TUM Data Innovation Lab

Munich Data Science Institute (MDSI)

Technical University of Munich

&

msg-life central europe gmbh

Final report of project:

Program Code Generator using SotA Transformer Models

Authors	Heba Abdelrazek, Jannes Elstner, Rebecca Radebold, Mohamed Saleh
Mentor	Dr. Osama Hamed
Co-Mentor	Dr. Alessandro Scagliotti
Project Lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (MDSI)

Aug 2023

Abstract

Data migration is a multifaceted and labor-intensive endeavor undertaken by companies in transitional periods whose operations require large amounts of data. The process involves a number of steps including the extraction, transformation, and transfer of data from one system to another. In recent years, the rise of artificial intelligence, particularly large language models (LLM), has allowed for the potential of large-scale automation of these processes, leading to increases in both efficiency and precision. In this project, the team was tasked by msg-life central europe gmbh with automating the transformation process with the use of transformer-based LLMs.

To this end, a large-scale literature review was conducted examining a variety of these models to gauge their relative performance and to assess their applicability to the project, both in terms of their design and their feasibility for implementation. The models were divided into baseline models and state-of-the-art models. Parallel to this process, the data provided by msg-life was examined and pre-processed in order to increase its usability by eliminating irregularities and standardizing aspects including language and code. A final selection of models was achieved by conducting a zero-shot comparison of the models examined in the literature review using the pre-processed data.

After identifying the most suitable models, several rounds of fine-tuning were conducted on the supplied GPU server. CodeT5, CodeGen, PolyCoder, and InCoder, all pre-trained LLMs, were downloaded and trained in multiple rounds on a subset of the pre-processed data. Their performances were assessed using a number of metrics, including but not limited to CodeBERTScore, METEOR, and ChrF, allowing for a systematic comparison between different rounds of training as well as relatively between models. Additionally, the models were trained on various versions of the dataset, which was enhanced in multiple rounds of processing.

Following the completion of the training and evaluation of the models, the results were examined and a clear divergence in performance emerged. Both the baseline model, CodeT5, and a subset of the SotA models, CodeGen and PolyCoder, delivered excellent results deemed very promising for the task of automating data migration. On the other hand, InCoder delivered subpar results, excluding it from consideration and prompting a discussion of the underlying reasons. In the end, PolyCoder was selected as the most suitable model for the task of C++ code generation in the context of data migration, due to its superior performance as well as its feasibility for large-scale implementation. A number of improvements to the fine-tuning process which would potentially lead to enhanced results are discussed as suggestions for future work.

Contents

1 Introduction

In the following sections, we give a formal statement of the problem and a brief overview of the project objectives. Thereafter, we discuss existing potential solutions in the form of current state of the art (SotA) code generation models, describing their basic features and using available data for a basic comparison between them.

1.1 Problem Definition and Project Objectives

Data migration is a complex process conducted by bodies, often governments or private companies, whose operations rely heavily on large amounts of data. Typically, the data migration problem consists of an initial and final data configuration as the input and an efficient method of transferring it between configurations as the output (Anderson 2010). In particular, the process can be broken down into the selection, preparation, extraction, transformation and permanent transfer of data from one repository to another (Morris 2012). This project focused on the *transformation* aspect of the data migration process in the context of msg-life, a company focusing on software for life insurance enterprises.

Given the extensive scale of data required for insurance companies to manage individual profiles and create elaborate, complex models for insurance-related phenomena, the migration of said data is labor-intensive and time consuming. As such, msg-life is often tasked with automating the process to enhance both the efficiency and accuracy of the procedure. The recent rise of artificial intelligence has provided software engineers with an opportunity to fully automate this process in a way that is independent of the individual task at hand, due to the ability of large language models (LLM) to learn transformation rules and other details specific to particular cases. The goal of the project was thus to implement such a process with the use of transformer-based LLMs with the potential of significantly improving the efficiency of the data migration process as a whole.

The central objective of the project was to produce a LLM capable of producing C++ code from natural language (NL) text and pseudocode such that the transformation step of the data migration process could be fully automated. This task was divided into two secondary points. First and foremost, the team was tasked with identifying the most promising models through a *systematic comparison.* To this end, one baseline model and a variety of SotA models used for code generation were researched, before a subset of them was implemented. To compare their performances, inference was conducted and the results were evaluated using a set of diverse, yet appropriate metrics. Secondly, in order to insure the viability of these models as solutions to the given problem, the best-performing models were fine-tuned using data supplied by msg-life, drastically enhancing the performance of some. The outcome of the project takes the form of a single, optimized LLM capable of producing C++ code from NL prompts to greatly improve the efficiency of the transformation step, as well as the data migration process as a whole.

1.2 Existing SotA Methods

While examples of transformer-based approaches to data migration appear to be entirely nonexistent in the sphere of published work, a large number of models designed for the task of code generation based on NL prompts or descriptions have emerged in the last five years. By treating code as a language sequence, neural sequential architectures, including transformers, can be utilized for the task of program synthesis (Zheng et al. 2023). The ability to scrape larger and larger amounts of data in the form of text-code pairs means larger training sets along with the general, rapid development of the field of LLMs has led to increasingly impressive results in the area of code generation. Earlier models include CodeBERT (Z. Feng et al. 2020) and CodeT5 (Kusupati and Ailavarapu 2022), which have since given way to superior models, in terms of relative performance, including CodeGen (Nijkamp et al. 2023), CodeGeeX (Zheng et al. 2023), PolyCoder (Xu et al. 2022), and InCoder (Fried et al. 2023). These models are currently considered SotA models in the task of code generation and are characterized by fairly similar architectures. We give a brief overview of these models, highlighting their similarities, before discussing their relative performances including comparisons with older models.

CodeGen is set of autoregressive language models developed by Salesforce for the task of code generation. These models vary in size from 350M to 16B parameters. The greatest contribution of CodeGen, on which the success of the model relies, is a multi-turn approach, which effectively breaks lengthy, complex specifications into subprograms, allowing the model to generate higher quality results for smaller portion of the input before joining them back together. The CodeGen architecture itself consists of standard transformer decoder with left-to-right causal masking (Nijkamp et al. 2023). Of similar scale, CodeGeeX is a 13B parameter code synthesis model, pre-trained on 23 programming languages, including C++. Just as with the CodeGen models, CodeGeeX was designed with a GPT-style architecture, namely as decoder-only autoregressive model (Zheng et al. 2023).

PolyCoder, another multilingual model with 2.7B parameters, is again designed with a GPT-style architecture and trained on over 249GB of data on 12 different programming languages. This model shows particular promise in the generation of C++ code, outperforming the largest language models in this task at the time of publication in 2022. PolyCoder was able to achieve lower perplexity than the rest of the models and seemed particularly great in C++ and C generation which makes it a good candidate for our project. Finally, InCoder, released by Facebook in 2022, is a set of 1.3B- and 6.7B-parameter code synthesis and infilling models. Unlike the previously discussed models, InCoder's architecture is based on the dense model architectures described in (Artetxe et al. 2021), allowing it to perform both the code synthesis and infilling tasks with exceptional results. The table below summarizes the performance of each of these models (Xu et al. 2022; Zheng et al. 2023).

While these models represent a subset of the current SotA models available for the task of code generation, they are not necessarily the ones implemented in the project. This is largely due to the massive amount of memory required to download and train these models, which in most cases exceeded the limitations of the hardware available. As such, versions of CodeGen, PolyCoder, and InCoder with fewer parameters were selected. Additionally, it is important to note that the models discussed here are open-source, in contrast to other high-performing SotA LLMs used exclusively in private-sector settings, presenting another hurdle and selection criterion. Prior to fine-tuning the models that did fulfill the selection criteria, the data received from msg-life was cleaned and enhanced to ensure the best possible performance.

Metric	CodeGen (16.1B)	CodeGeeX (13.6B)	PolyCoder (2.7B)	InCoder (6.7B)
pass @1	16.73 %	18.40 %	5.59 %	11.33 %
pass @10	32.09 %	33.29 %	9.84 %	20.25 %
pass @100	54.39 %	54.76 %	17.68 %	38.48 %

Table 1: A summary of the performances of a selection of the SotA models CodeGen, CodeGeeX, PolyCoder, and InCoder as assessed by the pass @ k metric. The *pass @ k* metric, where $k \in \{1, 10, 100\}$ is a measure of functional correctness in which test cases are generated and models are evaluated based on how many of these test cases are solved. More specifically, k pieces of code are generated per problem, and the total fraction of solved cases becomes the output of the metric (M. Chen et al. 2021). In evaluating the results of the models chosen for the project itself, a different set of metrics was used, as the *pass @k* metric was not compatible with all of them.

2 Data

The structure and quality of the data is essential for the success of any deep learning project. We encountered numerous challenges in training models due to the unique characteristics of our data. Firstly, the amount of data was limited and often of low-quality or duplicate. Furthermore, the data was bilingual, with variables in the implementations often referencing tables or attributes from a database. Complicating matters further, msg-life uses non-standard C++ coding conventions through custom variables and functions. Before delving into how we overcame these challenges, we will start with a brief overview of the data structure.

2.1 Data Description

The most important elements of our data are the description, transformation rule, standard code, and tree code. Table 2 showcases two representative examples of the data. To better understand each element of the data, we provide detailed descriptions:

- **Description:** The description offers a broad overview of the data's context but abstains from going into specific implementation details.
- **Transformation Rule:** The transformation rule comprises pseudocode and/or a natural language description of the implementation. The transformation rule is a critical resource for our model, providing most of the necessary information for generating the implementations.
- **Standard Code:** The standard code is a high-level implementation of the transformation rule. We call it high-level because it often includes auxiliary variables referring to specific locations in the database, where lower-level code resides.
- **Tree Code:** The tree code is a lower-level and typically more detailed implementation compared to the standard code. It is derived by resolving the auxiliary variables present in the standard code and as such, it tends to be longer and more complex.

Description	Transformation Rule	Standard Code	Tree Code

Type of Indexation (e.g. premium)	If POL > 0 Then 2 Else 0	`erg = vtg.dynid;`	`if(vtg.qt1_pol.pol > 0) {`
			`  erg = 2;`
			`}`
			`else {`
			`  erg = 0;`
			`}`

The field lvId identifies the contract to which this row belongs.	lv.lvid	`strcpy(erg, vtg.lvid);`	`long long lvid;`
			`wurzel_ptr -> lvidint++;`
			`lvid = wurzel_ptr->lvidint;`
			`sprintf(erg,"%lld",lvid);`

Table 2: Two illustrative examples of the main data elements. The 'Description' provides an abstract overview of the data context. The 'Transformation Rule' describes how to implement the code via pseudocode or natural language. The 'Standard Code' and 'Tree Code' offer high-level and low-level C++ implementations respectively."

Both standard code and tree code are written in C++, but additionally feature special variables we will cover in Section 2.3.2.

2.2 Data Preprocessing

We built a preprocessing pipeline to ensure the data's quality and usability. This included cleaning the data, dealing with bilinguality, and picking which of the two implementations we wanted our models to generate.

2.2.1 Data Cleaning

The first stage of our data preprocessing involved cleaning the dataset. This included discarding rows where either no implementation was available or no transformation rule existed, since such rows do not provide adequate information to construct a high-quality input-output pair of data. We also eliminated duplicate rows, i.e. rows with both identical transformation rules and descriptions. Furthermore, we removed comments present in the transformation rules and implementations, because the comments were almost exclusively unnecessary clutter that could confound the code generation model. Also problematic were data rows with transformation rules that had no connection at all to the implementations. To mitigate the worst cases of this, we chose to remove transformation rules with lengths that vastly exceeded the lengths of both of the implementations.

2.2.2 Handling Bilingual Data

The bilingual nature of our dataset, encompassing both German and English data, posed a unique challenge to us. All data rows consisted of a German description, an English

description, and English transformation rules. The only exception were the German transformation rules from one of the German datasets.
Given the overlapping information in the descriptions, we chose to use only one of them to avoid redundancy. As a rule, we favored the German descriptions due to their tendency to be more detailed and informative. We then translated these descriptions into English using a German-to-English translation model developed by Helsinki-NLP (Tiedemann and Thottingal 2020). In cases where the German description was not available, we directly used the English description.
The translation of the German transformation rules, however, necessitated a different approach. Due to the unusual formatting and pseudocode found in these rules, we encountered occasional difficulties with the translation model. To circumvent this, we leveraged DeepL (DeepL 2023) for translating the German transformation rules, which proved more robust and reliable.

2.2.3 Identifying Target Code

Our dataset contained two types of implementations: the standard code and the tree code. Although they often matched, in cases where they did not, the transformation rule clearly referred to only a single one of them (see Table 2). We needed to identify this "correct" implementation so that we could use it as the target code, i.e. the output for the code generation.
The time needed for manual annotation was too immense given the project's time constraints, hence our first approach to solving this problem was to leverage the power of language models for automation. The most promising model we tried is Falcon-7B-Instruct (Penedo et al. 2023), a 7B parameters causal decoder-only model based on Falcon-7B and fine-tuned on a mixture of chat/instruct datasets. We posed the choice between the two implementations as a multiple-choice question and gave the model the instruction to chose the implementation that matched the transformation rule best. While this approach offered moderate success and enhanced the baseline of always selecting the tree code, it was ultimately not reliable enough, resulting in frequent failures. We suspect the reasons for failure to be a combination of the unusual task, unique code, and the limited ability of current open-source language models. On the other hand, we suspect that models with ability of InstructGPT (Ouyang et al. 2022) could have easily solved this problem.
We ended up deciding between the two implementations using a series of heuristics, which were developed by identifying patterns and trends within the data. For a small percentage of particularly challenging cases, we employed manual annotation. Throughout this process, we also identified and removed some data where neither implementation accurately fit the transformation rule, further refining our dataset.

2.3 Problem Tractability

To increase the performance of our code generation model, we employed various strategies to improve the tractability of generating the target code from just the information contained in the description and transformation rule. Notably, these included enriching the input data and simplifying the target code.

2.3.1 Augmenting Transformation Rules with Database Variables

The variables in the implementation often refer to specific tables or attributes in the database, and a key factor affecting tractability is the fact that such variables are often not specified in the transformation rule. For example, the tree code in the first row of Table 2 contains the variable `vtg.qt1_pol`, which is integral to the implementation, yet specified in neither description nor transformation rule. The lack of specification for such variables is problematic, since such variables act as database keys that have to be generated exactly, rather than variable identifiers that can be chosen freely.
To address this issue, we chose to extract variables that reference the database and add them to the input data. In the case of our data, these variables coincide with the set of variables that are used prior to their explicit definition (similar to environment variables), which we identified using a series of regular expressions. We did not extract variables that are defined in the code, since their identifiers are arbitrary, e.g. naming an index variable `i` or `j`.
The extracted variables were then appended to the end of the transformation rule, labeled with a marker. To illustrate, for the tree code in the first row of Table 2 we would add `"Env: vtg.qt1_pol, pol"` to the end of the transformation rule. Note that `vtg` has a special meaning, which is why we do not add it as a separate variable (more in Section 2.3.2). Our inspiration for this approach came from the input formatting used in CodeT5 during its fine-tuning for code synthesis (B. Chen et al. 2023), wherein environment variables used in a function are appended to the end of the natural language description input.
At this point, we want to note that even after adding missing variables to the input, the information in the input was sometimes still insufficient for the generation of the target code. However, addressing this underspecification is not straightforward and would require a comprehensive overhaul of the transformation rules. We decided against such an overhaul because of time constraints and the intent to maintain the structure of the transformation rules, to minimize inconvenience for the msg-life employees creating them.

2.3.2 Code Simplification

The target code in our data is mostly written in standard C++. However, a distinctive characteristic is the inclusion of special functions and variables, so-called *MigSys* variables, leading to something we will refer to as msg-flavored C++.
These added variables present a unique challenge as they are not part of the pre-training data fed into our models. Consequently, our models would have to learn the meaning of MigSys variables from scratch during the fine-tuning phase. To better harness the capabilities of our pre-trained models, we adopted a different strategy. During the training process, we transformed the msg-flavored code to follow common coding conventions, e.g. by replacing variable names or the way a function is called. This adaptation increases the utility of pre-training and allows the model to process and learn these functions more efficiently. Crucially, the code simplifications are not permanent and can be reverted. This ensures that after the code generation phase, we can convert the code back to its original msg-flavored C++ format, thereby maintaining fidelity to the original code.
To offer a clearer understanding of the code simplifications, Table 3 presents a comprehensive overview of all the code simplifications we employed.

MigSys Variable	Description/Meaning	Original Code	Simplified Code
`vtg`	German: *Vertrag* ≙ *policy*, pointer to the current policy, the database root table	`vtg.Var`	`policy.Var`
akt	German: *aktuell* ≙ *current*, pointer to the current table which the following variable belongs to	`akt.Var`	`current.Var`
anz	German: *Anzahl* ≙ *cardinality*, returns the cardinality, i.e. the number of instances in the database, of the variable it precedes	`anz_Var`	`Var.size()`
idx	Returns the database index of the variable it precedes	`idx_Var`	`Var.index()`
erg	German: *Ergebnis* ≙ *result*, variable to which the result is assigned	`erg = 0`	`result = 0`
Fehler	German: *Fehler* ≙ *Error*	`Fehler()`	`throw std::runtime_error()`

Table 3: Simplifying the code for improved model training by transforming MigSys variables to standard C++ syntax. The column 'MigSys Variable' lists the original variables and functions in the msg-flavored C++ code. 'Description/Meaning' provides translations from German and describes the functionality of each variable or function. The 'Original Code' and 'Simplified Code' columns show how the MigSys variables are simplified in the code.

3 Model Selection and Performance Metrics

In the following sections, we discuss both the models selected for the project as well as the metrics used to compare them. We begin with an extensive discussion of each of the models, complete with argumentation for selection, before delving into the individual evaluation metrics deemed appropriate for the project as well as a discussion of their relative applicability.

3.1 Selected Models

In a preliminary literature review, a number of baseline models, including CodeBERT (Z. Feng et al. 2020) and CodeT5 (Wang et al. 2021) along with graph-based transformer models, such as TreeGen (Sun et al. 2020), and SoTA models, including CodeGen (Nijkamp et al. 2023) and InCoder (Fried et al. 2023), were reviewed and evaluated based on two criteria. Firstly, each of the models' performances was taken into account as measured by a variety of metrics (see the next section for details), such as BLEU and ROUGE scores, both in absolute and relative terms. While metrics were not consistent across literature, meaning different papers often chose differing sets of metrics and only compared their models to a subset of the other ones under consideration, the team was able to draw relatively systematic conclusions related to performance in the code-generation task based on available comparisons and evaluations. Secondly, the feasibility of implementation was examined for each of the models at hand. While some models performed exceptionally well, they were often either too large in size for the hardware available to the project or not open source, such as CodeGeeX (Zheng et al. 2023). Such models had to be excluded from consideration, leaving a handful for the team to choose from. Ultimately, CodeT5 was chosen as the baseline and CodeGen, PolyCoder (Xu et al. 2022), and InCoder were chosen as code-generation models to be implemented fully. In the following paragraphs, we give a brief overview of the specifics of each model as well as the rationale behind choosing it.

3.1.1 CodeT5

The existing methods for NLP pre-training on source code typically treat the code as a sequence of tokens, similar to natural language. However, this approach overlooks significant structural information, which is essential for a comprehensive understanding of its semantics. To address this issue, the authors propose CodeT5 (Wang et al. 2021), an encoder-decoder model that incorporates the token type information in code. CodeT5 aims to capture more code-specific features by utilizing token type information, particularly identifiers such as function names and variables. Identifiers are considered one of the most programming language-agnostic features and retain crucial code semantics. The pre-training phase of CodeT5 (ibid.) involves using both programming language-only and NL-programming-language input. The model is optimized using three loss functions with an equal probability: masked span prediction, identifier tagging, and masked identifier prediction. Although numerous recent works have built upon CodeT5, the performance gap between them is often minimal compared to the more significant gap between CodeBERT and CodeT5. As a result, CodeT5 is frequently employed as a benchmark model to assess the effectiveness of various models and approaches. Consequently, we also utilize CodeT5 as our baseline model to evaluate the efficacy of our proposed solutions.

3.1.2 InCoder

InCoder (Fried et al. 2023) is a generative model for code infilling and synthesis that is trained on a large corpus of permissively licensed code. InCoder is the first large generative code model that is able to infill arbitrary regions of code, making it capable of performing program synthesis and editing tasks such as type inference, comment generation, and variable renaming. The model is decoder-only and causally-masked, which enables it to infill arbitrary spans of text. By conditioning on bidirectional context, InCoder significantly improves performance on these tasks, while still performing comparably on standard program synthesis benchmarks compared to left-to-right only models that were pre-trained at a similar scale. InCoder can handle a variety of tasks including type inference, comment generation, variable re-naming, program synthesis, and code infilling. The model architecture is a decoder-only causally-masked language model, similar to GPT.

InCoder is trained on a large corpus of code, which allows it to learn patterns and structures in code that are common across different programming languages and domains. This makes it more robust and adaptable to different coding tasks. Furthermore, the ability of InCoder to condition on bidirectional context substantially improves its performance on these tasks by allowing it to take into account both left and right contexts when generating code. This makes the generated code more accurate and contextually appropriate. As such, InCoder is considered a viable model for implementation for the task at hand.

3.1.3 CodeGen

CodeGen (Nijkamp et al. 2023) is a collection of large language models that have been trained on both NL and programming language data. The researchers behind CodeGen have investigated a multi-step paradigm for program synthesis, in which a program is divided into multiple prompts for sub-problems. This approach leads to more accurate program synthesis, reduces the search space, and allows for better specification of user intent. To test their models, a Multi Turn Programming benchmark (MTPB) was constructed, consisting of 115 problem sets with multi-turn prompts. CodeGen was evaluated on this benchmark using pass rate on expert-written test cases. The results showed that CodeGen is competitive with the previous state-of-the-art models in zero-shot Python code generation on HumanEval. The model architecture of CodeGen (ibid.) involves autoregressive transformers with the regular next-token prediction language modeling as the learning objective. Models of different sizes (350M, 2.7B, 6.1B, and 16.1B parameters) were created to compare with other models. Therefore, CodeGen is a promising model due to its multi-step program synthesis paradigm and large parameter size of up to 16.1 billion, which results in superior performance in generating complex and sophisticated code.

3.1.4 PolyCoder

PolyCoder (Xu et al. 2022) is a GPT-based code generation model that was produced by Carnegie Mellon University (CMU), supporting code generation for over 12 programming languages including C++. The data with which PolyCoder has been trained exhibits greater focus on C++ and C, in which it had much lower perplexity compared to other code generation models, making it more suitable for our particular project. The training data was gathered from Github's largest repositories initially covering over 600 GB of data, before cleaning and deduplication. The data was fed into a GPT-2 tokenizer on a randomized shuffle of all the different programming languages. For training, they chose GPT-2 as the model architecture with a different bucket of model complexities from 160M parameters model to a 0.4B parameters to a 2.7B parameters model. The 2.7B model is a 32 layer, 2560-dimensional Transformer model with a maximal context window of 2048 tokens, trained with a batch size of 128 sequences (262K tokens). The 0.4B-parameter model is a 24-layer, 1024-dimensional variant trained on a context window with between 1024 and 2048 tokens. PolyCoder is trained in a causal way to perform next-token prediction based on concatenation of previous output and input pairs. Due to the large number of hyperparameters and lack of time, they could not perform a hyperparameter search, but adjusted the learning rate and the total number of steps up to 150k steps. As already mentioned in the introduction section, despite having lower performance on the pass@k metric compared with the other models, PolyCoder displayed superior performance as compared to the rest of the SoTA models regarding C++ generation with much lower perplexity. Additionally, the fact that it is open source allowed us to fine-tune PolyCoder as a SoTA model for C++ generation.

3.2 Metrics for Model Performance Comparison

In order to evaluate the performance of the selected models in the context of the code-generation task, both in absolute and relative terms, the following metrics were reviewed and chosen based on their applicability to our task. Our selection criteria correspond to the general criteria for machine-translation-metric quality, namely consistency, reliability, and generality, in particular how those criteria relate to text-to-code translation (Banerjee and Lavie 2005). We conduct an analysis of each metric and draw conclusions about its relevancy to the project based on demonstrated performance relative to other metrics, assessed suitability for the code-generation task at hand, and feasibility of implementation. In the following sections, the ROUGE, BLEU, METEOR, chrF, and CodeBERTScore metrics are discussed in detail.

3.2.1 ROUGE Scores

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) (C.-Y. Lin 2004) is a set of metrics used to evaluate the quality of machine translation outputs. It provides a quantitative measure of the overlap between the generated output and reference ground truth. ROUGE scores are computed based on precision, recall, and F1 score, which are standard metrics in information retrieval and natural language processing tasks. Precision measures the proportion of generated outputs relevant to references, recall measures the proportion of relevant information captured by the generated code, and F1 score is the harmonic mean of precision and recall. ROUGE scores are commonly computed at various levels, including ROUGE-1, ROUGE-2, and ROUGE-L.

3.2.2 BLEU Score

The BiLingual Evaluation Understudy, or BLEU score, was first introduced by (Papineni et al. 2002) as a metric for evaluating machine translation performation. The BLEU score focuses on time and computational efficiency, language independence, and high correlation with human judgement. The metric relies on two elements, namely a ”translation closeness” metric, which measures how close the given translation is to a human one, and a corpus of existing high-quality translations. The BLEU score computes the geometric mean of a test corpus' modified $n-$gram precision, calculated by comparing $n-$gram matches between candidate and reference translations, and multiplies this with a sentence brevity penalty penalizing translations which do not match reference translations in length. It is important to note that this approach is position-independent and therefore allows multiple translations of the same quality but differing word or code-snippet-orders to achieve similar scores. While BLEU metric has performed well empirically for text-to-text translation and has been used widely in the evaluation of code generation models, it is unclear whether or not the metric is well-suited to the latter task. In fact it has been shown that other metrics, such as METEOR or CHrF are more closely related to human judgement when it comes to code generation quality than BLEU (Evtikhiev et al. 2023).

3.2.3 METEOR

The Metric for Evaluation of Translation with Explicit ORdering, or METEOR, is a metric initially proposed by Banerjee and Lavie (Banerjee and Lavie 2005) based on a generalized version of unigram matching, similar to BLEU. The METEOR score itself is computed based on a number of factors including unigram precision, unigram recall, and fragmentation, the last of which measures the accuracy of the word order of the candidate translation relative to the given reference. If multiple references are given for one input, the translation is scored against each of these and the highest score is reported.

In the code generation task, word ordering is particularly useful in assessing the quality of an output since code syntax, including ordering, is significantly more strict than NL syntax. For example, the definition or initialization of a variable must appear earlier in a piece of code than its use and the order in which parameters are passed to a method is often highly relevant. For this reason, among others, METEOR appears better-suited as a metric for measuring the quality of output in a text-to-code scenario than BLEU. Furthermore, METEOR takes into account recall, unlike BLEU, making it a more suitable candidate to assess machine translation in general.

3.2.4 ChrF

The character $n-$gram F-score, or ChrF, is a metric for the automatic evaluation of machine translation output first proposed by Popovic (Popović 2015) in 2015. F-scores based on character $n-$grams have been used widely as instrumental parts of more complex metrics, but Popovic argues that they qualify as high-quality standalone metrics based on their ability to outperform other scores, such as BLEU, while retaining a simplicity not experienced by their counterparts. In a series of tests comparing chrF score accuracy with other methods, including BLEU, TER, METEOR, it was shown that the chrF metrics outperformed its counterparts in almost all scenarios. Due to both its simplicity as well as its demonstrated superiority relative to other metrics, as discussed in (Evtikhiev et al. 2023), the chrF score is a front-runner for the project at hand.

3.2.5 CodeBERTScore

Building on BERTScore, CodeBERTScore is a code-evaluation metric which encodes both the generated tokens, as BERTScore does, and the programmatic context of the generated code. Furthermore, it uses the contextual encodings of pre-trained LLMs to calculate a soft similarity score between the tokens in the generated code and the corresponding reference-code tokens. Given a candidate and reference pair, CodeBERTScore returns an array of four distinct measures, namely precision, recall, F1 and F3. Experiments conducted using CodeBERTScore for the code generation task across four different languages found that CodeBERTScore reflected both human preference and functional correctness more accurately than other metrics, including BLEU, ROUGE-1, ROUGE-2, ROUGE-L, METEOR, and chrF, particularly in C++ tasks (Zhou et al. 2023).

3.2.6 Assessment of Metrics

An evaluation of the above metrics reveals a number of advantages and drawbacks to each. A study of six metrics (BLEU, ROUGE-L, METEOR, ChrF, CodeBLEU, and RUBY) conducted in the context of the task of code-generation based on two different Python datasets, CoNaLa and Hearthstone, concluded that METEOR and chrF were best-suited to the task (Evtikhiev et al. 2023). This leads us to conclude that of the non-BLEU, NL-based metrics, METEOR and chrF are front-runners for implementation. Moreover, it is important to examine the task for which the metrics were conceived. ROUGE scores were designed for the evaluation of text summarization. BLEU, METEOR, and ChrF exist for the purpose of quantifying the performance of machine translation in the context of NL-NL conversion. As such, there are inherent characteristics of the NL-code task that these metrics simply do not take into account, such as code syntax and executibility of the code. As such, CodeBERTScore, which produces scores based on these aspects, among others, is significantly better-suited to the task at hand, making it by far the best metric for this project. While other code-generation-related scores exist, such as CodeBLEU (Ren et al. 2020), many do not support the evaluation of C++ code and must therefore be excluded. As such, the METEOR, ChrF, and CodeBLEU scores are considered in the evaluation of the performance of our models, but ROUGE and BLEU scores are added for completeness.

4 Methodology

In this section, we present the methodology employed in our research project. We begin by providing a detailed account of the fine-tuning process, followed by the procedures utilized for conducting inference on our models. Furthermore, we analyze the difficulty levels of the target code and perform a comprehensive comparison across our dataset.

4.1 Fine-tuning Process

After selecting pre-trained models for the code generation task, fine-tuning them with the given dataset is crucial to improving their performance. This process involves training the models with task-specific data to adjust them to the given task. Pre-training uses large datasets with diverse instances, teaching general language representations and patterns. However, pre-trained models may not perform as well when used directly on our dataset, especially if the dataset does not follow typical standards of code generation those models were trained using. The fine-tuning process starts by initializing pre-trained models as the underlying architecture. Subsequently, the dataset is prepared using the tokenizer, which generates input IDs and attention masks. For the CodeT5 model specifically, supervised labels are provided in order to calculate the loss. The output, i.e. target code, is then passed to the tokenizer, which generates our labels. The dataset is then mapped with the tokenization function, divided into batches, and loaded into a data loader. These steps are essential for processing the data before commencing the iterative fine-tuning process.

The fine-tuning process requires careful adjustment of hyperparameters to achieve the best performance. Hyperparameters are values that are set before training the model and and tuning them successfully relies heavily on finding the correct balance and values.

For example, the learning rate is a crucial hyperparameter that determines the step size during parameter updates. Choosing an appropriate learning rate is essential to ensuring that the model converges to an optimal solution without overshooting or getting stuck in a sub-optimal state. A learning rate that is too high may result in unstable training or missing the optimal solution, while a learning rate that is too low can lead to slow convergence or getting trapped in a local minimum. Further, the batch size refers to the number of training examples processed in each iteration. It affects the trade-off between computational efficiency and gradient accuracy. A larger batch size can expedite the training process, but may require more memory. Conversely, a smaller batch size consumes less memory but may introduce more noise in the gradient estimation. The size of the available GPU restricted the batch size to some degree, but we were still able to use up to 16, which was suitable given our dataset's small size. Another parameter worth consideration is the number of training iterations, often referred to as epochs. A higher number of iterations can allow the model to refine its parameters further, potentially improving performance. However, training for too many epochs may risk over-fitting, a phenomenon in which the model memorizes the training data instead of learning general patterns. Striking the right balance requires monitoring the model's performance on a validation set and stopping training when further improvement plateaus. For this, it is imperative to utilize early stopping method.

In order to choose optimal hyperparameters, we have referred back to the papers of our selected models, namely (Z. Feng et al. 2020), (Fried et al. 2023), (Nijkamp et al. 2023), and (Xu et al. 2022) and followed the authors' recommendations regarding which hyperparameters can be used to fine-tune our models efficiently, including batch size, learning rate, and number of epochs. After selecting those parameters we initialize a trainer module, and we can use multiple options such as utilizing Hugging Face Trainer framework or utilizing PyTorch modules. Then we provide this module with our training dataset, as well as the validation dataset. Further, we specify the optimizers used in the training, such as AdamW optimizer (Loshchilov and Hutter 2019) and other options such as logging parameters and schedulers. After the fine-tuning process halts, we can then proceed to evaluate our models using the inference step, which is detailed in the next section.

4.2 Inference Process

Subsequent to fine-tuning a model, inference is conducted to facilitate an evaluation of its performance. In order to achieve the best performance, we use the same prompt structure that the model has been given as input during fine-tuning i.e. the same prompt generation method. Next, the input is encoded using the tokenizer of the model to create the input representation required by the fine-tuned model. This involves converting the tokens into numerical representations, such as token IDs or embeddings. By doing this, the model can process and interpret the input effectively, preparing it for code generation. Once the input is encoded, it is fed into the fine-tuned model for code generation. The model processes the input using its transformer layers, which capture the context and relationships between the tokens. Based on its learned parameters and the encoded input, the model predicts the most probable code generation output. After the model predicts the output, the generated code is decoded back into natural language format. This decoding step involves converting the model's numerical representation of the generated tokens back into the corresponding textual form using the tokenizer. The decoded output represents the final generated code based on the provided input prompt. We have created a standardized inference notebook that carries all the steps mentioned above in a smooth, efficient manner. An evaluation section was integrated in the notebook to evaluate the results of the inference step, i.e. to evaluate how similar the code generated is to the target code. This evaluation section contains several different metrics which each has been researched to form a suitable comparison between the selected models and the task at hand. Furthermore, the inference notebook was enhanced with the ability to evaluate each level of difficulty of target code individually to form a more coherent understanding of the models' performances. We elaborate on this in the experiments section.

4.3 Evaluation Across Code Difficulties

One challenge made apparent from early experiments is that the trained models performed well on easy examples, yet failed in accurately generating more complex code. Generating easy implementations is not the central goal of the project, since it does not require an AI solution. To better evaluate how the models perform on different difficulty levels, we chose to classify the target code into three difficulty levels: easy, medium, and hard. The difficulty classification is based on the number of lines in the implementation. While in general this is an imperfect measure of complexity, for our specific dataset, it showed strong alignment with the assessments given by our team members and supervisors.
The majority of the original dataset falls into the 'easy' category, constituting 87.33% (2018 samples), while 'medium' and 'hard' categories comprise 6.66% (154 samples) and 6.01% (139 samples) respectively. The high prevalence of 'easy' samples suggests that our code generation models may inherently develop a bias towards generating simpler code. Their effectiveness in accurately generating more complex code could thereby be limited, as the models might not be adequately trained on enough 'medium' and 'hard' examples. To study this, we compute the evaluation metrics on each difficulty separately in many of our experiments.

5 Experiments

Our experiments were conducted utilizing the GPU server prepared by msg-life. The server hosts a *Nvidia Quadro RTX 6000* GPU, offering 24GB of VRAM. In the following sections, we detail the experimental processes and results for each of the selected models.

5.1 The Baseline Model: CodeT5

For our model experiments, we utilized CodeT5 as the baseline model, which means that we measured the performance of all our other models against this well-established, robust architecture. This provides us with a reference point to evaluate the effectiveness and efficiency of the more complex models we tested. Throughout the project, numerous experiments were conducted with CodeT5, exploring various prompts and assessing the impact of data improvements. All of the various fine-tuned versions of CodeT5 utilized the same training parameters and setup on the GPU. We trained with an early stopping strategy based on the convergence of the validation loss, ensuring efficient training by halting the process once the model ceased to show significant improvement. For fine-tuning, we utilized PyTorch Lightning (Falcon and The PyTorch Lightning team 2019). We have also utilized the AdamW optimizer (Loshchilov and Hutter 2019) as recommended by the authors, and specified the learning rate, number of epochs, and warm-up steps as per the paper (Wang et al. 2021). After each of the models was been fine-tuned, we further evaluated each of them with our selected metrics where each model took as input the same prompt structure which it was fine-tuned on.

5.1.1 CodeT5: Experiments

To provide a comprehensive analysis, we begin by focusing on the first two fine-tuned models. These models were trained using the same dataset, which consisted of initially pre-processed data without the majority of data transformations mentioned earlier. Additionally, this dataset did not include the supplementary dataset that became available later in the project. The key distinction between these models lies in the input they were fine-tuned with. The first model, henceforth referred to as Model 1, was fine-tuned using a model input that solely comprised the specification or transformation rule. On the other hand, Model 2 was fine-tuned using a prompt derived from both the text description column and the transformation rule. Model 1 has stopped training after 21 epochs, while Model 2 has stopped training after only 11 epochs. Moreover, the third CodeT5 model that was fine-tuned, i.e. Model 3, was fine-tuned using the transformed data. This data also contains the additional dataset, providing access to more data points to help the trainer. Model 3 completed fine-tuning in 11 epochs. We note that the data Model 3 was fine-tuned on had a different ratio for data splits than that of the original pre-processed data. In other words, the number of training examples were reduced for the transformed data in order to accommodate a better split with the test and validation data. Hence, we have further run Model 3 to reach 21 epochs, referencing this model as Model 3.2, to compensate such a decrease and to allow it to be more comparable with other models.

5.1.2 CodeT5: Results

The first two rows of Table 4 display the evaluation scores obtained from both Model 1 and Model 2 after their fine-tuning was finalized. The results clearly indicate a significant improvement across all metrics for Model 2. Therefore, it can be concluded that for CodeT5, fine-tuning using a model input prompt constructed from both the text description and the transformation rule is more optimal. This approach yields better performance and should be prioritized in further experiments of this model. It is important to note that during our evaluation of these models, we have indeed seen score improvement for all the metrics, however, when inspecting the code generated from Model 2, we can see that for easy target code, the results have been indeed improved. However, this is not the case with medium or hard levels of difficulty of target code. This assessment will be further defined as we look into evaluating the different levels of difficulty later in this section.

Furthermore, we can see from Table 4 the results of Model 3 and Model 3.2. These models utilize the test split of the transformed data which were fine-tuned on. We notice from the results that Model 3.2 has better results than Model 3. This is implied since

	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU score	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Model 1	78.4	73.8	74.1	68.0	86.9	68.4	72.0	90.0	89.0	89.8	89.4
Model 2	81.0	**78.0**	**77.8**	**75.0**	**88.0**	**76.8**	**79.2**	**92.0**	**91.0**	**92.2**	**91.0**
Model 3	82.5	73.6	73.2	62.6	86.6	70.0	68.7	89.0	87.0	88.0	87.0
Model 3.2	**83.8**	75.0	74.3	63.8	87.5	71.0	69.8	90.4	87.8	88.7	87.9

Table 4: CodeT5 results, each Model is evaluated with the respective test file of the data split that it was fine-tuned on. Model 1 is fine-tuned on the original pre-processed data using the transformation rule only as a prompt, while Model 2 uses the text description as well. Model 3 was fine-tuned on the most recently transformed and augmented dataset, and Model 3.2 is the same but extended to more epochs.

Model 3.2 was allowed to fine-tune on a larger number of epochs, therefore capturing more of the data features. We can also note that the results of Model 3.2 have been improved across the ROUGE Score precision, however, they are not improved across the other metrics. We will further attempt to explore this performance difference. However, it is worthy to say that we have noticed an improvement in the code generated of Model 3.2, where it is more coherent and consists of much more *msg-life* flavored variables. Model 3.2 was fine-tuned on a higher-quality data. However, the metric scores have not improved compared to those of Model 2. Table 5 illustrates the distinction of the metrics scores relative to the individual difficulty levels for both Models 2 and 3.2. We can deduce from the results that Model 3.2 has better performance than Model 2 for the target code with *medium* level of difficulty, whereas Model 2 seems to be performing better than Model 3.2 for *easy* difficulty target code. As previously mentioned, Model 2 was fine-tuned using the original dataset, which consisted of more easier examples compared to the transformed dataset used for fine-tuning Model 3.2. This accounts for the increased scores specifically for the easy target code. Referring to Table 5, we observe that Model 3.2 outperforms Model 2 in all metrics for medium scores, despite both models being trained on a similar number of examples. This supports the hypothesis that Model 3.2 is capable of better capturing the dataset's features and generally performs better. Furthermore, it confirms that the transformed dataset yields superior results compared to the original dataset.

	Target Code Difficulty	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Score)	BLEU (Prec)	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
	Easy	82.9	84.0	82.4	**81.6**	**89.8**	83.2	**85.4**	**94.1**	**93.8**	**93.8**	**93.8**
Model 2	Medium	61.6	42.5	45.5	31.8	70.5	40.0	41.2	83.2	80.5	81.8	80.8
	Hard	71.3	21.8	29.4	10.4	79.5	14.5	22.6	82.7	71.7	76.65	72.6
	Easy	**86.5**	**87.8**	**85.8**	77.0	89.0	84.0	81.0	92.0	92.0	92.0	92.0
Model 3.2	**Medium**	**71.5**	**53.8**	**54.9**	**36.8**	**77.9**	**46.1**	**49.4**	**87.7**	**83.0**	**85.0**	**83.3**
	Hard	78.0	19.2	25.5	10.3	85.9	16.0	18.7	81.5	66.2	72.5	67.3

Table 5: CodeT5 Model 2 and Model 3.2 Level of target code difficulties percentage. Model 2 has exhibited better performance with easy examples, and Model 3.2 has exhibit better performance with Medium examples.

	Models	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Score)	BLEU (Prec)	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Original Test data	Model 1	78.4	73.8	74.1	68.0	86.9	68.4	72.0	90.0	89.0	89.8	89.4
	Model 2	**81.0**	**78.0**	**77.8**	**75.0**	**88.0**	**76.8**	**79.2**	**92.0**	**91.0**	**92.2**	**91.0**
	Model 3.2	67.0	57.3	59.3	47.0	79.4	68.1	52.9	85.1	81.3	82.0	81.6
New Test data	Model 1	70.8	57.9	58.5	39.6	77.9	61.3	49.8	81.8	78.0	79.2	78.1
	Model 2	75.0	64.6	64.0	51.0	82.3	67.0	58.2	84.7	81.4	82.4	81.5
	Model 3.2	**83.8**	**75.0**	**74.3**	**63.8**	**87.5**	**71.0**	**69.8**	**90.4**	**87.8**	**88.7**	**87.9**

Table 6: CodeT5, all 3 models tested with the same test split for more comparable results. New data is the transformed dataset and it contains better data quality. Model 3.2 exhibits the best performance after evaluating on it.

To further understand the performance of Model 2 and Model 3.2, we have tested all three models on the same test file, in other words, since Model 1 and 2 were using the original pre-processed data test file and Model 3 was evaluated on new data test file (transformed and augmented data), we then evaluated all three models on the same test file. In Table 6 we can see that Model 3.2 exhibits the highest performance for the new data test file, and Model 2 has the highest performance for the old data test split. This further affirms our understanding that the newer dataset is indeed richer, cleaner, and hence the Model fine-tuned on it exhibits the better performance. And puts metrics in a more comparable context than that of Table 4

5.2 InCoder

A number of experiments were conducted using InCoder. First, inference was conducted on the test dataset, supplying a baseline performance. Next, after a number of prompt styles were explored, the model was trained extensively on the training portion of the dataset and the performance evaluated once again in a subsequent inference step. Finally, the model was evaluated once again with the data classified by difficulty. These steps and their corresponding results are detailed in the following sections.

5.2.1 InCoder: Experiments

As mentioned in prior sections, inputs for inference and fine-tuning are identical for consistency purposes. However, identifying an optimal structure for the combination of NL description and transformation rule to serve as a prompt posed a challenge. A number of prompt structures were explored, and ultimately two prevailed. Supplying just the rule performed best and a simple concatenation of rule and NL prompt performed only slightly inferior. Since the dataset contained points in which the *rule* element was blank, the former approach proved less applicable than the latter. Inference was conducted by comparing the output of the pre-trained model, prior to fine-tuning, given the NL-transformation-rule prompt with the target code using the metrics outlined in the previous section.

Next, following the splitting of the data, the model was fine-tuned on the training subset using the HuggingFace Trainer. A batch size of 16 was chosen and the Adam optimizer was specified, for which the β_1 and β_2 values were set to 0.9 and 0.98 as per the original paper describing the pre-training of InCoder (Fried et al. 2023). The model was fine-tuned a total of 25 epochs, with inference conducted along the way to monitor its performance across training stages. The entire training process was conducted on the GPU and required an excess of 24 hours to complete. Following the completion of the training, inference was conducted once again in a manner identical to the one outlined above so as to preserve the consistency of the process. Finally, this process was repeated with the data which was split into *easy*, *medium*, and *difficult* subsets. The results can be found in the following section.

5.2.2 InCoder: Results

The results of the inference conducted before, during, and after fine-tuning can be found in Table 7. At first glance, it is clear that the pre-trained version of InCoder, while not displaying overwhelmingly strong performance, particularly as compared to the baseline, still demonstrated potential as a suitable model for the task at hand. However, surprisingly, the performance of the model dropped noticeably across all metrics after the first six epochs of fine-tuning and only recovered slightly after the last five epochs. This counter-intuitive behavior, while not advantageous to the goals of the project, is a well-documented, though narrowly understood phenomenon. (Kumar et al. 2022) argue that fine-tuning can distort pre-trained features leading this method to under-perform relative to other transfer-learning methods, a phenomenon which they argue cannot be avoided with early stopping. As such, it appears that the pre-trained version of InCoder, though not ideal, was better-suited for code-generation in the context of the project than fine-tuned versions. While the results improved slightly after 20 epochs, they did still did not surpass those of the pre-trained model, leading the team to conclude that the model did not display enough promise for further training or implementation in the framework of the project.

	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU score	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Pre-trained	2.89	23.39	2.85	3.79	0.05	3.87	3.95	53.72	64.32	58.08	62.84
2 Epochs	1.76	22.05	2.37	1.84	0.03	3.19	2.55	47.27	61.13	52.61	58.99
6 Epochs	1.76	22.05	2.37	1.84	0.03	3.01	2.55	47.25	61.13	52.60	58.98
20 Epochs	2.18	23.25	2.66	2.22	0.04	3.02	2.88	47.90	61.38	53.07	59.28
25 Epochs	2.16	23.25	2.68	2.32	0.03	3.01	2.89	47.84	61.41	53.05	59.30

Table 7: The results of inference conducted throughout the fine-tuning process of InCoder shows a clear decrease in performance across all metrics following the first few epochs, before displaying a slight increase after epoch 20. Even so, the results remain poor relative to the baseline model, CodeT5.

To further understand the shortcomings of InCoder, inference was conducted on the segmented data, where data points were classified as *easy*, *medium*, or *hard* cases and the performance of the model was evaluated within these categories. The results can be found in Table 8. Interestingly, both the pre-trained and the fully-trained versions of the model displayed a clear and staggering increase in performance in harder categories, with the *hard* category scoring well above the others in almost every single case. The most reliable metric, CodeBERTScore, displayed a clear, yet less drastic increase in performance, relativizing this jump. Even so, the difference in performance is remarkable and can perhaps be attributed to the increased amount of context available in longer prompts and pieces of code.

	Target Code Difficulty	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Score)	BLEU (Prec)	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Pre-trained	Easy	0.63	23.16	1.21	0.0	1.08	2.34	2.32	46.28	61.57	52.68	59.52
	Medium	5.96	29.36	9.30	0.01	6.72	7.12	7.80	59.67	66.96	62.98	66.09
	Hard	17.27	28.70	19.90	0.43	21.23	10.41	13.51	65.39	68.89	67.05	68.50
25 Epochs	Easy	0.39	23.20	7.12	0.0	0.62	1.30	1.53	41.12	59.95	48.50	57.16
	Medium	3.18	**25.53**	4.30	0.0	3.11	4.88	4.29	51.03	65.06	56.89	63.17
	Hard	**19.10**	23.90	**17.09**	**1.87**	**16.62**	**7.89**	**9.75**	**62.17**	**67.36**	**64.46**	**66.72**

Table 8: The results of inference conducted on the pre-trained and fine-tuned versions of InCoder with data separated into *easy*, *medium*, and *hard* data points. Interestingly, both models exhibit far superior performance in the *hard* category according to almost all metrics.

5.3 CodeGen

Due to our hardware constraints, we were only able to train the 350M parameter CodeGen, but not the 2B parameter version. According to (Nijkamp et al. 2023), a DeepSpeed (DS) (Li et al. 2022) configuration is needed to fine-tune CodeGen efficiently, but this was not supported on Windows. We were able to overcome this limitation by taking several parameters of the DS configuration file and integrating them into our fine-tuning script.

5.3.1 CodeGen: Experiments

A series of experiments were conducted on the CodeGen 350M model. The initial experiment aimed to observe the convergence of the model loss using standard parameters, including default optimizers and learning rate. The fine-tuning process was carried out for 20 epochs. However, it was observed that neither the training loss nor the validation loss reached convergence. It is important to note that this was a preliminary test conducted without the DS configuration, and the expected outcome was non-convergence. Furthermore, after specifying several parameters from the DS configuration file, the model loss was converging as expected. Two subsequent experiments were conducted utilizing the modified and enhanced dataset. In the first experiment, both the text description and the transformation rule were included as components of the prompt provided to the model fine-tuned, i.e. Model 1. Following the evaluation of this experiment, we noticed that using the text description as part of the model input generated noisy results with CodeGen, to the extent that 46.2% of the test samples do not have generated code. Therefore, in the second experiment, the focus was solely on incorporating the transformation rule as pseudo code input enclosed within triple quotes, similar to a docstring, which has allowed us to reduce that percentage of non-generated code to 14.6% after fine-tuning Model 2. The first experiment was concluded at 46 epochs with early stopping configuration, while the second was concluded at 48 epochs.

5.3.2 CodeGen: Results

In Table 9, we present the performance metrics of the two models, including their evaluation against the pre-trained model. The results clearly indicate a significant performance enhancement when fine-tuning the pre-trained model with our dataset. Moreover, Model 2 outperformed Model 1, leading us to infer that, for language models geared towards code generation, utilizing pseudo code tends to yield superior performance compared to employing textual descriptions that lack explicit clarification regarding the nature of the target code. In Table 10, we present a comparative evaluation of Models 1 and 2, considering the varying levels of difficulty of the target code. It is evident that Model 2 demonstrates higher performance over Model 1, exhibiting notable improvements across the majority of our evaluation metrics. Moreover, in the context of the CodeGen model, we can deduce that while fine-tuning has led to improvements compared to the pre-trained model, it has not yet attained a performance level comparable to the baseline model.

	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU score	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Pre-trained	6.1	28.7	6.1	6.7	0.4	04.8	04.7	50.6	65.6	56.3	63.3
Model 1	34.3	29.4	27.5	**69.0**	30.6	54.1	40.5	**76.9**	72.6	73.6	72.7
Model 2	**47.3**	**50.1**	**43.6**	63.1	**32.7**	**59.9**	**44.2**	73.2	**77.7**	**74.4**	**76.8**

Table 9: CodeGen results: In this table, we present a comparison between the evaluation of the pre-trained model and our experimental findings. Fine-tuning Model 1 involves incorporating both the text description and transformation rule as part of its input, whereas fine-tuning Model 2 solely utilizes the latter. Our observations lead us to the conclusion that employing only the specification rule as the model input results in improved performance, primarily due to its resemblance to Pseudo Code.

	Target Code Difficulty	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU (Score)	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Model 1	Easy	33.2	34.7	31.5	68.5	39.5	68.5	50.1	77.1	76.2	75.5	75.9
	Medium	38.8	17.9	20.8	65.9	10.6	19.4	21.9	77.6	68.6	72.2	69.2
	Hard	37.6	11.2	13.5	71.2	6.7	15.8	14.2	76.1	62.3	67.7	63.2
Model 2	Easy	48.8	60.0	50.8	60.9	37.5	68.7	49.6	71.7	79.7	74.5	78.4
	Medium	39.4	21.8	23.2	78.8	20.6	33.6	33.3	84.0	75.9	79.1	76.4
	Hard	43.1	17.9	19.6	71.4	12.4	22.7	20.5	78.7	68.7	72.9	69.4

Table 10: CodeGen levels of difficulty comparison between Model 1 and Model 2. There is a noticeable improvement across the majority of metrics in Model 2 compared to Model 1; further affirming our earlier observation that Model 2 displays enhanced performance.

5.4 PolyCoder

Multiple rounds of fine-tuning were performed on the 0.4B-parameter PolyCoder model with varying sets of hyperparameters, including the number of epochs, learning rate and batch size. Additionally, we utilized two different types of training targets. The first had labels from the target code and the second one used causal next-token prediction. We used model checkpointing to store the models that did not overfit, similar in essence to early stopping. We also used the PyTorch Huggingface trainer with tokenization on our dataset for training and testing. After each model was fine-tuned, we conducted inference on it using the trainer prediction module to evaluate the performance and calculate the metrics.

5.4.1 PolyCoder: Experiments

We conducted multiple experiments for both fine-tuning and determining a proper prompt structure. PolyCoder was fine-tuned in 2 different ways, one using defined target code as labels and the other using next-token prediction in a causal way. As previously mentioned, identical prompt structures were used during inference and training to preserve consistency. To identify the best prompt-structure for PolyCoder, we experimented with multiple prompting structures both for input data and inference data. One type of prompting consisted solely of transformation rule, while another used only the description. A third structure used a combination of both. We concluded that inputting the rule only yielded the best results. Some lines in the dataset did not include a *rule*, but included a *description* and vice versa, so we had to make sure to remove these when creating a combination of both. Additionally, as previously mentioned, we used certain code transformations to alter the variables specific to msg-life which the generalized LM simply could not understand. Furthermore, a hyperparameter search was conducted to optimize the context window length, which was 256 and different learning rates were employed during the training, ranging from 4e-5 to 1.667e-6. We have also experimented with different batch sizes between 8 and 64. We have trained the model for 50 epochs on the GPU for a total period of 2 hours. Some models were trained on the CPU for up to 15 hours and others were trained on the GPU for much smaller amounts of time. The training loss reached 0.00589 and validation loss reached 0.00724 in the best case. Following the training, inference was conducted using the test dataset in a similar manner by passing the input data to the model to preserve the consistency using the Huggingface Trainer test module.

5.4.2 PolyCoder: Results

The following table contains the results for each experiment, displaying the improvement from one experiment to the next. The first type of experiment was performed on the data prior to its transformation, using a batch size of 64 and having target labels for the code and trained only for 3 epochs. This particular model was trained on CPU with the poorest performance amongst the other models. From this model, we focused our attention on altering some training variables to improve performance, including the batch size and the number of epochs. Since we trained the second experiment on the CPU, it could not continue training for more than 15 epochs and took 12 hours with a batch size of 16. We used the same settings from the previous experiment, but only altered those two variables. We refer to this model with those settings as Experiment 2. The results significantly improved as compared to the previous experiment. However, the performance remained poor with regards to code generation in medium and hard cases. We continued in this direction for the third experiment and did not change the training settings except for increasing the number of epochs to 30 and changing the split sizes for the test and validation dataset to be 0.15 each instead of 0.1. Several rounds of training were conducted as we were using the CPU and we used check-pointing to load the model from prior checkpoints. In other words, the 30 epochs did not happen in a continuous fashion, but rather with loading from the latest checkpoints. Training time in total took 17 hours. We can call this experiment with these settings Experiment 3. Following this experiment, the performance improved both on the metric-level and in terms of the quality of the code generation examples, where the model performed very well on the difficulty levels *easy* and *medium*, but was hallucinating in the *hard* level. After this experiment, we changed the training pattern to use causal training with next-token prediction without feeding labels. In addition, we used a batch size of 8 and a context window of 256. We also used code transformations to remove the duplicates and improve the names of variables. We completed 50 epochs continuously using the GPU, which took 2 hours to complete. These altered settings improved the results significantly both on the metric-level and in the code generation samples especially on the *hard* level which is superior to all other alternatives. We denote this version with the new settings Experiment 4.

	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU score	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
Experiment 1	3.71	26.01	5.18	7.89	2.45	5.41	4.34	45.61	48.36	44.69	44.86
Experiment 2	38.53	34.48	36.83	37.42	38.86	29.82	31.58	61.57	62.39	61.74	61.93
Experiment 3	72.15	70.45	70.78	55.89	76.16	61.71	69.03	79.43	80.52	78.49	79.31
Experiment 4	**88.23**	**87.51**	**87.83**	**81.75**	**90.79**	**83.96**	**86.32**	**95.19**	**96.02**	**95.60**	**95.93**

Table 11: PolyCoder Results: clearly the change of the training manner and data alterations have largely increased the performance of the model.

In Table 12 below we show the difficulty assessment for code chunks in the test dataset. We have done a calculation over all the metrics for all the difficulty levels to demonstrate the differences in performance. Since the model was superior and good performing, the differences between the difficulty levels were minute. We have done this comparison on only the latest model because it was superior to the others.

	Target Code Difficulty	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU (Score)	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
	Easy	91.21	90.86	91.00	86.82	95.58	88.64	90.27	96.89	96.28	95.44	94.81
Model 4	Medium	90.31	90.54	90.17	85.34	94.21	87.62	89.27	96.72	97.11	96.86	96.61
	Hard	88.01	86.98	87.12	81.23	89.73	82.68	85.43	95.12	95.97	95.26	94.8

Table 12: PolyCoder levels of difficulty comparison. There is a noticeable minute difference in performance between the different difficulty levels.

5.5 Comparative Analysis and Findings

We summarize the final performance of our models in Table 13. Our experiments uncovered distinct patterns in performance across the models. CodeT5 demonstrated a strong initial performance, with an upward trend traced back to prompt style variations and data quality enhancements. On the other hand, InCoder faltered at the onset, with its performance further deteriorating - a trend we hypothesize is attributed to the distortion of its pre-trained features during fine-tuning. CodeGen managed to score above InCoder but fell short of surpassing CodeT5. Interestingly, PolyCoder, despite an unpromising start with scores below the baseline, exhibited a consistent rise in performance throughout the experimentation process. This growth trajectory was primarily attributed to the increasing number of epochs and strategic modifications in the training pattern. Consequently, PolyCoder outperformed all other models in our analysis, making it the clear choice as the most suitable model for the task of code-generation in the context of the data-migration challenge presented by msg-life.

	R1 (Prec)	R1 (Recall)	R1 (F1)	BLEU (Prec)	BLEU score	Meteor	ChrF	CB (Prec)	CB (Recall)	CB (F1)	CB (F3)
CodeT5	83.80	75.00	74.30	63.80	87.50	71.00	69.80	90.40	87.80	88.70	87.90
InCoder	2.89	23.39	2.85	3.79	0.05	3.87	3.95	53.72	64.32	58.08	62.84
CodeGen	47.30	50.10	43.60	63.10	32.70	59.90	44.20	73.20	77.70	74.40	76.80
PolyCoder	**88.23**	**87.51**	**87.83**	**81.75**	**90.79**	**83.96**	**86.32**	**95.19**	**96.02**	**95.60**	**95.93**

Table 13: The best results, according, in particular, to the CodeBERTScore metric, of each model are displayed. PolyCoder exhibits superior performance across all metrics.

6 Conclusion and Future Work

This project undertook a comprehensive comparison of various models for code-generation tasks within a data-migration context. Our extensive set of experiments led us to the clear conclusion that PolyCoder is the best-suited model for this task. It consistently scored significantly higher across all considered metrics compared to the other models, demonstrating its effectiveness and reliability. While CodeT5 showed promise initially, it was ultimately surpassed by PolyCoder. InCoder and CodeGen struggled to match the baseline performance, revealing their limitations in the context of this study. As we move forward, we recommend employing PolyCoder for tasks related to code-generation in the context of data-migration challenges. However, this recommendation is not a conclusion, but should serve as the beginning of the next phase of research.
As for future work, back-translation (Ahmad et al. 2022) has shown promise in improving the performance of sequence-to-sequence models in code generation tasks. These models map source-code sequences to a shared multilingual space using unlabeled data, such as source code from platforms like Github. The primary objective of these models is to reconstruct original code snippets or predict missing code tokens from corrupted code snippets. However, when parallel data for a specific programming language is limited or

unavailable, back-translation can be an effective technique to augment training data. This involves training a source-to-target model alongside a target-to-source model, both trained in parallel. Bi-modal data, which includes both code snippets and docstring descriptions, has been explored in papers, but our data lacks explicit docstrings or comments. Despite this, bi-modal data may still be a worthwhile avenue for future experimentation, although it presents challenges due to the nature of our available data.
Another area of research that deserves attention is the management of large-scale models. The current state-of-the-art models for code generation tasks are typically language models with over 1 billion parameters. However, working with such models presents challenges due to hardware limitations in terms of loading and fine-tuning. To address this issue, it is recommended to explore various options in future studies. One such option is LoRA: Low-rank adapters, a technique proposed in the paper (Hu et al. 2022). LoRA aims to reduce the computational requirements of large models by introducing low-rank adapters. Additionally, quantization approaches such as 8-bit quantization can be explored, which reduce the precision of model parameters to achieve more efficient computation and memory usage. It is also worthwhile to investigate approaches that combine different techniques to manage the computational demands of large models effectively.

References

Ahmad, Wasi Uddin et al. (2022). "Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages". In: *CoRR* abs/2205.11116. arXiv: 2205.11116. URL: https://arxiv.org/abs/2205.11116.

Anderson, E. et al (Feb. 2010). "Algorithms for Data Migration". In: *Algorithmica* 57, pp. 349–380. DOI: 10.1007/s00453-008-9214-y.

Artetxe, Mikel et al. (2021). "Efficient large scale language modeling with mixtures of experts". In: *arXiv preprint arXiv:2112.10684*.

Banerjee, Satanjeev and Alon Lavie (2005). "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments". In: *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72.

Chen, Bei et al. (2023). "CodeT: Code Generation with Generated Tests". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. URL: https://openreview.net/pdf?id=ktrw68Cmu9c.

Chen, Mark et al. (2021). "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374. arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

DeepL (2023). *DeepL Translator*. [Online; accessed 15-July-2023]. URL: https://www.deepl.com/translator.

Evtikhiev, Mikhail et al. (2023). "Out of the BLEU: How should we assess quality of the Code Generation models?" In: *J. Syst. Softw.* 203, p. 111741. DOI: 10.1016/j.jss.2023.111741. URL: https://doi.org/10.1016/j.jss.2023.111741.

Falcon, William and The PyTorch Lightning team (Mar. 2019). *PyTorch Lightning*. Version 1.4. DOI: 10.5281/zenodo.3828935. URL: https://github.com/Lightning-AI/lightning.

Feng, Zhangyin et al. (2020). "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155*.

Fried, Daniel et al. (2023). "InCoder: A Generative Model for Code Infilling and Synthesis". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. URL: https://openreview.net/pdf?id=hQwb-lbM6EL.

Hu, Edward J. et al. (2022). "LoRA: Low-Rank Adaptation of Large Language Models". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. URL: https://openreview.net/forum?id=nZeVKeeFYf9.

Kumar, Ananya et al. (2022). "Fine-Tuning can Distort Pretrained Features and Underperform Out-of-Distribution". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. URL: https://openreview.net/forum?id=UYneFzXSJWh.

Kusupati, Uday and Venkata Ravi Teja Ailavarapu (2022). "Natural Language to Code Using Transformers". In: *CoRR* abs/2202.00367. arXiv: 2202.00367. URL: https://arxiv.org/abs/2202.00367.

Li, Conglong et al. (2022). "DeepSpeed Data Efficiency: Improving Deep Learning Model Quality and Training Efficiency via Efficient Data Sampling and Routing". In: *CoRR* abs/2212.03597. DOI: `10.48550/arXiv.2212.03597`. arXiv: `2212.03597`. URL: `https://doi.org/10.48550/arXiv.2212.03597`.

Lin, Chin-Yew (July 2004). "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, pp. 74–81. URL: `https://aclanthology.org/W04-1013`.

Loshchilov, Ilya and Frank Hutter (2019). "Decoupled Weight Decay Regularization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: `https://openreview.net/forum?id=Bkg6RiCqY7`.

Morris, J. (2012). *Practical Data Migration*. Bcs Series. BCS Learning & Development Limited. ISBN: 9781906124847. URL: `https://books.google.de/books?id=AZKMrGyZGTcC`.

Nijkamp, Erik et al. (2023). "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis". In: *ICLR*.

Ouyang, Long et al. (2022). "Training language models to follow instructions with human feedback". In: *Advances in Neural Information Processing Systems* 35, pp. 27730–27744.

Papineni, Kishore et al. (2002). "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, pp. 311–318. DOI: `10.3115/1073083.1073135`. URL: `https://aclanthology.org/P02-1040/`.

Penedo, Guilherme et al. (2023). "The RefinedWeb dataset for Falcon LLM: outperforming curated corpora with web data, and web data only". In: *arXiv preprint arXiv:2306.01116*.

Popović, Maja (2015). "chrF: character n-gram F-score for automatic MT evaluation". In: *Proceedings of the tenth workshop on statistical machine translation*, pp. 392–395.

Ren, Shuo et al. (2020). "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis". In: *CoRR* abs/2009.10297. arXiv: `2009.10297`. URL: `https://arxiv.org/abs/2009.10297`.

Sun, Zeyu et al. (2020). "Treegen: A tree-based transformer architecture for code generation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 05, pp. 8984–8991.

Tiedemann, Jörg and Santhosh Thottingal (2020). "OPUS-MT—Building open translation services for the World". In: *Proceedings of the 22nd Annual Conference of the European Association for Machine Translation (EAMT)*. Lisbon, Portugal.

Wang, Yue et al. (2021). "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *arXiv preprint arXiv:2109.00859*.

Xu, Frank F et al. (2022). "A systematic evaluation of large language models of code". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10.

Zheng, Qinkai et al. (2023). "CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X". In: *CoRR* abs/2303.17568. DOI: `10.48550/arXiv.2303.17568`. arXiv: `2303.17568`. URL: `https://doi.org/10.48550/arXiv.2303.17568`.

Zhou, Shuyan et al. (2023). "CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code". In: *CoRR* abs/2302.05527. DOI: `10.48550/arXiv.2302.05527`. arXiv: `2302.05527`. URL: `https://doi.org/10.48550/arXiv.2302.05527`.