generally decreased BLEU and CodeBLEU scores since it was optimizing code in favor of brevity and correctness sometimes at a cost of surface form mismatch from that of the target code.

Weakness in Representation of Assessment Criteria: While used very widely, BLEU and CodeBLEU poorly represent real-world usefulness. They ignore runtime performance, exception handling, and maintainability that are extremely crucial constituents within production codebases.

8.3 Future Improvement

To overcome these aforementioned challenges and expand the system, different future improvements are proposed:

Broaden and Diversify Dataset: Incorporating real-code from open-source repositories, commercial-quality projects, and domain-specific programs would aid improved generalization and quality on production-scale codebases.

Apply Unit Testing and Runtime Verification: After the translation process, the code can be automatically verified using test cases to check its logical correctness. A test-driven feedback loop can be created, allowing the model to be retrained on the failed outputs.

Language-Aware Fine-tuning: Specialized training methods can be devised to successfully deal with idiomatic expressions and language-specific constructs, such as Java's ArrayList vs. Python's list, to improve accuracy in certain contexts.

Bidirectional Translation Capability: This feature can be extended to include translation from Python to Java, as well as other programming languages like C++, JavaScript, or Rust, thus making it a complete multilingual code translation system.

Ethical and Responsible AI Needs: The further conception should make sure the equality will be kept, the data confidentiality will remain, and no discriminatory matters will occur in the translational area. Translation biases can happen in case training of the model is realized by using the content recruiting from the community. Besides it, it is necessary to guarantee that the code, which we use in datasets, is licensed correctly.

Educational Integration: Besides, it is an educative instrument, and consequently, students' understanding of multi-language programming paradigms is elevated by the provision of examples, quizzes, and comparisons between pairs of code.

References

[1] J. X. X. X. C. L. X. J. Xiang Chen, "A Systematic Literature Review on Neural Code Translation," *arXiv.*

[2] N. S. N. P. J. U. L. J. A. N. G. L. K. I. P. Ashish Vaswani, "Attention Is All You Need," *arXiv,* 2017.

[3] S. C. B. R. K.-W. C. Wasi Uddin Ahmad, "Unified pre-training for program understanding and generation," *arXiv,* 2021.

[4] N. S. A. R. K. L. S. N. M. M. Y. Z. W. L. P. J. L. Colin Raffel, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv,* 2019.

[5] W. W. S. J. S. C. H. Yue Wang, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv,* 2021.

[6] B. R. L. C. G. L. Marie-Anne Lachaux, "Unsupervised Translation of Programming Languages," *arXiv,* 2020.

[7] J. X. X. X. C. L. X. J. Xiang Chen, "A Systematic Literature Review on Neural Code Translation," *arXiv,* 2025.

[8] O. V. Q. V. L. Ilya Sutskever, "Sequence to Sequence Learning with Neural Networks," *arXiv,* 2014.

[9] N. S. A. R. K. L. S. N. M. M. Y. Z. W. L. P. J. L. Colin Raffel, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *arXiv,* 2019.

[10] N. C. J. W. X. L. M. G. Qiushi Sun, "TransCoder: Towards Unified Transferable Code Representation Learning Inspired by Human Skills," *arXiv,* 2023.

[11] D. G. S. R. J. H. A. S. A. B. C. C. D. D. D. J. D. T. G. L. L. Z. L. S. L. Z. M. T. M. G. M. Z. N. D. N. S. S. K. D. Shuai Lu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *arXiv,* 2021.

[12] D. G. D. T. N. D. X. F. M. G. L. S. B. Q. T. L. D. J. M. Z. Zhangyin Feng, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *arXiv,* 2020.

[13] K. C. Y. B. Dzmitry Bahdanau, "Neural Machine Translation by Jointly Learning to Align and Translate," *arXiv,* 2014.

[14] P. a. J. P. a. J. H. a. K. G. a. M. A. a. G. V. Jana, "CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution," *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI),* pp. 4011--4018, 2024.

[15] B. S. D. R. J. Z. Q. Z. Y. M. G. L. Y. L. Q. W. T. X. Hao Yu, "CoderEval: A Benchmark of Pragmatic Code Generation with," *arXiv,* 2024.

[16] W. U. A. a. M. G. R. T. a. S. C. a. K.-W. Chang, "AVATAR: A Parallel Corpus for Java-Python Program Translation," *arXiv,* 2023.

[17] Y. L. N. G. M. G. A. M. O. L. V. S. L. Z. Mike Lewis, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," *arXiv,* 2019.

[18] K. a. R. S. a. W. T. a. Z. W.-J. Papineni, "Bleu: a Method for Automatic Evaluation of Machine Translation," *Association for Computational Linguistics,* pp. 311--318, 2002.

[19] D. L. S. L. D. T. N. S. M. Z. B. a. S. M. S. Ren, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," *arXiv,* 2020.