Complex & Intelligent Systems
https://doi.org/10.1007/s40747-024-01373-8

ORIGINAL ARTICLE

# Leveraging pre-trained language models for code generation

Ahmed Soliman[1,2] · Samir Shaheen[1] · Mayada Hadhoud[1,3]

Received: 13 December 2023 / Accepted: 31 January 2024

**Abstract**
Code assistance refers to the utilization of various tools, techniques, and models to help developers in the process of software development. As coding tasks become increasingly complex, code assistant plays a pivotal role in enhancing developer productivity, reducing errors, and facilitating a more efficient coding workflow. This assistance can manifest in various forms, including code autocompletion, error detection and correction, code generation, documentation support, and context-aware suggestions. Language models have emerged as integral components of code assistance, offering developers the capability to receive intelligent suggestions, generate code snippets, and enhance overall coding proficiency. In this paper, we propose new hybrid models for code generation by leveraging pre-trained language models BERT, RoBERTa, ELECTRA, and LUKE with the Marian Causal Language Model. Selecting these models based on their strong performance in various natural language processing tasks. We evaluate the performance of these models on two datasets CoNaLa and DJANGO and compare them to existing state-of-the-art models. We aim to investigate the potential of pre-trained transformer language models to revolutionize code generation, offering improved precision and efficiency in navigating complex coding scenarios. Additionally, conducting error analysis and refining the generated code. Our results show that these models, when combined with the Marian Decoder, significantly improve code generation accuracy and efficiency. Notably, the RoBERTaMarian model achieved a maximum BLEU score of 35.74 and an exact match accuracy of 13.8% on CoNaLa, while LUKE-Marian attained a BLEU score of 89.34 and an exact match accuracy of 78.50% on DJANGO. Implementation of this work is available at https://github.com/AhmedSSoliman/Leveraging-Pretrained-Language-Models-for-Code-Generation.

**Keywords** Code generation · Code assistant · Language models · Marian model · RoBERTaMarian · LukeMarian

## Abbreviations

| | |
|---|---|
| RNN | Recurrent neural networks |
| Seq2Seq | Sequence to sequence |
| ASTs | Abstract syntax trees |
| BERT | Pre-training of deep bidirectional transformers for language understanding |
| RoBERTa | A robustly optimized BERT pretraining approach |
| GAN | Generative adversarial networks |
| MLM | Masked language modeling |
| LUKE | Language understanding with knowledge-based embeddings |
| AI | Artificial intelligence |

✉ Ahmed Soliman
ahmed.shokry@engl.cu.edu.eg

Samir Shaheen
sshaheen@eng.cu.edu.eg

Mayada Hadhoud
mayada.hadhoud@eng.cu.edu.eg

[1] Department of Computer Engineering, Cairo University, Giza, Egypt

[2] Department of Computer Engineering, Al-Azhar University, Nasr City, Egypt

[3] School of Computational Sciences and Artificial Intelligence (CSAI), Zewail City of Science and Technology, 6th of October City, Giza 12578, Egypt

## Introduction

Deep Learning [1] is the most interesting field in Artificial Intelligence that is inspired by biologic neural networks to recognize patterns in the real world. These neural networks have a remarkable ability to process and learn from massive volumes of data. It is used in the image, video, text, and voice analysis. The pre-trained language model is a neural network that has been trained on unlabelled text data, such as in an unsupervised, task-agnostic way, to convert a sequence of input words into a context-dependent embedding. On the other hand, there is the notion of fine-tuning the pre-trained model, which is the training of a previously initialized pre-trained model using the weights of this pre-trained language model as the beginning weights during the upcoming training of other models.

Recurrent neural networks (RNN) were proposed as the foundation for the first pre-trained language models [2]. Also, it is proved that pre-training RNN-based model on unlabeled data and then fine-tuning it on a specific task delivers better results than training a randomly initialized model on such a task directly. Two limitations exist with RNN-based sequence models. First, the processing manner of the tokens is in a sequential way which must process token after token, so RNNs don't remember the non-sequential tokens perfectly. Second, RNN-based models may fail to capture long-term relationships between code tokens.

Pre-trained language models are neural networks designed for various NLP tasks, utilizing a pre-train fine-tuning approach. They are trained on vast text corpora and then fine-tuned for specific downstream tasks [3]. These models have revolutionized NLP by providing accurate and efficient text representation.

The key advantage of using pre-trained models for code generation lies in their ability to generalize across various codebases and programming languages. Instead of starting from scratch, pre-trained models already possess a substantial knowledge base, enabling them to better comprehend and generate code in a wide range of programming paradigms. This generalization is crucial in dealing with multilingual code generation tasks, as these models can seamlessly switch between different languages without the need for extensive language-specific training.

Another significant benefit is the reduction in training time and resource requirements. Pre-trained models have been fine-tuned on vast corpora, allowing researchers and developers to transfer this knowledge to specific code generation tasks with relatively small amounts of domain-specific data. Fine-tuning a pre-trained model requires less computational power and data compared to training from scratch, which makes it more accessible to the broader community.

Furthermore, pre-trained models proved their capabilities in capturing contextual dependencies and understanding the surrounding context when generating code. This context awareness helps in producing more coherent and semantically meaningful code snippets. Code generation tasks often involve complex syntactic structures, which pre-trained models can effectively handle by learning intricate patterns and dependencies in code expressions.

Transformers [4], a neural network architecture, play a crucial role in pre-trained language models, allowing them to process input sequences of any length and handle long-range dependencies. The self-attention mechanism in transformers enables the model to weight the significance of different words or phrases, improving performance on various NLP tasks. Transformers, such as BERT, RoBERTa, and XLNet, have shown remarkable results in NLP tasks. ELMO [5] and ULMFit [6] are pre-trained language models that have demonstrated improvement on several natural language understanding tasks. Pre-training transformers work in a self-supervised manner and achieve great success when fine-tuned on downstream tasks like machine translation, question answering, and text summarization.

Pre-trained language models, empowered by transformers, have transformed the field of NLP by offering effective and efficient text representation. Their ability to be fine-tuned for specific tasks has made them a go-to solution for many NLP applications.

Several leading companies, including OpenAI, Microsoft, Google, and HuggingFace, have introduced transformer-based pre-trained language models featuring different architectures through the models. Transformer models generally fall into three categories. The first category comprises encoder-only transformers, exemplified by BERT, RoBERTa, ELECTRA, and Luke. The second type is represented by decoder-only transformers, such as GPT-2, Llama, Falcon, and ChatGPT. The third category includes encoder–decoder transformer models, like MarianMT, T5, and BART. In encoder–decoder models, the encoder processes input sequences, utilizing embedding and attention mechanisms to construct an intermediate representation. Subsequently, the decoder utilizes this representation to generate an output sequence. Notably, models like BERT and GPT have demonstrated exceptional performance with minimal fine-tuning, achieving state-of-the-art results across numerous natural language understanding (NLU) tasks.

Transformer-based language models, such as GPT-2, BERT, RoBERTa, and others, have surpassed the efficiency of traditional recurrent neural networks (RNNs). Leveraging this efficiency, these models can be pre-trained on extensive volumes of unlabeled text data. The massive pre-trained encoder–decoder models have proven to significantly enhance performance in various sequence-to-sequence applications. Massive pre-trained encoder–decoder models have been proven to greatly improve performance on a range of sequence-to-sequence applications [7, 8]. Rothe et al. [9] proposed their work to avoid costly pre-training by constructing the encoder–decoder model using pre-trained encoder and/or decoder-only checkpoints (e.g., BERT, GPT-2). This is referred to as leveraging pre-trained checkpoints.

The proposed process and pipelines for the AI code assistant system are shown in Fig. 1 and involve three distinct phases and pipelines that form a comprehensive approach to the code assistant system, encompassing code generation from natural language input to the corresponding code as output. Then linting and code analysis using Flake8, and finally the error correction and refining the generated code.

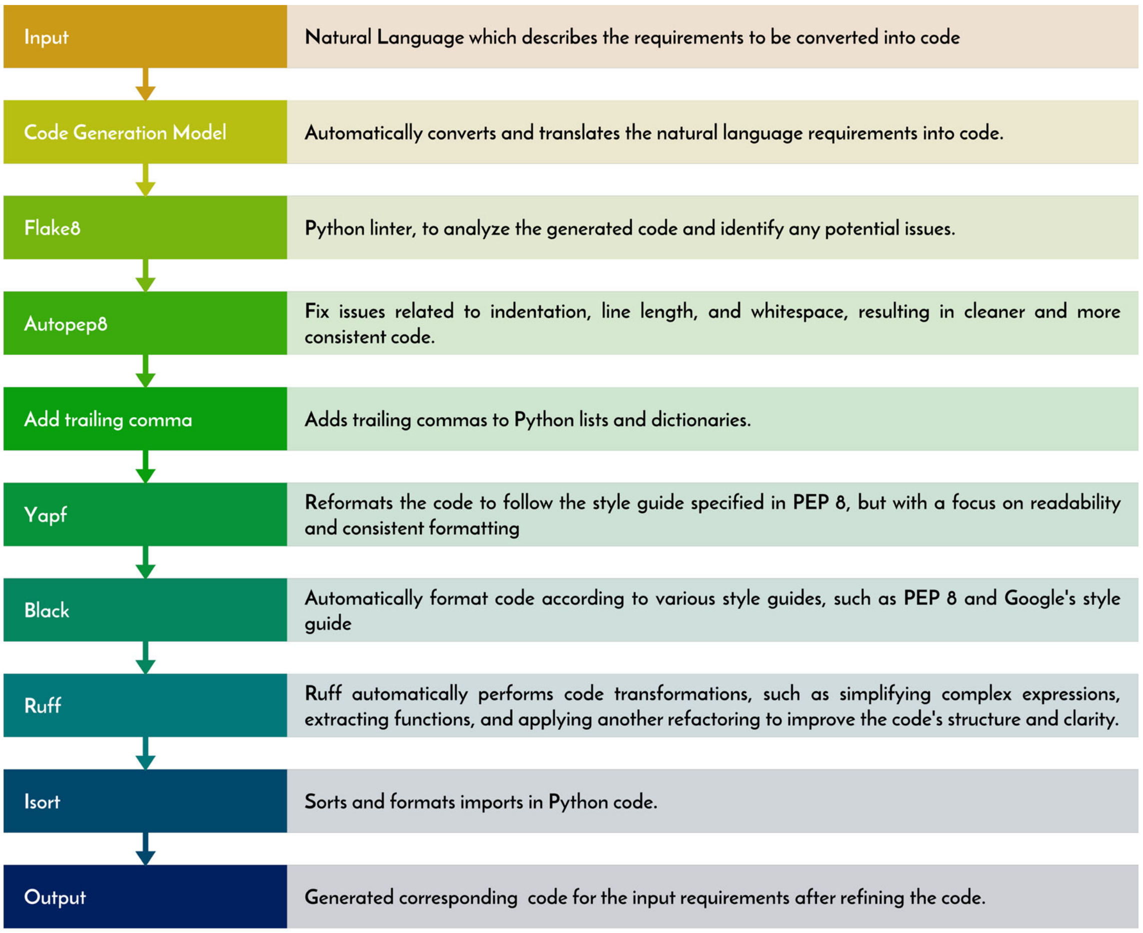

Input
Natural Language which describes the requirements to be converted into code
Code Generation Model
Automatically converts and translates the natural language requirements into code.
Flake8
Python linter, to analyze the generated code and identify any potential issues.
Autopep8
Fix issues related to indentation, line length, and whitespace, resulting in cleaner and more consistent code.
Add trailing comma
Adds trailing commas to Python lists and dictionaries.
Yapf
Reformats the code to follow the style guide specified in PEP 8, but with a focus on readability and consistent formatting
Black
Automatically format code according to various style guides, such as PEP 8 and Google's style guide
Ruff
Ruff automatically performs code transformations, such as simplifying complex expressions, extracting functions, and applying another refactoring to improve the code's structure and clarity.
Isort
Sorts and formats imports in Python code.
Output
Generated corresponding code for the input requirements after refining the code.

**Fig. 1** Pipelines for the proposed code assistant system

A significant milestone in this paper is the evolution and emergence of the transformer language models, which have demonstrated exceptional capabilities in capturing intricate language nuances. These models, such as DistilRoBERTa, DistilBERT, ELECTRA, and LUKE, have found success in various natural language processing tasks. Harnessing the power of pre-trained transformer language models in combination with the Marian Decoder for code generation represents a novel approach to further enhance precision and efficiency in navigating intricate coding scenarios. We proposed and implemented new state-of-the-art models in the code generation problem which employ the idea of leveraging pre-trained language models for sequence generation tasks to get more accurate results. Thanks to transformer models with multi-functions in the NLP field such as machine translation, sentiment analysis, classification, and other tasks. Our implementation depends on the transformer pre-trained checkpoints for the encoder and Marian Decoder from Marian Neural Machine Translation model. All proposed models in our experiments are stack of six layers in the encoder and stack of six layers in the decode architecture.

This paper makes significant contributions to the code generation problem through the following key findings:

1. Proposing new hybrid models for code generation by leveraging pre-trained language models such as BERT, RoBERTa, and Marian. We propose novel hybrid models with small sizes in the encoder and decoder to address the code generation problem. The proposed models achieve high accuracy in the code generation task on the CoNaLa and DJANGO datasets.
2. Highlighting the importance of pre-trained language models. Our research emphasizes the necessity of pre-trained encoders in sequence formation tasks. Furthermore, we demonstrate that weight sharing between the encoder and the decoder is often beneficial.
3. Conducting Error Analysis and Refining Generated Code. We conduct a comprehensive error analysis of the generated code and refine it to adhere to Python code standards. This process ensures the produced code meets the requirements of quality and consistency.
4. Ensuring code compliance and code refinement with coding standards and best practices becomes imperative by

using techniques for error analysis, syntax correction, and enhancing the reliability and maintainability of the generated code.
5. Leveraging pre-trained language models such as DistilRoBERTa with Marian Machine Translation Decoder yields a BLEU score of 35.74 and a ROUGE score of 44.25 in the code generation task.
6. Exploration of hybrid models, such as seq2seq or encoder–decoder models, as potential solutions for code assistance. These models, combining a language model encoder and decoder, offer insights into enhancing the overall code generation process.

The rest of the paper is organized as follows: section two delves into related work, thoroughly examining the literature on code generation challenges and distinguishing the proposed approach from existing methods. Section three presents the proposed hybrid models, detailing their architectures and design principles. Section four demonstrates error detection and refining the generated code using Flake8 and Python tools. Section five introduces datasets and the experimental setup, clarifying the selection process and parameters. Section six showcases comprehensive experimental results, providing insights into the proposed models' performance. Also, this section analyzes the nature and frequency of errors and warnings in generated code and conducts a thorough evaluation, comparing the proposed models against state-of-the-art approaches. Finally, section seven concludes by summarizing key findings, emphasizing the proposed models' impact, and suggesting avenues for future research.

## Related work

In recent years, the advent of machine learning-based methods, particularly neural Seq2Seq models with attention mechanisms, has shown promising results in generating code from natural language descriptions and pseudocode [10, 11]. These approaches have the potential to bridge the gap between human-readable descriptions and machine-executable code, although they still face challenges in handling variable-length code and ensuring syntactic and semantic correctness [12]. Technology-aided learning has significant contributions to the understanding of technology-assisted language learning adaptive systems, offering valuable insights for researchers, educators, and policy-makers in the field [13]. Also, Plug-ins enhance the program, add assistance to the programmer and add more advancement and software capabilities to prevent the need of the starting from the scratch when developing [14].

Furthermore, research has expanded to cater to specific application domains, such as domain-specific language code generation, model-driven engineering, code-to-code translation, and performance-driven code generation. Each of these domains comes with its unique set of requirements and constraints, necessitating tailored solutions [15]. The growing availability of large-scale code datasets has also contributed significantly to advancements in code generation. Researchers have used various datasets, ranging from Python code repositories to multilingual translation benchmarks, to train and evaluate their models [16]. Human evaluation remains a crucial component in assessing the quality and correctness of generated code, complementing traditional automated evaluation metrics [17].

Previous studies demonstrate the continuous evolution and innovation in code generation techniques, driven by the integration of machine learning, domain-specific knowledge, and large-scale datasets. However, challenges related to code quality, scalability, and adaptability to multiple languages and programming paradigms persist. There are some difficulties in this problem because the output has a well-defined structure and the domain, structure of the input, and the output are not similar. There are various techniques used in code generation, including tree-based and deep learning-based approaches. Also, semantic parsing-based techniques were used in this task. Tree-based techniques involve the use of syntax trees to generate code, while deep learning techniques use neural networks to learn the mapping between natural language descriptions and source code.

### Tree-based techniques

Tree-based techniques in semantic parsing are task-driven methods that convert natural language input into a formal, machine-executable representation. These techniques often represent code as Abstract Syntax Trees (ASTs), serving as a syntactic tree representation capturing the structure of expressions and the program's control components. Demonstrating effectiveness in code generation for specific domains over several decades [18], the goal of ASTs is to describe the semantic structure of sentences in a computer language as trees. Semantic parsers, categorized into shallow and deep semantic parsing, map natural language utterances into semantic representations, such as logical forms or meaning representations [19]. The use of tree-based methods in semantic parsing offers advantages, including the ability to address code generation challenges, enhance accuracy, and handle various data types. However, challenges arise in representing code as ASTs, particularly in managing extensive node counts and synchronicity issues in the generation process [20].

Researchers employ sequence-to-tree models for code generation, where the tree represents the AST of the target source code [18, 21–29]. These models aim to improve the code snippet creation process using ASTs. The use of tree-based methods presents multiple advantages, addressing code generation challenges by converting Natural Language input into a corresponding AST. Tree-based approaches offer visual intuitiveness, making complex predictive models more comprehensible. However, challenges include difficulties in consistently generating accurate code based on less common training data and synchronous deviations in output structure [20, 30].

Yin and Neubig [23] proposed a syntax-driven neural code generation technique, constructing an abstract syntax tree through actions from a probabilistic grammar model. Rabinovich et al. [24] introduced Abstract Syntax Networks for code generation and semantic parsing, utilizing datasets like JOBS, GEO, and ATIS. In 2018, Yin and Neubig presented TRANX [25], parsing utterances into formal meaning representations using a transition system and probabilistic models. However, TRANX exhibited incoherence issues in generation, as evidenced by a BLEU score of 24.30 with the CoNaLa dataset.

## Deep learning-based techniques

The generation of source code falls into the categories of text-to-text or sequence-to-sequence, achievable through the application of Deep Learning models for both development and maintenance. The integration of machine intelligence capable of comprehending and constructing intricate software structures holds significant potential within Software Engineering [31]. Deep learning, a subset of artificial intelligence, offers a promising solution to alleviate the challenges associated with manual code creation, demonstrating success in various domains [29, 32, 33].

Transfer Learning has proven effective in fine-tuning pre-trained models for new tasks. By adapting pre-trained models to specific jobs, consistent outcomes and findings are achieved in the seq2seq code generation task [34–37]. Deep learning techniques exhibit promise in generating code from natural language descriptions, offering benefits for complex domains while requiring less manual effort compared to tree-based techniques [38].

Various researchers have worked on code generation tasks using datasets such as CoNaLa, DJANGO, ATIS, CodeSearchNet, and others. Notable models and methods include Dong and Lapata's syntax-driven neural code generation [22], Yin and Neubig's reranking model [26], Shin et al.'s PATOIS [27], Sun et al.'s TreeGen [28], and Xu et al.'s deep learning model with external knowledge incorporation [29].

In 2021, Dahal et al. conducted an analysis of tree-structured architectures, evaluating text-to-tree, structured tree-to-tree, and linearized tree-to-tree models on constituency-based parse trees [18]. Constrained decoding of language models by Shin et al. [21] demonstrated the paraphrasing of user utterances into a regulated sublanguage for enhanced semantic parsing.

Recent contributions by Norouzi et al. [32] showcased transformer-based seq2seq models competing with or outperforming models specifically designed for code generation. Beau and Crabbé [34] proposed an encoder–decoder model using BERT as an encoder and a grammar-based decoder, achieving a BLEU score of 34.2 on the CoNaLa dataset.

## Semantic parsing based techniques

Semantic parsing, a subset of natural language processing, involves translating natural language utterances into machine-understandable logical forms. The overarching aim is to extract precise meaning, enabling machines to comprehend and execute these utterances. Its applications span various domains, including machine translation, question answering, ontology induction, automated reasoning, and code generation [39].

The Context and Variable Copying method in neural semantic parsing for code generation integrates contextual information to enhance disambiguation. Utilizing an encoder–decoder system, this approach leverages program context during decoding. A two-step attention mechanism aligns words in the language with environment identifiers, and a supervised copy mechanism replicates environment tokens, even if unseen during training [40]. Iyer et al. [41] introduced an architecture-enhancing contextually relevant code generation by incorporating programmatic context. An encoder–decoder system processes input utterances and environment identifiers, employing a two-step attention mechanism for effective word-identifier association.

The decoding process of neural semantic parsing models is significantly shaped by language structure. Techniques like sequence-based knowledge base query generation [42] and enforcing type constraints in query generation [43] highlight the importance of grammatical constraints. Ling et al. [44] presented a sequence-to-sequence code generation approach, incorporating generative and pointer models for keyword copying from input. Yin and Neubig proposed leveraging abstract syntax trees (ASTs) for coherence in general-purpose programming languages [23]. Also, Iyer et al. [45] introduced idiom-based decoding to streamline grammar-constrained semantic parsing systems. Shin et al. [27] mined code idioms to support high-level and low-level reasoning.

Another way in the semantic parsing techniques can be Sketching using Pattern Recognition and Symbolic Reasoning Sketching in program synthesis involves articulating high-level descriptions through incomplete programs or sketches. Nye et al. [46] introduced Sketching systems with a sketch generator and a program synthesizer. Shin et al. [27] in the same year incorporated mined code idioms into the grammar for unified high-level and low-level reasoning.

Through Conversational Semantic Parsing, Dong et al. [47] introduced a method to enhance the interpretability of neural semantic parsers. Their approach focused on a confidence modeling framework, which systematically characterizes different forms of uncertainty, including model, data, and input uncertainties. By integrating these confidence metrics as features in a regression model, a confidence score is generated. This score serves as a valuable tool for identifying the sources of uncertainty in predictions, thereby augmenting the model's overall comprehension. Importantly, this framework lays the groundwork for conversational programming strategies, allowing models to engage users in clarifying discussions when confronted with uncertain or missing information.

Another work from researchers [48, 49] delved into the application of conversational AI within the domain of natural language semantic parsing. This technique proves particularly advantageous in rectifying incomplete or inaccurate user requests during the parsing process. Through establishing an interactive dialog between the program and the user, conversational AI plays a pivotal role in addressing gaps and errors in logical forms. The iterative exchanges not only fill in missing details but also correct inaccuracies. Moreover, these conversational interactions contribute significantly to narrowing down the search space, resulting in more precise and reliable output predictions. Polozov et al. [50] introduced the FlashMeta framework, a neural network-independent approach for program synthesis. Parisotto et al. presented Neuro-symbolic program synthesis in 2016 [51]. DAPIP, a system for Programming-By-Example, was introduced in 2017 [52]. DeepCoder and RobustFill were proposed in the same year [53, 54].

Furthermore, programming using Reinforcement Learning. Xu et al. [55] introduced Auto Assembler, leveraging reinforcement learning for autonomous generation of assembly instructions. Finally, programming from Description using Rich Domain-Specific Languages. Semantic parsing methods like Neural Program Search [56] and Language to Logical Form with Neural Attention [22] align with program synthesis from descriptions, mapping natural language to predefined program structures [57].

This literature survey highlights the diverse techniques employed in code generation, ranging from classical approaches to advanced deep learning models. While classical methods lay the foundation, modern deep learning approaches demonstrate adaptability and effectiveness in addressing the complexities of code generation.

## Proposed hybrid models

Leveraging pre-trained models for sequence generation tasks [9], particularly in the context of code generation, has become a transformative approach in recent years. Pre-trained models are language models that have been extensively trained on large and diverse datasets to learn contextual representations of language. These models, such as GPT-3 [58], RoBERTa [59], and BERT [60], capture intricate patterns and relationships in text data, making them adept at understanding the nuances of programming languages and code syntax.

Using pre-trained models for code generation does come with a set of challenges. One significant concern is the safety and reliability of generated code. Pre-trained models can sometimes produce incorrect, insecure, or inefficient code, which poses risks in real-world applications. Careful consideration of validation and verification techniques is essential to ensure the safety and correctness of generated code.

Another challenge is the potential exposure of sensitive code or intellectual property. Pre-trained models may inadvertently memorize or expose confidential code snippets present in their training data, which can lead to privacy and security issues. Researchers and practitioners must employ robust techniques to prevent such data leakage. So, leveraging pre-trained models for code generation tasks offers a powerful and efficient approach to address the complexities of the task. Also, these models continue in many applications.

The ability of pre-trained models to generalize across languages, context awareness, and reduced training requirements make them an invaluable resource for accelerating progress in the field of automated code generation. However, responsible use, safety considerations, and privacy protection measures are critical to fully harness the potential of pre-trained models for code generation and ensure their successful integration into real-world software development workflows.

We leveraged pre-trained models to obtain various code generation models with different pre-trained encoders models combined and integrated with Marian decoder as shown in Fig. 2. Our proposed models are indicated with their encoders and decoders in Table 1, and Marian Decoder can be indicated from MarianCG paper [61].

### RoBERTaMarian model

In 2019, RoBERTa model was introduced by Facebook and it is based on Google BERT pre-trained model that was proposed before it in 2019. It was built on the BERT model by eliminating the next-sentence pretraining aim and training with considerably bigger mini-batches and learning rates. This model is identical to BERT Model, except for a minor embedding adjustment and a setup for RoBERTa pre-trained models.

RoBERTa is built on the same architecture as BERT, but it uses a byte-level BPE as a tokenizer (as does GPT-2) and a different pretraining strategy. Token type ids do not exist in the RoBERTa model. The architecture of the RoBERTa-base

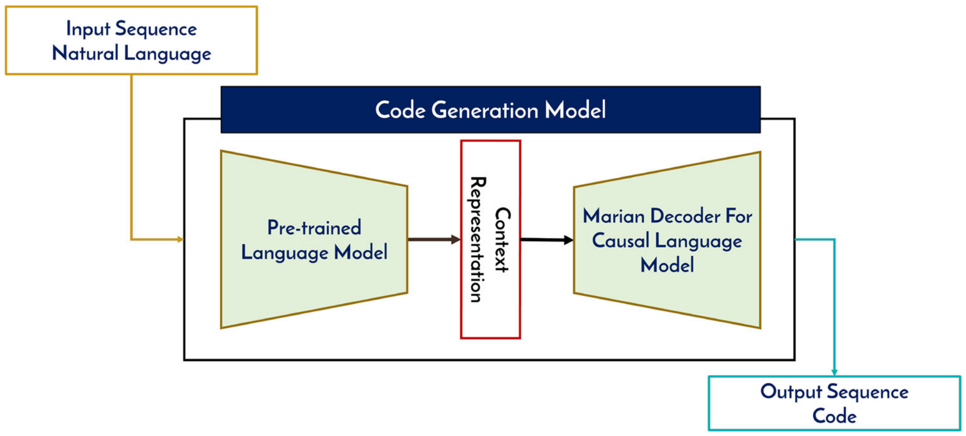


**Fig. 2** Leveraging pre-trained language models for building code generation models

**Table 1** Our proposed code generation models

| No. | Model | Model architecture | |
|---|---|---|---|
| | | Encoder model (encoder) | Decoder model (decoder) |
| 1 | RoBERTaMarian | DistilRoBERTa | Marian decoder |
| 2 | BERTMarian | DistilBERT | Marian decoder |
| 3 | ELECTAMarian | ELECTRA | Marian decoder |
| 4 | LUKEMarian | LUKE | Marian decoder |

model has 12 layers. RoBERTa is pre-trained with the MLM task (and without the NSP task).

We used the distilled version of the RoBERTa-base model which is called DistilRoBERTa and it has the training procedure as DistilBERT [62]. The pre-trained DistilRoBERTa model has 6 layers, 768 dimensions, and 12 heads, with a total of 82 million parameters (compared to 125 million parameters for RoBERTa-base). DistilRoBERTa is twice as fast as the Roberta-base model. DistilRoBERTa model distinguishes between English and english. It is a case-sensitive model. We combined DistilRoBERTa as an encoder with Marian Decoder as shown in Fig. 3.

Figure 4 shows the architecture of the RoBERTaMarian model and the encoder architecture is declared and shown in Fig. 5.

### RoBERTa pooler layer

The term “Roberta Pooler” pertains to the specialized layer within the RoBERTa model known as the pooler layer. RoBERTa represents a refinement of the BERT (Bidirectional Encoder Representations from Transformers) model, a widely used framework for tasks involving Transformer-Based Processing. Within the RoBERTa architecture, the pooler layer holds the responsibility of condensing the information amassed from the encoder layers into a consistent and predetermined representation. This resultant representation proves valuable for subsequent tasks like classification or text generation. This is achieved by processing the hidden states of the final layer as input and subsequently applying a pooling operation, often involving mean or max pooling techniques, to derive a singular vector representation. This vector is subsequently suitable for utilization within a classifier or decoder layer to facilitate predictions or text creation.

The pooler layer within RoBERTa constitutes a fully connected layer equipped with learnable weights and biases. These parameters are optimized during the model’s training phase to effectively capture the salient details from the input sequence and generate a purposeful representation aptly serving the task at hand. Typically, access to the pooler layer is facilitated through the ‘pooler_output’ attribute embedded within the RoBERTa model’s output. This attribute holds the outcome of the pooler layer’s operations, manifesting as a tensor characterized by dimensions (batch_size, hidden_size). In this context, ‘batch_size’ denotes the number of input sequences within a batch, while ‘hidden_size’ signifies the dimensional extent of both the encoder layers and the pooler layer. Notably, the incorporation of the pooler layer within RoBERTa is not obligatory and certain variations or implementations might choose to omit it. Nevertheless, in the majority of scenarios, the pooler layer is embraced due to its practical utility in generating a standardized representation of the input sequence

## BERTMarian model

DistilBERT is a student version BERT as shown in Fig. 6. It is smaller and quicker than BERT. The DistilBERT transformer

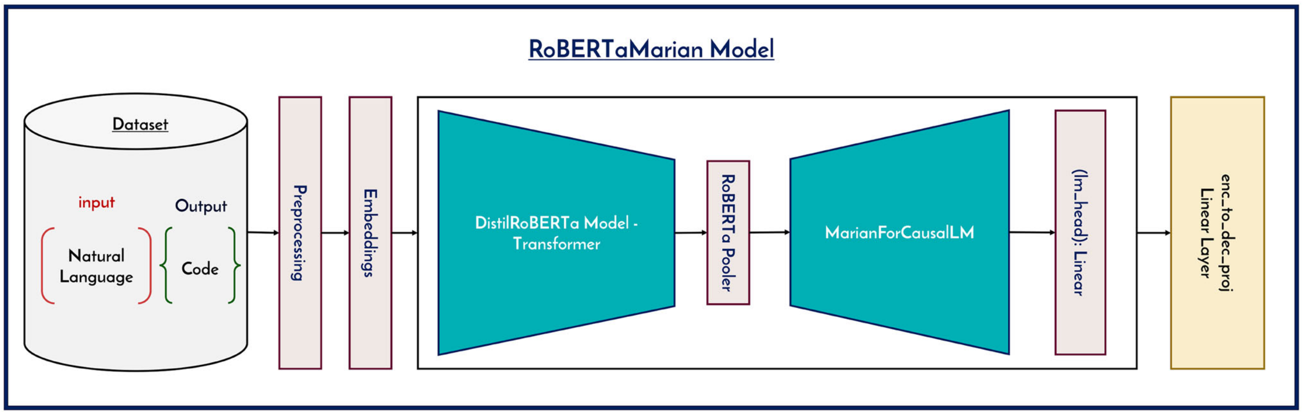


**Fig. 3** RoBERTaMarian code generation model

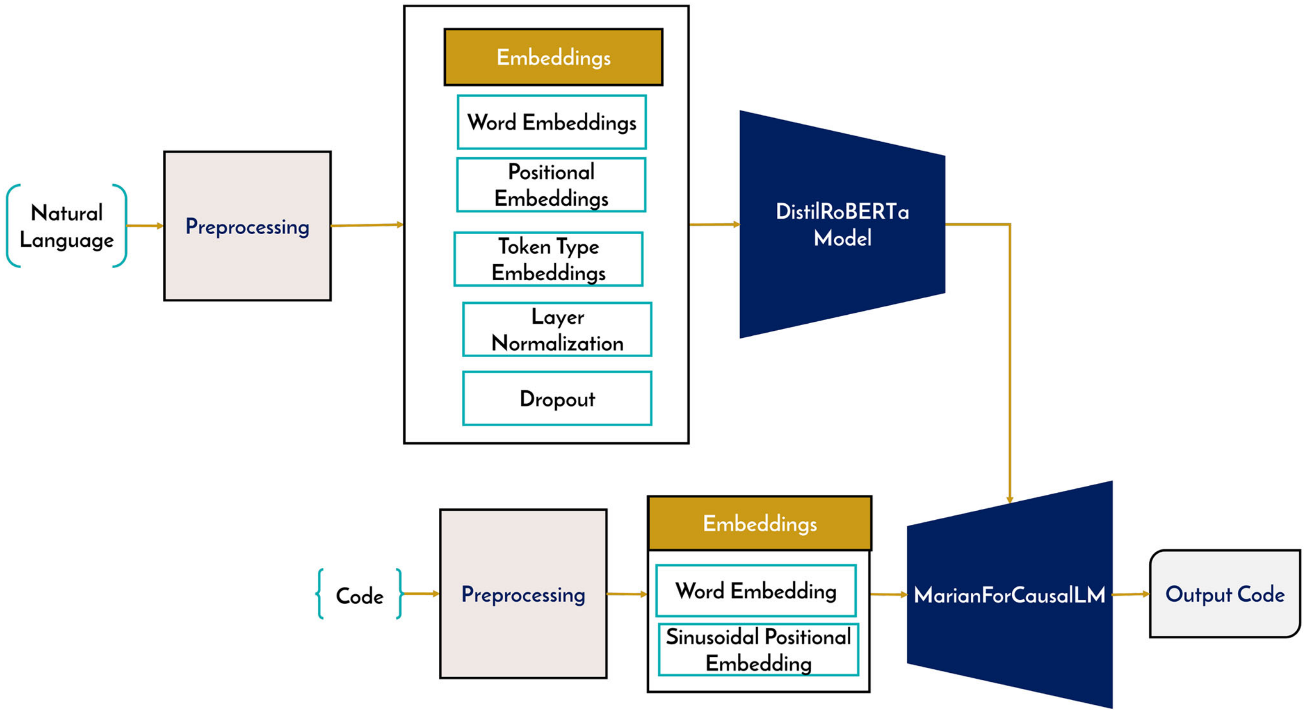


**Fig. 4** RoBERTaMarian model architecture

model was created using the same basic design as BERT. The pooler and token-type embeddings are removed, and the number of layers is reduced by a factor of two, resulting in a smaller encoder and decoder with six layers. DistilBERT proved that variations in the tensor's last dimension (hidden size dimension) have a smaller impact on computation efficiency (for a fixed parameters budget) than variations in other factors such as the number of layers.

As a result, DistilBERT is meant to focus on reducing the number of layers. It is self-supervised pre-trained on the same corpus as a teacher using the BERT base model. It used an automatic procedure to produce inputs and labels from those texts using the BERT base model. DistilBERT has 40% fewer parameters than BERT-base-uncased DistilBERT has 40% less parameters than BERT-base-uncased. It runs 60% faster while maintaining over 95% of BERT's performance on the GLUE language comprehension test benchmark.

We built a code generation model as shown in Fig. 7 that contains DistilBERT as an encoder and Marian Decoder. Figure 8 shows the architecture of BERTMarian model and the encoder architecture is declared and shown in Fig. 9

## ELECTRAMarian model

In 2020, ELECTRA [63] model was introduced by the Stanford University in collaboration with Google, and it is a novel

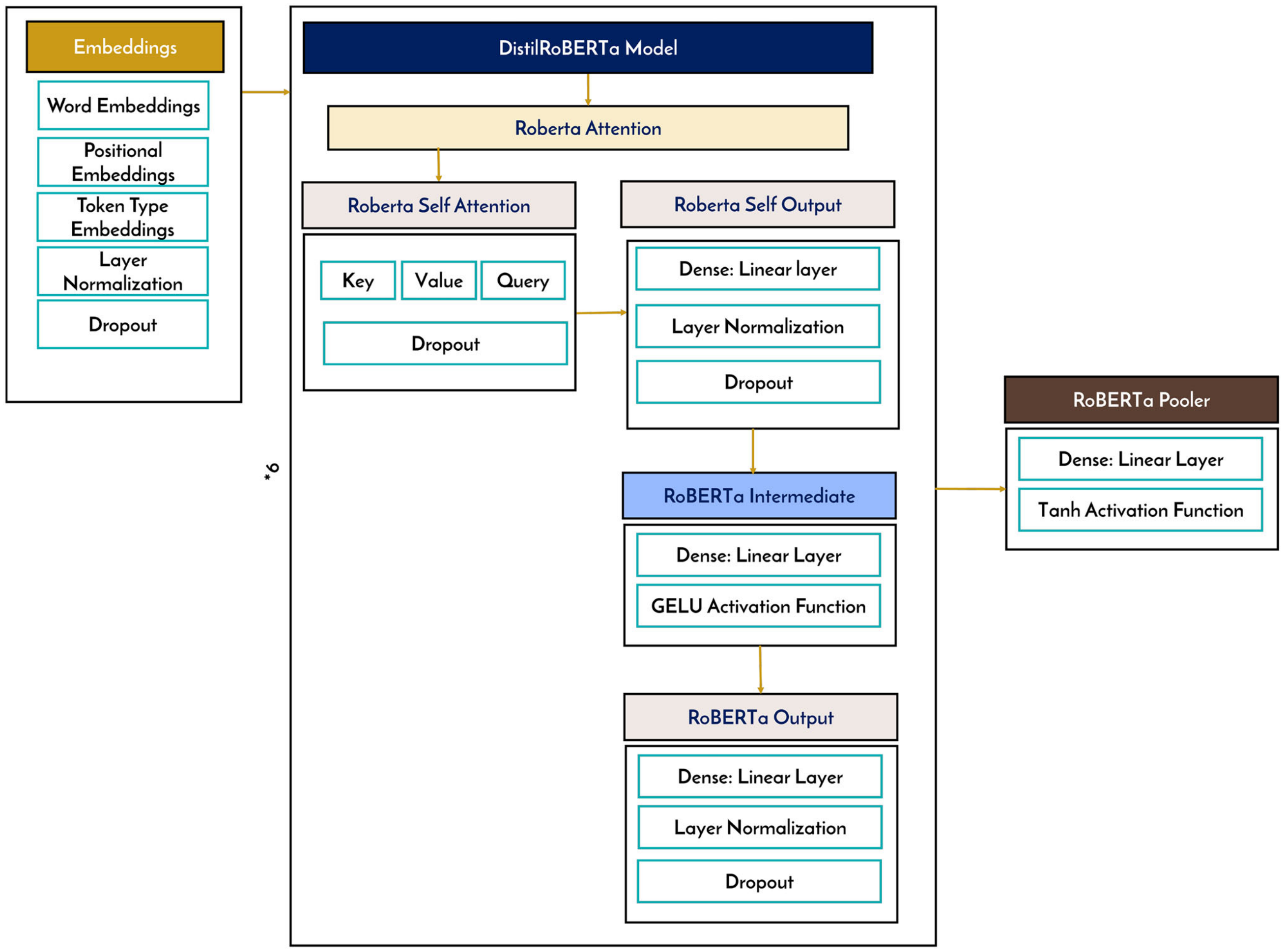


**Fig. 5** RoBERTaMarian encoder

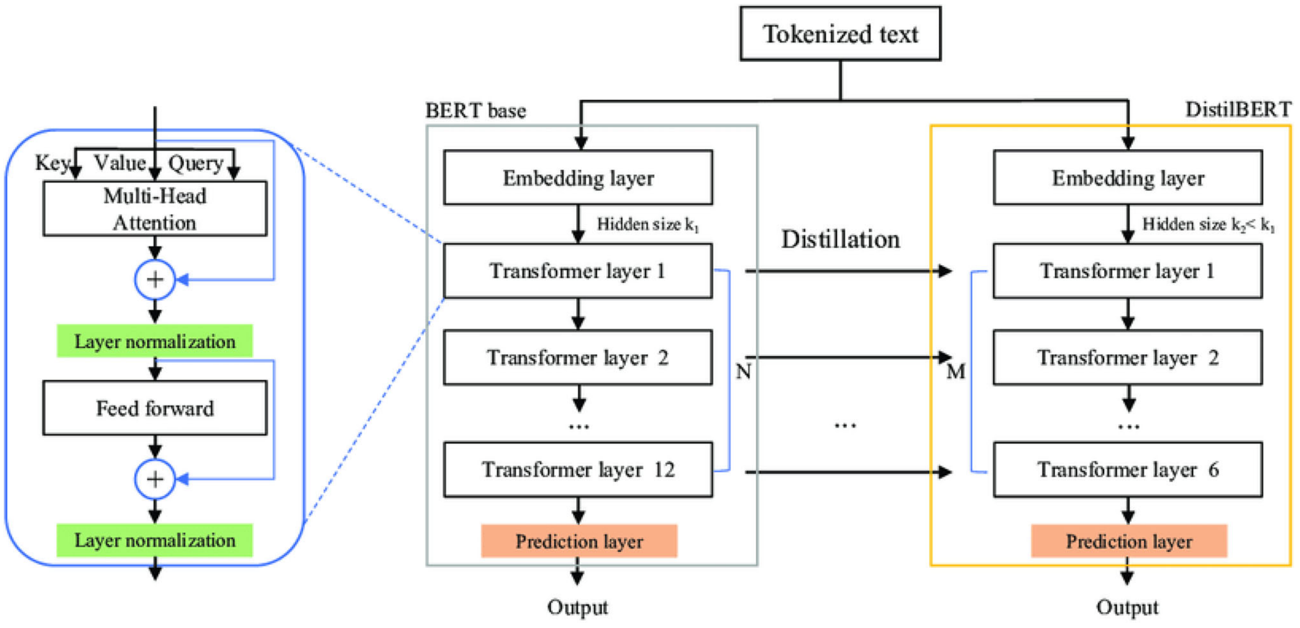


**Fig. 6** DistilBERT model architecture [62]

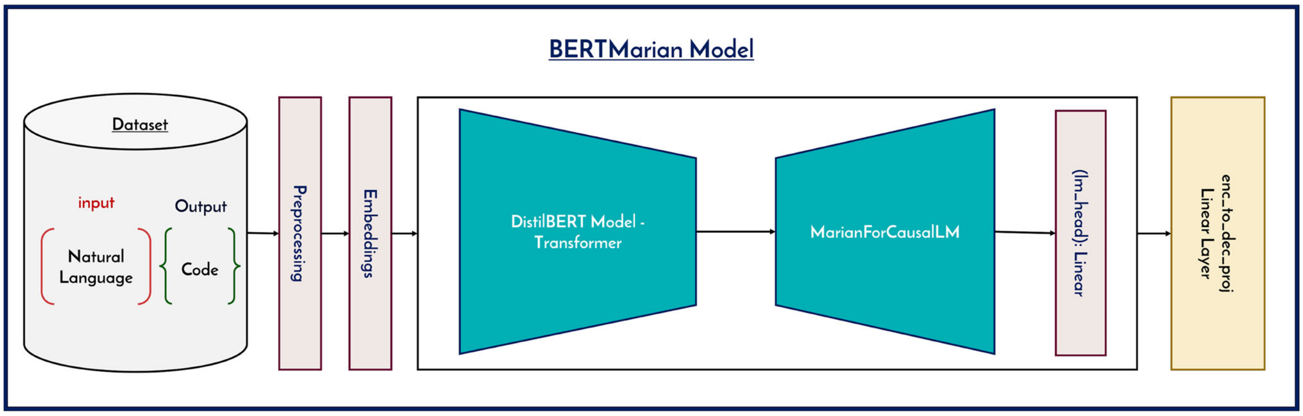


**Fig. 7** BERTMarian code generation model

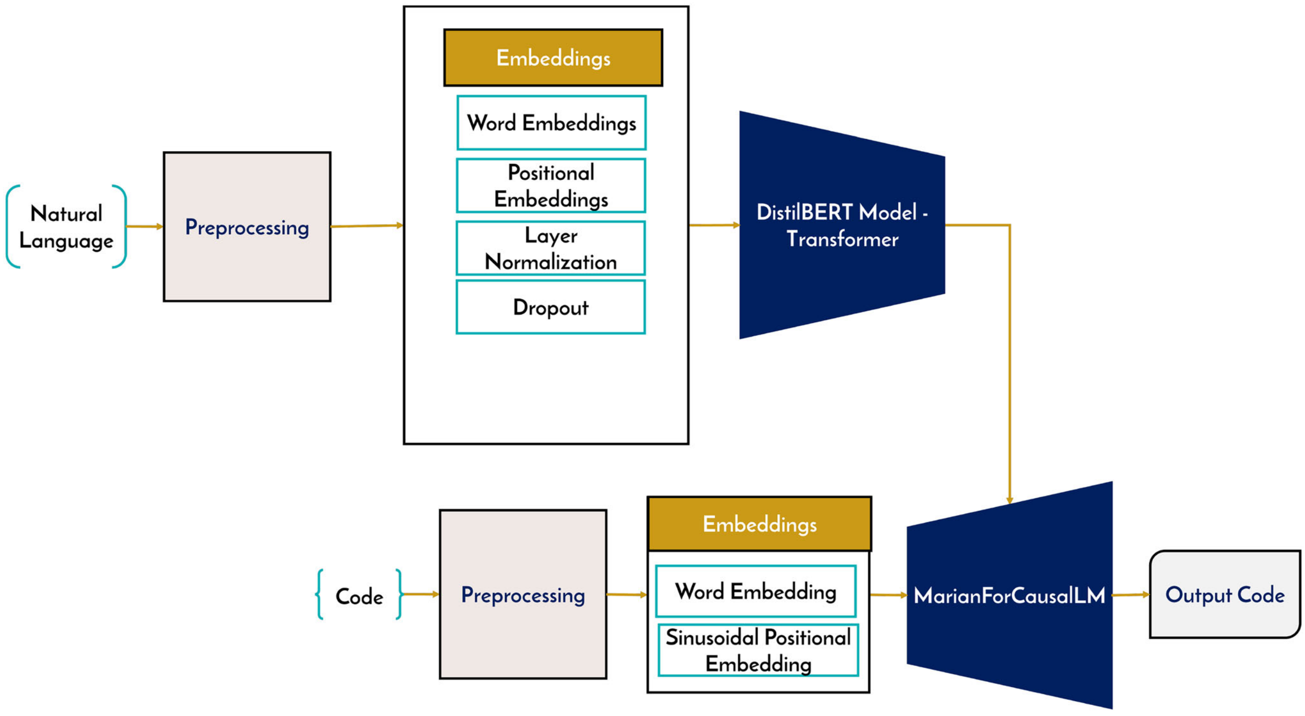


**Fig. 8** BERTMarian model architecture

approach for learning self-supervised language representations. It is used to pre-train transformer networks with a small amount of computation. ELECTRA models, like GAN discriminators, are trained to identify "real" input tokens from "false" input tokens created by another neural network as shown in Fig. 10. ELECTRA provides impressive results at a modest scale. On the SQuAD 2.0 dataset, ELECTRA produces cutting-edge outcomes at big scale.

The ELECTRA pre-training approach involves training two neural networks, a generator G and a discriminator D, each consisting of an encoder that maps input tokens into contextualized vector representations. The generator is trained to perform masked language modeling (MLM), where it predicts the original identities of masked-out tokens, while the discriminator is trained to distinguish between tokens in the data and tokens replaced by generator samples. After pre-training, the generator is discarded, and the discriminator is fine-tuned on downstream tasks.

This pre-training model is more efficient than Masked Language Modeling (MLM) because it was defined across all input tokens rather than just the tiny selection that was masked away. With the same model size, data, and computation, the contextual representations acquired by ELECTRA model outperformed those trained by BERT. The improvements are especially significant for tiny models. We used google electra-base-discriminator which has 6 hidden layers

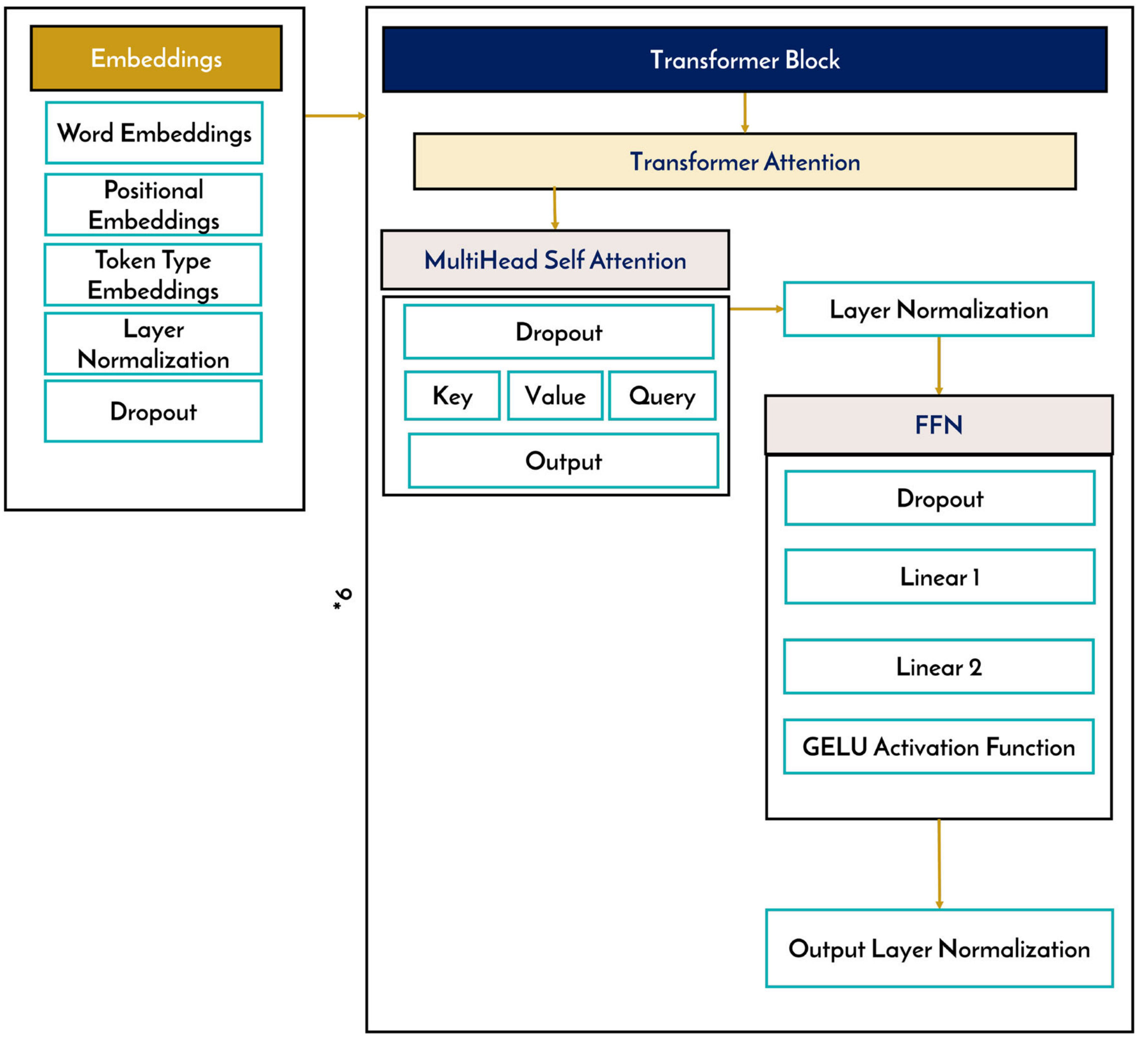
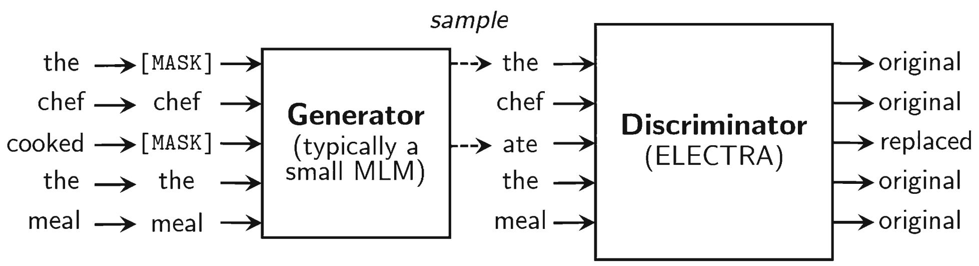


**Fig. 9** BERTMarian encoder

**Fig. 10** ELECTRA model like GAN [63]



as shown in Fig. 11. This model as an ELECTRA encoder with Marian Decoder.

Figure 12 shows the architecture of the ELECTRAMarian model and the encoder architecture is declared and shown in Fig. 13.

## LUKEMarian model

LUKE [64] is a transformer model that was trained using a vast quantity of entity-annotated data acquired from Wikipedia. It treats not just words but also entities as separate tokens and uses the transformer to construct intermediate and output representations for all tokens. LUKE varies from previously contextualized word representations (CWRs) and can directly simulate entity relationships since entities are represented as tokens as shown in Fig. 14. LUKE is trained with a novel pretraining task that is a simple extension of BERT's masked language model (MLM). The job entails masking entities at random by substituting them with [MASK] ones and training the model by predicting the originals of these

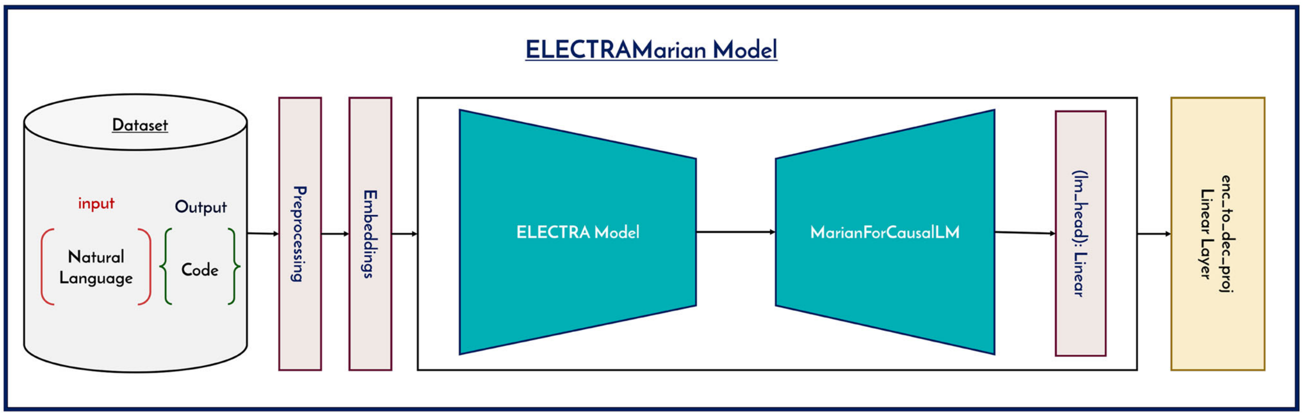


**Fig. 11** ELECTRAMarian code generation model

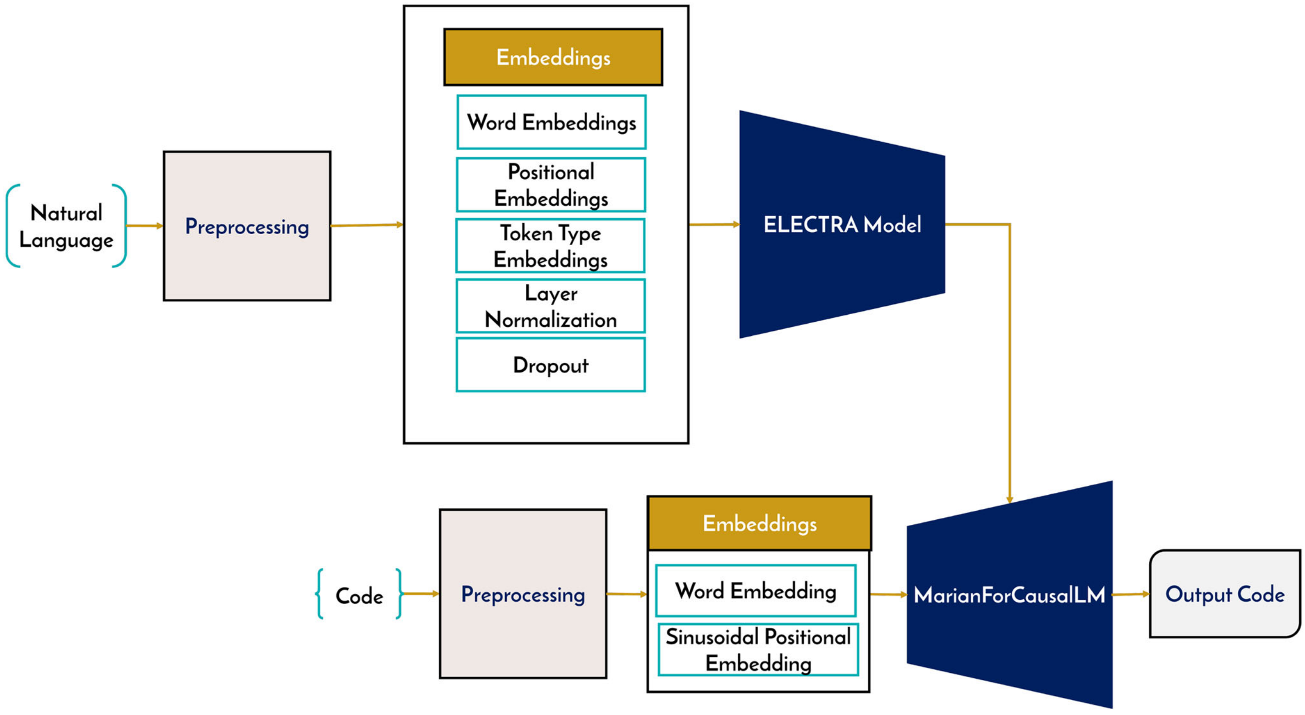


**Fig. 12** ELECTRAMarian model architecture

masked entities. RoBERTa was employed as the basic pre-trained model and pre-train the model by simultaneously maximizing the MLM objectives.

As shown in Fig. 15 we combined LUKE model with Marian decoder to construct our final Seq2Seq hybrid model. This idea has some advantages for representing words by their values and their entities.

LUKE is a contextualized representation created primarily for entity-related activities. Using a vast quantity of entity-annotated corpus acquired from Wikipedia, LUKE is trained to predict randomly masked words and entities. LUKE represents the word and its entity. The LUKE base model has 12 hidden layers and the hidden size equals 768. It has 253 M total number of parameters. When LUKE is applied to the downstream tasks, it computes representations of arbitrary entities in the text by employing [MASK] entities as inputs. If the task includes entity annotation, the model generates entity representations based on the rich entity-centric information stored in the relevant entity embeddings. We used the LUKE-base model as an encoder but only 6 layers from it. Combining this 6-layer LUKE-base model with Marian Decoder generated the LUKEMarian. Figure 16 shows the architecture of the LUKEMarian model and the encoder architecture is declared and shown in Fig. 17.

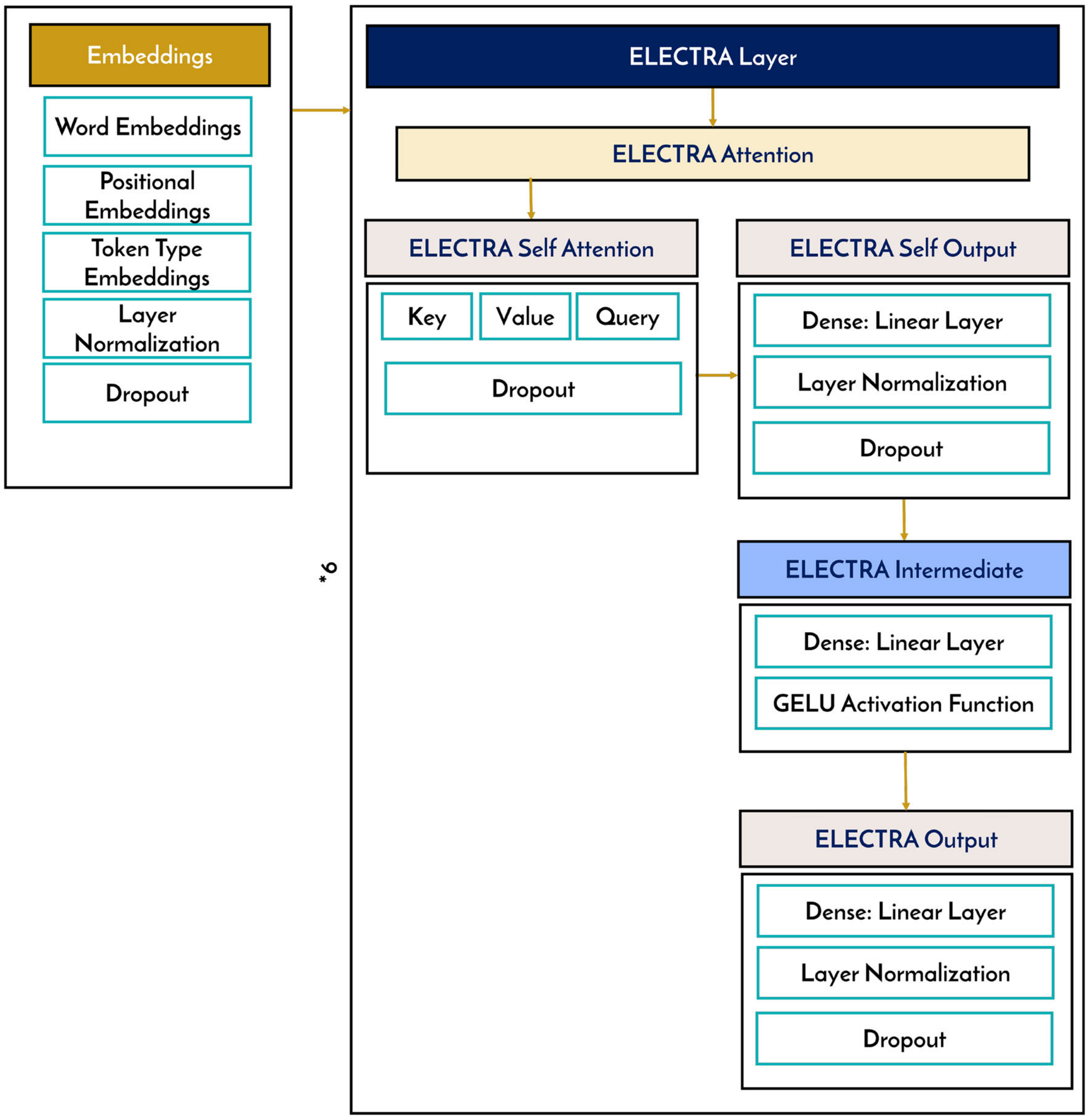


**Fig. 13** ELECTRAMarian encoder

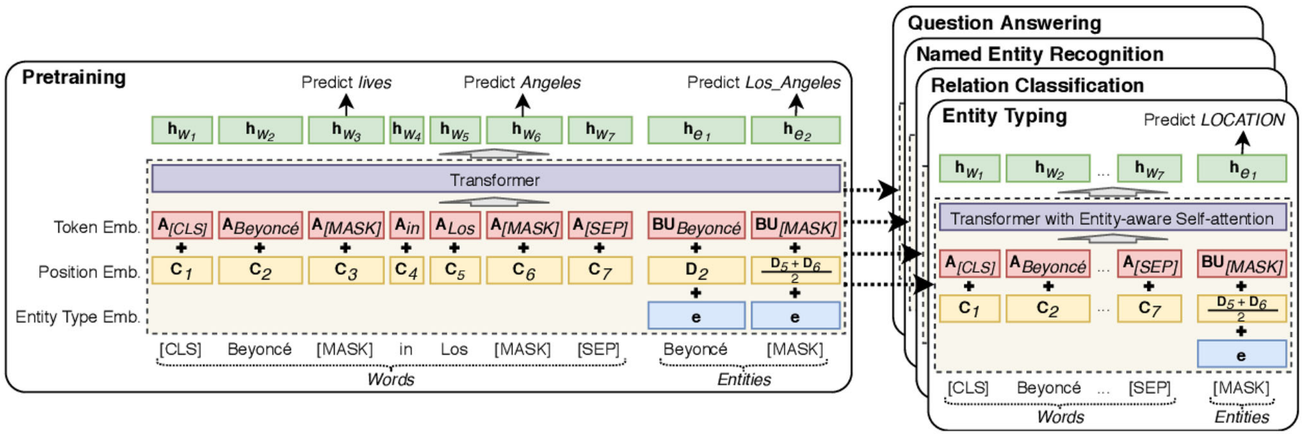


**Fig. 14** LUKE outputs are contextualized representations for each word and entity [64]

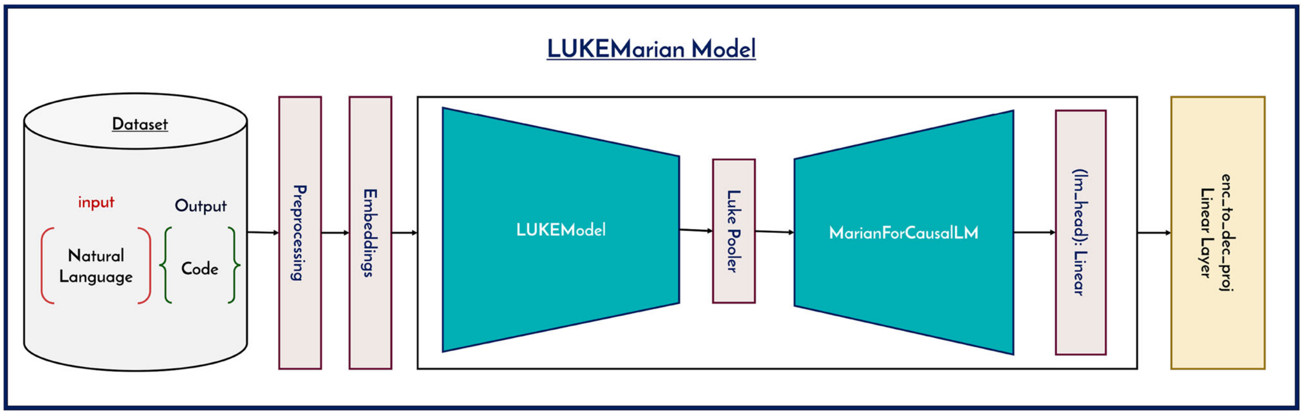


**Fig. 15** LUKEMarian code generation model

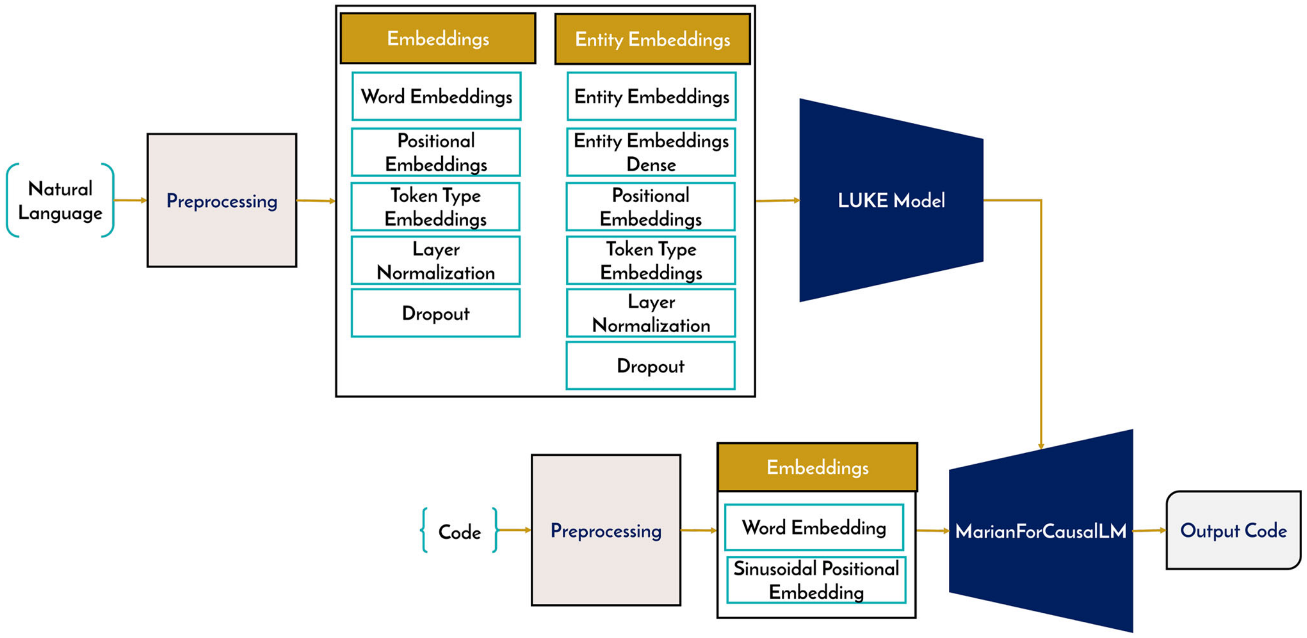


**Fig. 16** LUKEMarian model architecture

## LUKE pooler layer

The pooler layer in the LUKE model (LUKE stands for "Language Understanding with Knowledge-based Embeddings") serves as a summarization layer that takes the contextualized representations of the input tokens and produces a fixed-length representation, often referred to as a pooled representation or a sentence-level representation.

The specific functionality of the pooler layer in LUKE includes the following:

- Aggregating token representations: The pooler layer takes the contextualized representations of the input tokens, typically obtained from the transformer encoder layers, and aggregates them into a single representation. This aggregation process summarizes the information from the individual tokens and captures the overall meaning of the input sequence.
- Sentence-level encoding: The pooled representation produced by the pooler layer encapsulates the contextual information from the entire input sequence. This representation is often used as a sentence-level encoding that can be fed into downstream tasks, such as entity recognition or relation extraction.
- Semantic extraction: The pooler layer helps extract high-level semantic information from the input tokens. By summarizing the token representations, it focuses on capturing the most salient features of the input sequence and discards less relevant or noisy information.

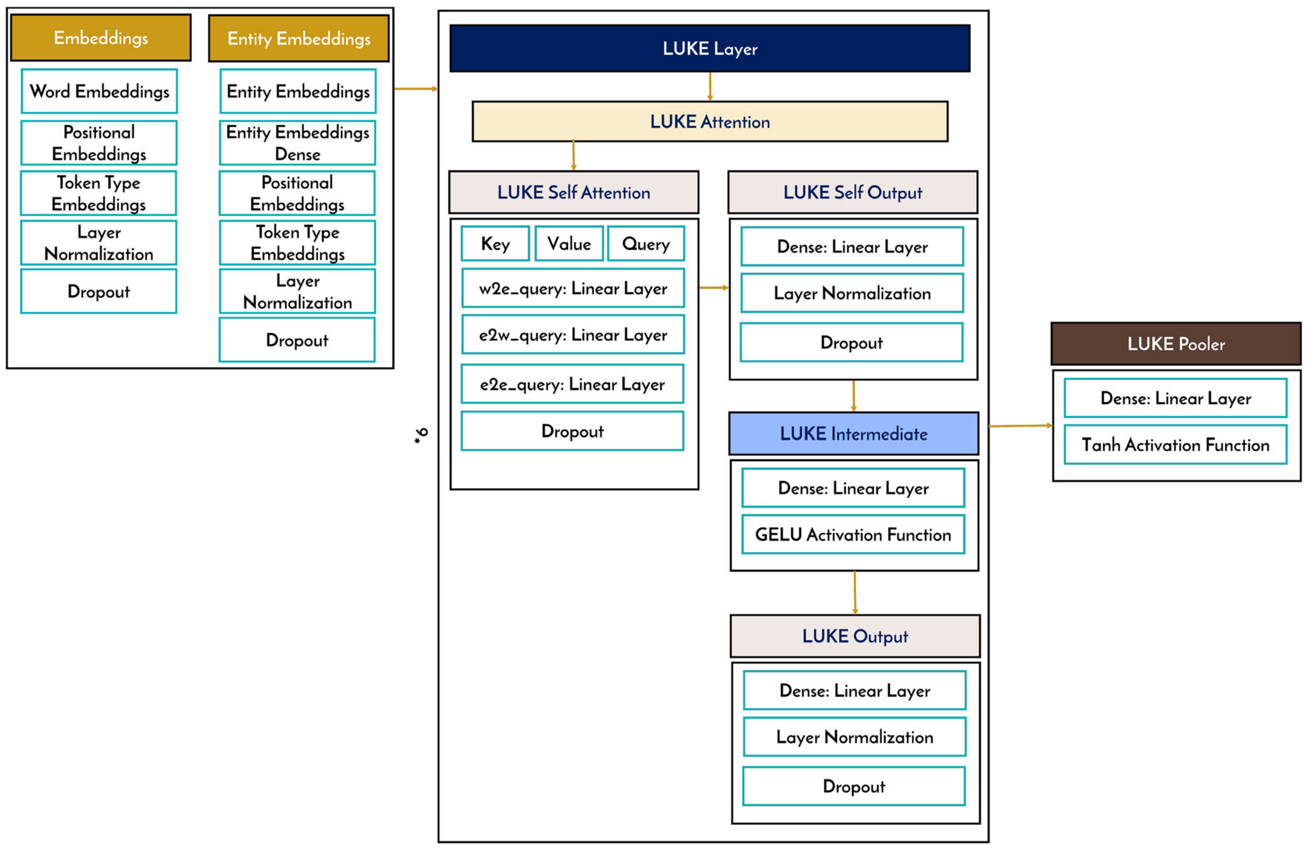


**Fig. 17** LUKEMarian encoder

Overall, the pooler layer in LUKE plays a crucial role in producing a fixed-length representation that captures the contextual information and semantic meaning of the input sequence. This pooled representation can then be used for various downstream tasks that require sentence-level understanding or knowledge extraction.

The proposed models share a foundational connection in their reliance on pre-trained language models for sequence generation tasks. DistilBERT, a streamlined version of BERT, serves as a rapid and efficient encoder–decoder model. The reduction in layers and removal of certain components make it computationally efficient while maintaining performance. Its self-supervised pre-training utilizes the BERT base model, emphasizing efficiency gains without sacrificing effectiveness.

RoBERTa, an evolution from BERT by Facebook, demonstrates a further enhancement in training methodology. Eliminating next-sentence pretraining, utilizing larger mini-batches and learning rates, and training on longer sequences contribute to improved performance. DistilRoBERTa, a distilled version, maintains efficiency with six layers, 768 dimensions, and a case-sensitive approach, using a smaller dataset than its teacher model.

ELECTRA, an innovative model from Stanford and Google, introduces a novel approach to self-supervised learning. Its efficiency surpasses Masked Language Modeling (MLM), as it identifies "real" input tokens from "false" ones created by another neural network. Its pre-training involves a discriminator distinguishing between genuine and generated tokens, showcasing effectiveness on large-scale datasets like SQuAD 2.0.

LUKE, based on the RoBERTa model, takes a unique approach by treating entities as separate tokens. Its pre-training involves masking entities and predicting originals, providing contextualized representations for both words and entities. LUKE's 12 hidden layers and 768 hidden size contribute to a comprehensive model for entity-related tasks.

These models share the use of pre-trained language models, each introducing specific optimizations and training strategies. While DistilBERT emphasizes efficiency, RoBERTa focuses on enhanced training, ELECTRA introduces a discriminator approach, and LUKE extends BERT's masked language model to entities. Together, they represent a spectrum of approaches, offering a versatile toolkit for sequence generation tasks.

## Error analysis and correction

Code assistance is a rapidly advancing field that combines the power of artificial intelligence (AI) and machine learning with the expertise of human programmers to enhance and streamline the process of software development. This technology leverages intelligent algorithms and data-driven models to assist developers in writing code, improving code quality, and increasing overall productivity. Traditional software development requires significant manual effort to write and debug code, often resulting in time-consuming and error-prone processes.

Code assistance aims to alleviate these challenges by providing automated tools that analyze existing codebases, understand programming patterns, and generate suggestions or complete sections of code. The underlying concept behind AI-assisted coding involves training machine learning models on large amounts of code repositories to learn patterns, syntax, and best practices. These models can then assist developers by providing code completion suggestions, identifying potential bugs or vulnerabilities, and offering optimizations based on established coding standards.

One of the key advantages of code assistance is its ability to enhance developer productivity. By automating repetitive tasks and providing intelligent suggestions, developers can focus on higher-level design decisions and problem-solving, accelerating the overall development process. Furthermore, AI-assisted coding can significantly improve code quality. By analyzing vast amounts of code, AI algorithms can identify common coding mistakes, detect potential bugs, and suggest code refactoring or optimizations. This helps in reducing errors and enhancing the reliability, maintainability, and performance of software applications.

Another benefit of code assistance is its potential to facilitate knowledge transfer and skill-sharing within development teams. By capturing and analyzing coding patterns, best practices, and successful code implementations, code assistance tools can assist less experienced developers in learning from the collective expertise of their peers, ultimately elevating the skill level of the entire team. However, it is important to note that AI-assisted coding is not meant to replace human programmers. Rather, it serves as a powerful tool to augment their capabilities, improve efficiency, and enable them to tackle complex coding tasks more effectively.

### Error analysis in the generated code

Code Generation is an exciting area of research and development that aims to bridge the gap between human language and programming languages. In this task, the input is the human description and specifications, and the output is Python code. To accomplish this, we implemented the MarianCG model specifically designed for the code generation task. However, generating code automatically can sometimes lead to linting issues and errors in the generated code. To address this, we employed various tools and techniques to ensure the generated code meets the required standards.

Firstly, we used Flake8,[1] a popular Python linter, to analyze the generated code and identify any potential issues. Flake8 examines the code for violations of style guidelines and common errors, providing feedback on areas that need improvement. Additionally, we performed syntax analysis on the generated code to verify its correctness as Python code. This analysis involved checking for proper indentation, correct syntax usage, and overall adherence to the Python language rules. Any modifications required to make the code suitable as valid Python code was implemented.

### Code refining and correction

We employed several error correction tools and libraries to further enhance the quality and readability of the generated code. One such tool is Autopep8,[2] which automatically formats Python code according to defined style guidelines. It can fix issues related to indentation, line length, and whitespace, resulting in cleaner and more consistent code.

We also utilized the Add-trailing-comma[3] library, which adds trailing commas to Python lists and dictionaries. Trailing commas can improve code readability and help prevent errors, especially when modifying or extending these data structures. For consistent code formatting, we employed Yapf[4] and Black,[5] both popular Python code formatters. These tools automatically format code according to various style guides, such as PEP 8 and Google's style guide. By applying these formatters, the generated code becomes easier to read and maintain. YAPF is a formatter for Python files developed by Google. It is designed to be highly configurable and have a low impact on the code being formatted. YAPF reformats the code to follow the style guide specified in PEP 8, but with a focus on readability and consistent formatting.

YAPF is available for installation through pip and can be used as a Python module or as a command-line interface. It provides options for specifying the maximum line length, variable naming style, indentation, and more. YAPF can be used along with other Python code analysis tools like Flake8 and PyLint to ensure consistent code quality. To ensure proper import organization, we used Isort, a Python library that sorts and formats imports in Python code. Isort helps maintain a consistent and logical order for imports, improving code organization and readability.

1 https://flake8.pycqa.org/en/latest/.

2 https://github.com/hhatto/autopep8.

3 https://github.com/asottile/add-trailing-comma.

4 https://github.com/google/yapf.

5 https://github.com/psf/black.

Lastly, we incorporated Ruff, a Python library for code refactoring, to enhance the readability and maintainability of the generated code. Ruff automatically performs code transformations, such as simplifying complex expressions, extracting functions, and applying another refactoring to improve the code's structure and clarity.

By combining these error correction and code enhancement techniques, the generated code from the human description can be refined, ensuring it meets the required Python coding standards and best practices. This approach contributes to the overall reliability, maintainability, and readability of the codebase, facilitating efficient software development. All the complete process for the code generation, lining, analysis, and correction is shown in Fig. 18.

So, there are three phases to the code generation and AI-assistant code as follows:

1. Code generation from natural language
2. Linting and Error Analysis using Flake8
3. Refining the generated code and error correction using the rest of the tools.

## Implementation and experimental setup

### Datasets

By fine-tuning the MarianMT transformer model on the CoNaLa and DJANGO datasets, we got MarianCG which is a new transformer model that was built on the pre-trained transformer. We followed [29, 33, 61] in the CoNaLa dataset and selected the top mined samples depending on the probability that the NL-Code pair is correct.

Training the models can be done by using the CoNaLa-train and/or CoNaLa-mined datasets, then take the rewritten intent field from the CoNaLa-test dataset as input and generating output from it. The CoNaLa-Large dataset contains about 26K different NL-Code. This dataset contains the CoNaLa-train and examples from CoNaLa-mined and the 500 examples in CoNaLa-test to compare by the same benchmarks as other state-of-the-art contributors. Also, we used DJANGO which contains 19K examples, and got the results.

Table 2 displays the datasets employed in each experiment, as well as the dataset size and number of records in the train, validation, and test sets of data.

### Experimental setup

In our experiments, we employed DJANGO and the CoNaLa dataset of 26K NL-Code pairings from CoNaLa-train and CoNaLa-mined. The testing data set contains 500 samples from CoNaLa-test that were compared using the same benchmarks as other state-of-the-art contributors.

**Table 2** Datasets in each experiment and distribution of the data

| Dataset | Dataset size | Dataset split | | |
|---|---|---|---|---|
| | | Train | Validation | Test |
| DJANGO | 19K | 16,000 | 1000 | 1805 |
| CoNaLa Large | 26K | 24,687 | 1237 | 500 |

The implementation of our generated models was done with the CoNaLa dataset using Google's Colab Pro. Our development was on Python and PyTorch, and thanks to pre-trained transformer models where we used various pre-trained language models from their model hub. For training, we relied on HuggingFace's trainer and the implementation of the learning rate scheduler with batch size equals 2. For the implementation, we applied these parameters: Adam optimizer, Weight decay $= 0.01$, Learning rate $= 1e^{-5}$, Number of hidden layers for the encoder $= 6$, Number of hidden layers for the decoder $= 6$, a linear learning rate scheduler, warmup ratio of 0.05 seed $= 1995$ early stopping, a length penalty of 0.9, and we used beam search with four beams for generation.

## Experimental results

We present the results of our proposed hybrid model which evaluated on the CoNaLa and DJANGO datasets. For the MarianCG experiments, we measured the model's code generation performance by calculating metrics such as code accuracy and code similarity scores. The results showed that MarianCG achieved competitive code generation accuracy on both datasets, demonstrating its effectiveness in generating syntactically correct code snippets.

We introduced our novel hybrid model, which leverages the strengths of MarianCG while incorporating additional contextual information from BERT embeddings and various encoder-only models. The hybrid model significantly improved the quality of generated code, achieving higher code accuracy and semantic relevance compared to MarianCG on both datasets. These findings demonstrate the efficacy of our proposed hybrid approach for enhancing code generation performance, making it a promising solution for code-related Transformer-Based Processing tasks.

### Results of the proposed models

we present four novel transformer-based architectures designed to excel in code generation tasks, specifically targeting the Python programming language. Our proposed models combine the strengths of popular pre-trained models like DistilBERT, Electra, and Luke, with the versatile Marian

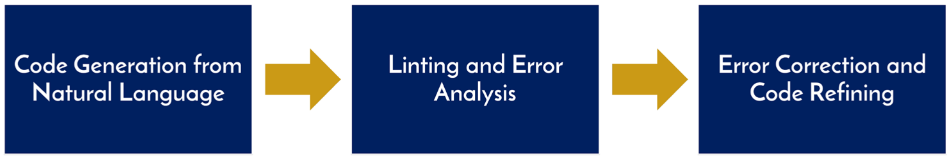

Code Generation from Natural Language
Linting and Error Analysis
Error Correction and Code Refining

**Fig. 18** Code assistant process

decoder. We evaluate our models on two widely used datasets, CoNaLa and DJANGO, to demonstrate their performance and capabilities.

Our first model, RoBERTaMarian, integrates the DistilROBERTa pre-trained model as the encoder and the Marian decoder. It achieves impressive performance on both datasets, particularly on DJANGO, where it attains a superior BLEU score of 88.91, an exact match accuracy of 77.95%, SacreBLEU of 74.08, and a ROUGE score of 92.76. These results demonstrate the model's proficiency in generating high-quality code solutions that accurately match human-generated references.

The second model, BERTMarian, utilizes the DistilBERT encoder and Marian decoder. On the CoNaLa dataset, it achieves a BLEU score of 32.46, an exact match accuracy of 12.4%, SacreBLEU of 29.48, and a ROUGE score of 43.95. Similarly, on the DJANGO dataset, it attains a BLEU score of 56.55, an exact match accuracy of 76.78%, SacreBLEU of 29.48, and a ROUGE score of 43.95. These results indicate the model's ability to generate code solutions that closely align with human-written code while maintaining linguistic fluency and capturing critical code semantics.

The third model, ELECTRAMarian, combines the powerful ELECTRA encoder with the Marian decoder. On the CoNaLa dataset, it achieves a BLEU score of 30.18, an exact match accuracy of 10.0%, SacreBLEU of 27.15, and a ROUGE score of 42.42. Meanwhile, on the DJANGO dataset, it attains a BLEU score of 53.02, an exact match accuracy of 65.32%, SacreBLEU of 58.16, and a ROUGE score of 83.91. These results showcase the model's versatility and high performance in generating code solutions that accurately match human-authored references while demonstrating linguistic fluency and capturing important code semantics.

Lastly, the LUKEMarian model, which employs the LUKE encoder and Marian decoder, shows promising results on both datasets. On the CoNaLa dataset, it achieves a BLEU score of 29.83, an exact match accuracy of 7.6%, SacreBLEU of 25.32, and a ROUGE score of 39.84. On the DJANGO dataset, it attains a BLEU score of 89.34, an exact match accuracy of 78.50%, SacreBLEU of 74.66, and a ROUGE score of 93.11. These results confirm the model's efficiency in generating code solutions that closely align with human-written code while maintaining linguistic fluency and capturing critical code semantics.

These results pave the way for further advancements in code generation technology and its potential integration into real-world programming and automation applications. Table 3 shows the results and the performance metrics of our proposed models on the CoNaLa dataset. Also, Table 4 shows the results of those proposed models on the DJANGO dataset.

Figures 19 and 20 show these results for the proposed code generation models on the CoNaLa and DJANGO datasets respectively. The findings show that RoBERTaMarian model produced the highest BLEU score of 35.74.

Our proposed transformer-based models, RoBERTaMarian, BERTMarian, ELECTRAMarian, and LUKEMarian, demonstrate exceptional performance in code generation tasks, particularly on the Python programming language. Their ability to generate accurate, relevant, and contextually appropriate code outputs solidifies their standing as cutting-edge solutions in the programming and Transformer-Based Processing domain.

## Results evaluation

This paper unveiled a range of powerful encoder–decoder models for code generation, each demonstrating unique strengths. RoBERTaMarian and BERTMarian stood out with their competitive performance on the CoNaLa dataset, while ELECTRAMarian and LUKEMarian showcased impressive proficiency on the DJANGO dataset. Additionally, MarianCG served as a strong baseline, delivering promising results on both datasets. Understanding the trade-offs between these models allows us to tailor their application to various code generation scenarios, catering to the specific requirements of each task. When compared to other researchers who worked on the code generation problem on CoNaLa, this result has the highest BLEU score. The second item to mention is the highest ROUGE score or ROUGE-L result, which indicates the LCS-based statistics. The longest common subsequence problem takes sentence-level structural similarities into account and automatically selects the longest co-occurring in sequence n-grams. The MarianCG model has the highest ROUGE score of 49.63. The results of the state-of-the-art code generation models on the CoNaLa dataset are shown in the Table 5 and indicated in Fig. 21 for the CoNaLa models.

RoBERTaMarian is considered the first among the models that get accurate results of the generated code. This model records a BLEU score of 35.74 with the advantage of being

**Table 3** Performance metrics of all models on the CoNaLa dataset

| No. | Model | Evaluation metrics | | | |
|---|---|---|---|---|---|
| | | Bleu score | Exact match accuracy (%) | Sacrebleu | Rouge score |
| 1 | MarianCG | 34.4291 | 10.2 | 30.6776 | 49.6272 |
| 2 | RoBERTaMarian model | 35.7365 | 13.8 | 31.3025 | 44.2547 |
| 3 | BERTMarian | 32.4618 | 12.4 | 29.479 | 43.9467 |
| 4 | ELECTRAMarian | 30.1819 | 10.0 | 27.1495 | 42.4232 |
| 5 | LUKEMarian | 29.8281 | 7.6 | 25.3187 | 39.8431 |

**Table 4** Performance metrics of all models on the DJANGO dataset

| No. | Model | Evaluation metrics | | | |
|---|---|---|---|---|---|
| | | Bleu score | Exact match accuracy (%) | Sacrebleu | Rouge score |
| 1 | MarianCG | 90.41 | 81.83 | 75.906 | 94.647 |
| 2 | RoBERTaMarian | 88.9123 | 77.95 | 74.083 | 92.7351 |
| 3 | BERTMarian | 56.55 | 76.676 | 64.884 | 88.692 |
| 4 | ELECTRAMarian | 53.02 | 65.319 | 58.155 | 83.905 |
| 5 | LUKEMarian | 89.3424 | 78.504 | 74.6583 | 93.1133 |

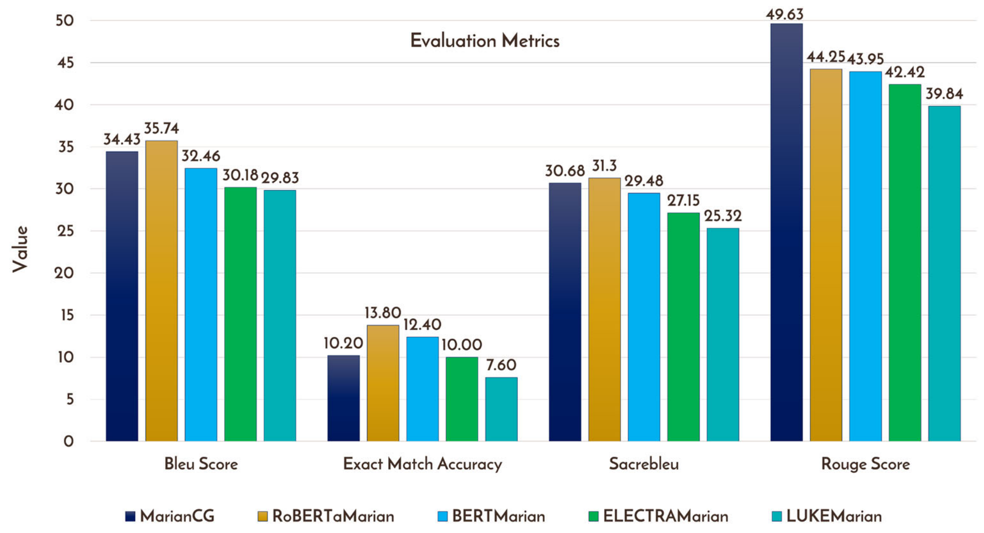


**Fig. 19** Results and performance metrics of our proposed models on the CoNaLa dataset

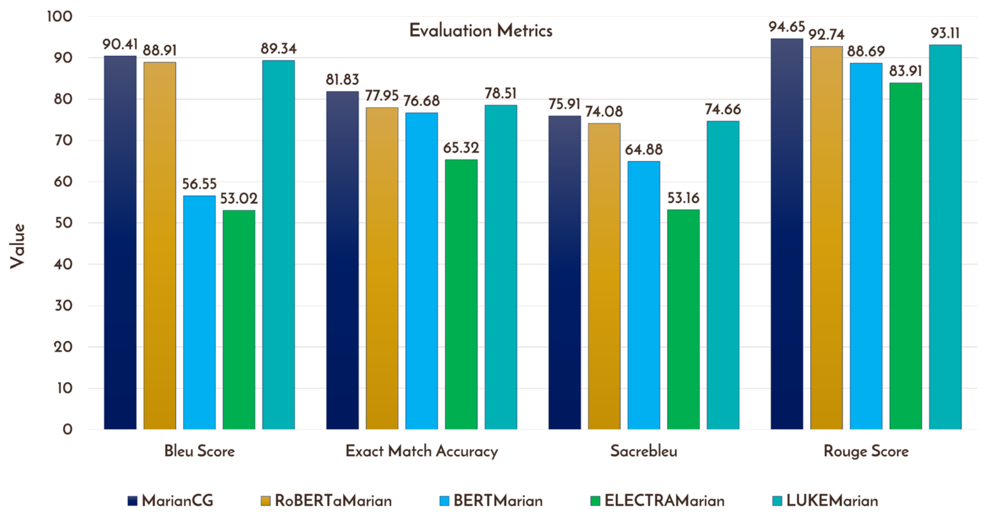


**Fig. 20** Results and performance metrics of our proposed models on the DJANGO dataset

**Table 5** State of the art code generation models on CoNaLa

| Rank | Model | Evaluation metrics | | Year |
|---|---|---|---|---|
| | | BLEU score | Exact match accuracy (%) | |
| 1 | RoBERTaMarian (ours) | 35.7365 | 13.8 | 2023 |
| 2 | MarianCG (ours) [61] | 34.4291 | 10.2 | 2022 |
| 3 | TranX + BERT w/mined [34] | 34.2 | 5.8 | 2022 |
| 4 | BERT + TAE [32] | 33.41 | – | 2021 |
| 5 | BERTMarian (ours) | 32.4618 | 12.4 | 2023 |
| 6 | External knowledge with API + Reranking [29] | 32.26 | – | 2020 |
| 7 | External knowledge with API [29] | 30.69 | – | 2020 |
| 8 | BART W/mined [33] | 30.55 | – | 2021 |
| 9 | ELECTRAMarian (ours) | 30.1819 | 10.0 | 2023 |
| 10 | Reranker [26] | 30.11 | – | 2019 |
| 11 | LUKEMarian (ours) | 29.8281 | 7.6 | 2023 |
| 12 | BART base [33] | 26.24 | – | 2021 |
| 13 | TranX [25] | 24.3 | – | 2018 |

fast, small and containing multi-head attention. Besides that, MarianCG is considered the second model to have a BLEU score of 34.43. These are our models which record the first and second in the code generation models results. Besides that, our generated models have the number of BLEU score which are in the top-ranking code generation models.

Also, Table 6 shows the results of all state-of-the-art models on DJANGO, and the performance metrics are indicated in Fig. 22. Comparing all models that worked on the DJANGO dataset proved that our models got high values of BLEU score such as MarianCG, LUKEMarian model, and RoBERTaMarian model. These models got BLEU scores of 90.41, 89.34, and 88.91 respectively.

RoBERTaMarian model is the first among the models that get accurate results of the generated code on the CoNaLa dataset. Also, the MarianCG model is ranked in the top models for its accurate predictions in terms of BLEU score and exact match accuracy. This model has a smaller size architecture. It has six layers in the encoder and six layers in the decoder, whereas other models have larger model sizes.

The contrast analysis presented in this paper demonstrates the validity of the proposed models by comparing their performance with other state-of-the-art models on multiple datasets and highlighting their strengths in terms of code quality, semantic similarity, and computational efficiency.

The proposed models (RoBERTaMarian, BERTMarian, ELECTRAMarian, LUKEMarian, and MarianCG) are compared with other models that have been proposed in the literature for code generation tasks, specifically on the CoNaLa and DJANGO datasets. The results show that the proposed models achieve competitive performance on both datasets, with RoBERTaMarian and BERTMarian performing well on CoNaLa, and ELECTRAMarian and LUKEMarian performing well on DJANGO. This suggests that the proposed models are capable of generating high-quality code across different domains and tasks.

On the CoNaLa dataset, RoBERTaMarian has the highest BLEU score among all models, and MarianCG has the highest ROUGE score, indicating its ability to generate code with high semantic similarity to the input. This suggests that the proposed models are able to generate code that is not only syntactically correct but also semantically meaningful.

On the DJANGO dataset, the proposed models achieve high BLEU scores, with MarianCG, LUKEMarian, and RoBERTaMarian ranking in the top positions. This suggests that the proposed models are effective in generating code for a variety of programming tasks and languages.

Additionally, the proposed models have advantages in terms of computational efficiency, with MarianCG having a smaller model size than other models while still achieving high performance. This suggests that the proposed models are not only effective in generating high-quality code but also efficient in terms of the computational resources required.

Overall, the contrast analysis presented in this paper demonstrates the validity of the proposed models by comparing their performance with other state-of-the-art models on multiple datasets and highlighting their strengths in terms of code quality, semantic similarity, and computational efficiency.

## Error/warning analysis and refining

Code assistant represents a powerful symbiosis between human programmers and AI technology, providing intelligent tools that assist in code writing, error detection, and optimization [65, 66]. By leveraging AI algorithms

**Table 6** State-of-the-art code generation models on DJANGO

| Rank | Model | Evaluation metrics | | Year |
|---|---|---|---|---|
| | | BLEU score | Exact match accuracy (%) | |
| 1 | MarianCG (ours) [61] | 90.41 | 81.83 | 2022 |
| 2 | LUKEMarian (ours) | 89.3424 | 78.504 | 2023 |
| 3 | RoBERTaMarian (ours) | 88.9123 | 77.95 | 2023 |
| 4 | TranX + BERT w/mined [34] | 79.86 | 81.03 | 2022 |
| 5 | BERT + TAE [32] | – | 81.77 | 2021 |
| 6 | BERTMarian (ours) | 56.55 | 76.68 | 2023 |
| 7 | ELECTRAMarian (ours) | 53.02 | 65.32 | 2022 |
| 8 | Reranker [26] | – | 80.2 | 2019 |
| 9 | TranX [25] | – | 73.7 | 2018 |

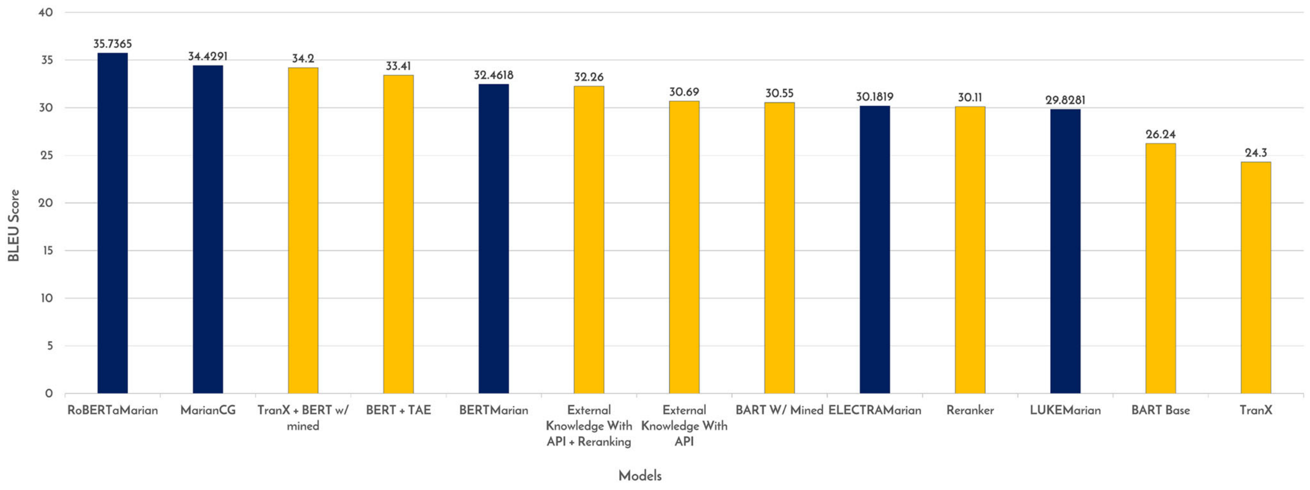


**Fig. 21** All results of the state of the art models in the code generation problem with CoNaLa

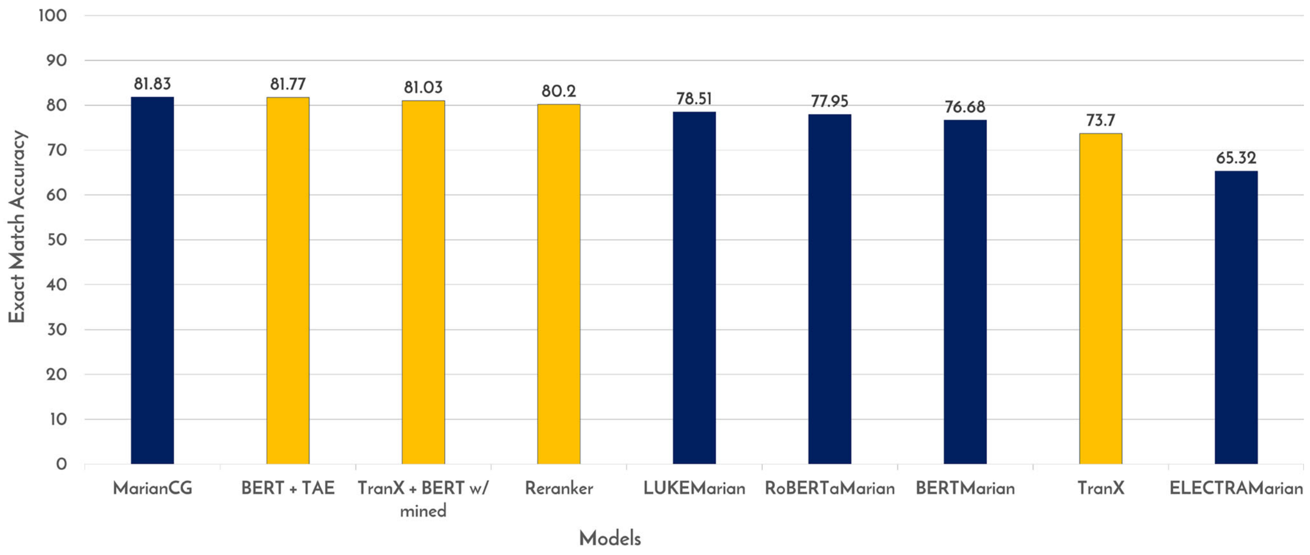


**Fig. 22** All results of the state-of-the-art models in the code generation on DJANGO with respect to accuracy

and machine learning, developers can enhance productivity, improve code quality, and facilitate knowledge sharing within development teams. As the capabilities of code assistant tools advance, they hold the potential to transform the software development landscape, making the process more efficient, reliable, and collaborative. We conducted a comprehensive error and warning analysis using Flake8, a powerful tool for code linting and static analysis. Flake8 helped identify potential issues in the generated Python code, such as syntax errors, undefined variables, and code style violations.

To address these concerns and improve code quality, we employed automatic code formatting tools such as Autopep8, which automatically corrected issues related to indentation, line length, and whitespace. This resulted in cleaner and more consistent code, aligning with established style guidelines. In addition to Autopep8, we utilized the Add-trailing-comma library to enhance code readability and reduce errors associated with Python lists and dictionaries. By adding trailing commas to these data structures, we ensured a consistent format, especially when modifying or extending them during the code generation process. To maintain uniform code formatting across the generated solutions, we integrated two popular Python code formatters, Yapf and Black. Both tools automatically formatted the code according to various style guides, such as PEP 8 and Google's style guide. By employing these formatters, the readability and maintainability of the generated code significantly improved, making it easier for developers to comprehend and work with the code.

Yapf,[6] developed by Google, demonstrated high configurability and minimized its impact on the code being formatted. It strictly adhered to the PEP 8 style guide while emphasizing readability and consistent formatting. The tool provided flexible options for setting maximum line length, variable naming style, and indentation, among others. Yapf seamlessly integrated with other Python code analysis tools, such as Flake8 and PyLint, to ensure comprehensive and consistent code quality. To organize imports effectively, we implemented Isort, a Python library that efficiently sorts and formats import statements in Python code. By utilizing Isort, we maintained a logical and consistent order for imports, thereby enhancing code organization and readability. Lastly, we incorporated Ruff, a powerful Python library for code refactoring, to further enhance the readability and maintainability of the generated code. Ruff automatically performed code transformations, such as simplifying complex expressions, extracting functions, and applying various refactoring techniques, resulting in improved code structure and clarity.

The error analysis and distributions of the code generated from MarianCG and RoBERTaMarian models are shown in Table 7. Also, Figs. 23 and 24 show the distribution for these errors and warnings in the generated code and after refining.

After using these tools, the error and warning analysis on the MarianCG and RoBERTa models became less than before. By leveraging these tools and libraries in our code generation process, we achieved code solutions that not only met high-quality standards but were also consistently formatted and easy to maintain. The combination of static analysis, linting, formatting, and refactoring tools contributed significantly to the overall success of our code generation models and reinforced the importance of code quality in modern software development.

[6] https://github.com/google/yapf.

## Conclusion

This paper has demonstrated the promising application of pre-trained transformer language models in code generation, specifically through the combination of DistilRoBERTa, ELECTRA, and LUKE with the Marian Decoder. Our experimental results show that these models can generate high-quality code, as measured by static error detection and refactoring. The RoBERTaMarian model achieved a peak BLEU score of 35.74 and an exact match accuracy of 13.8% on the CoNaLa dataset, while the LUKEMarian model attained a BLEU score of 89.34 and an exact match accuracy of 78.50% on the DJANGO dataset. These results indicate that pre-trained language models have the potential to revolutionize code generation, offering a viable solution for converting human descriptions into executable code.

However, several study limitations should be acknowledged. Firstly, the dataset sizes used in our experimentation were relatively small compared to other code generation datasets, which may limit the generalizability of our findings to larger, more diverse datasets. Also, it is one line code generation and it is the start to be completed in the near future to work with functions, classes and see the connected speech in the natural language to have integrated and complete program. Secondly, we explored four pre-trained language models and one decoder, leaving room for investigation of other models and combinations that may yield superior performance or different trade-offs between accuracy and efficiency.

Despite these limitations, our work opens up exciting avenues for future research. One emerging topic is multimodal code generation, which involves generating code based on visual, audio, or video inputs. Integrating pre-trained language models with computer vision or multimedia processing techniques could unlock new possibilities in this area. Another important direction is explainable AI and interpretability, which is becoming increasingly relevant as AI systems become more pervasive. Incorporating attention mechanisms, saliency maps, or other interpretability techniques into pre-trained language models for code generation would allow developers to understand how the models arrive at their output.

Moreover, as AI models become increasingly relied upon in software development, understanding their robustness against adversarial attacks becomes crucial. Investigating attack strategies and developing corresponding defense mechanisms will ensure the security and trustworthiness of

**Table 7** Error and warning analysis on MarianCG and RoBERTaMarian models

| Error/warning message | MarianCG | | RoBERTaMarian | |
|---|---|---|---|---|
| | Using Flake8 | After correction | Using Flake8 | After correction |
| "undefined name" | 593 | 450 | 295 | 295 |
| "no newline at end of file" | 438 | 0 | 230 | 0 |
| ".format(...) has unused arguments" | 0 | 0 | 1 | 1 |
| "SyntaxError: invalid character" | 62 | 62 | 270 | 64 |
| "ambiguous variable name 'l'" | 3 | 3 | 2 | 2 |
| "missing whitespace around operator" | 4 | 0 | 0 | 0 |
| "'return' outside function" | 4 | 3 | 0 | 0 |
| "missing whitespace after ','" | 3 | 0 | 5 | 0 |
| "test for membership should be 'not in'" | 1 | 1 | 0 | 0 |
| "whitespace after '('" | 0 | 0 | 1 | 0 |

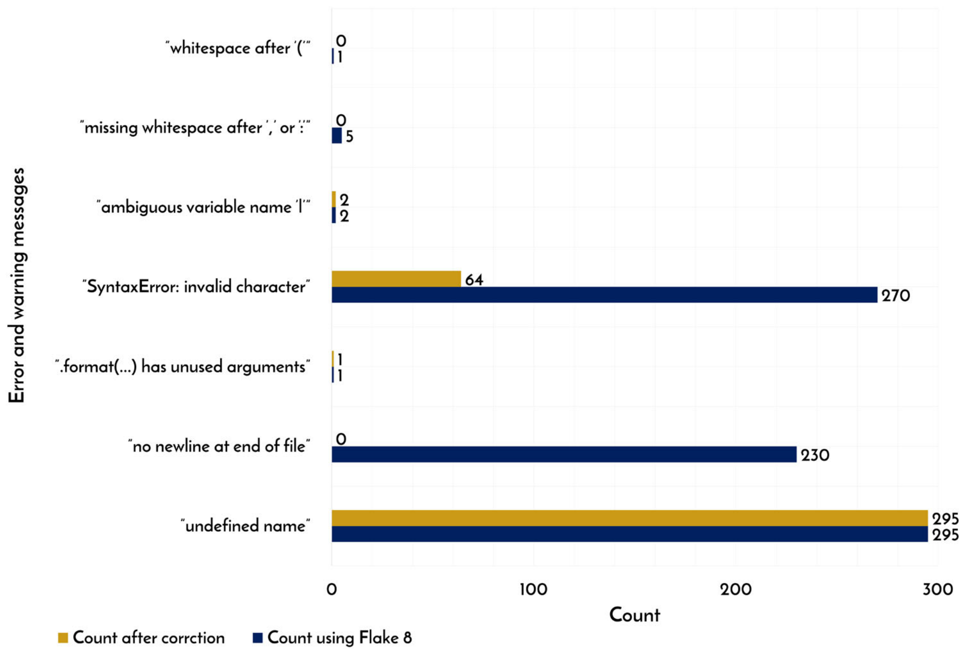


**Fig. 23** Error analysis in the code generated from MarianCG

these models in practice. Additionally, human-AI collaboration, where AI models assist humans in generating code, could lead to novel approaches in software development, blending the strengths of both humans and machines. Finally, it is essential to acknowledge the ethical implications of AI models automating parts of software development. Questions regarding ownership, accountability, and potential biases in the generated code necessitate open discussions, guidelines, and regulatory frameworks that promote responsible AI practices in software engineering.

The complexity of the input text is a critical factor in determining the complexity of the code generation task. As the input text becomes longer, more structured, and contains more domain-specific vocabulary, the task becomes increasingly challenging. This is because the model needs to capture longer-range dependencies, handle ambiguity, and generate coherent and accurate code that meets the requirements of the given task.

To quantify the complexity of the future work in the input text, we can consider several factors such as:

- Sentence length: Longer sentences with multiple clauses and phrases require more sophisticated encoding and decoding mechanisms to capture the relationships between tokens, entities, and actions.
- Domain-specific vocabulary: Specialized domains such as programming languages, legal documents, or medical reports often contain technical jargon and terminology that demand a deeper understanding of the subject matter.
- Ambiguity and uncertainty: Natural language is inherently ambiguous, with words and phrases having multiple meanings and contexts. The model must be able to resolve

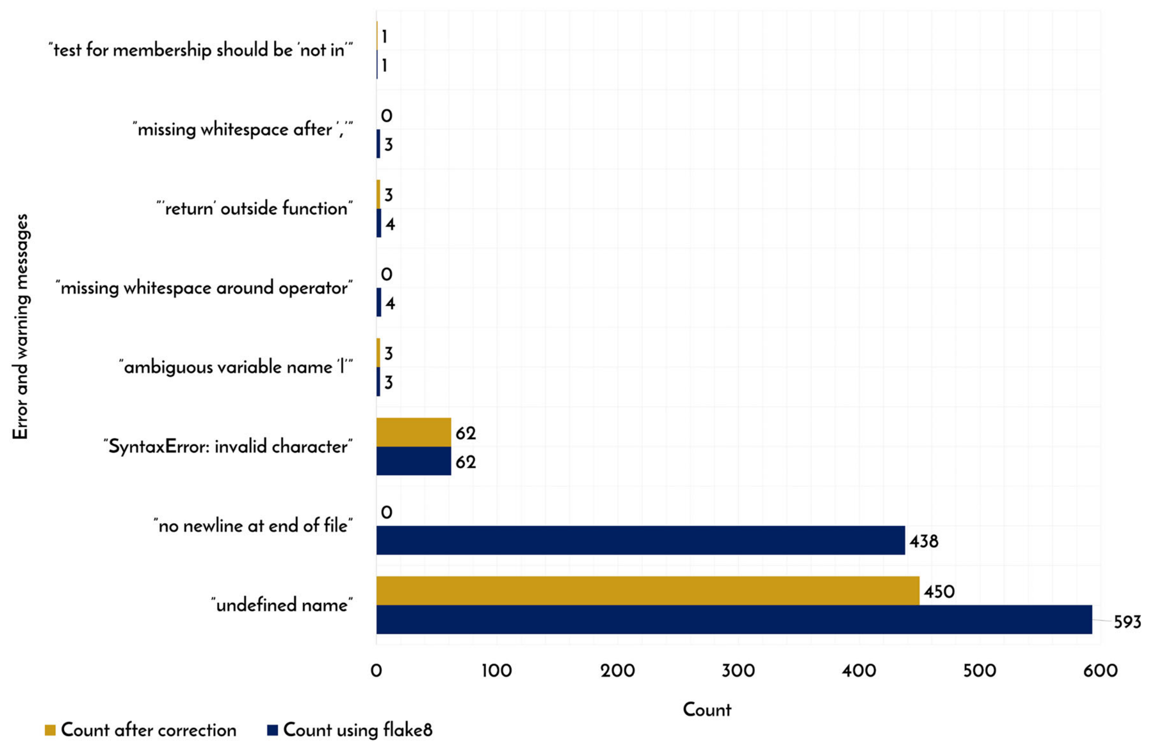


**Fig. 24** Error analysis in the generated code from RoBERTaMarian model

these ambiguities and generate appropriate code that satisfies the constraints of the task.

- Structural complexity: Code generation tasks may involve generating code that includes nested structures, loops, conditionals, and functions, which requires a deep understanding of programming concepts and syntax.
- The complexity of the input text directly impacts the complexity of the resulting code. For instance, a simple command-line interface (CLI) script may require less complex code than a web application with multiple routes, user authentication, and database interactions. Similarly, a code snippet that performs basic arithmetic operations may be simpler than one that implements machine learning algorithms or performance computations.

In conclusion, this paper demonstrates the potential of pre-trained transformer language models in code generation and highlights opportunities for future research in emerging topics such as multimodal code generation, explainable AI, adversarial attacks, and human-AI collaboration. Addressing the study limitations and ethical implications will help ensure the responsible adoption of these models in software development, ultimately enhancing the productivity and efficacy of software engineers.

**Acknowledgements** We would like to express our sincere gratitude to Prof. Mohamed Zaki for his invaluable guidance and mentorship throughout this research. His expertise and support have been instrumental in shaping the trajectory of this work.

**Author Contributions** The authors contributed to this work as follows: [AS]: Conceptualization, methodology, investigation, formal analysis, writing-original draft, visualization, Code implementation, experimentation, validation, data curation, and writing-review and editing. [SS]: Supervision. [MH]: Supervision.

**Funding** Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

**Data Availability** Datasets: CoNaLa Dataset: https://huggingface.co/datasets/AhmedSSoliman/CoNaLa-Large. DJANGO Dataset: https://huggingface.co/datasets/AhmedSSoliman/DJANGO. Models are available now at the huggingface hub, and can be used or tested and integrated into any project. You can find these models and easily deal with various datasets through the following links: RoBERTaMarian: https://www.huggingface.co/AhmedSSoliman/DistilRoBERTa-Marian-Model-on-CoNaLa. https://www.huggingface.co/AhmedSSoliman/DistilRoBERTa-Marian-Model-on-DJANGO. BERTMarian: https://www.huggingface.co/AhmedSSoliman/DistilBERT-Marian-Model-on-CoNaLa. https://www.huggingface.co/AhmedSSoliman/DistilBERT-Marian-Model-on-DJANGO. ELECTRAMarian: https://www.huggingface.co/AhmedSSoliman/ELECTRA-Marian-Model-on-CoNaLa. https://www.huggingface.co/AhmedSSoliman/ELECTRA-Marian-Model-on-DJANGO. LUKEMarian: https://www.huggingface.co/AhmedSSoliman/LUKE-Marian-Model-on-CoNaLa. https://www.huggingface.co/AhmedSSoliman/LUKE-Marian-Model-on-DJANGO.

## Declarations

**Conflict of interest** Not applicable.

**Ethics approval** Not applicable.

**Consent to participate** Not applicable.

**Consent for publication** Not applicable.

**Code availability** Implementation and availability of the code are in the following repository: https://github.com/AhmedSSoliman/Leveraging-Pretrained-Language-Models-for-Code-Generation.

## References

1. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444
2. Dai AM, Le QV (2015) Semi-supervised sequence learning. ArXiv arXiv:1511.01432
3. Elazar Y, Kassner N, Ravfogel S, Ravichander A, Hovy E, Schütze H, Goldberg Y (2021) Erratum: measuring and improving consistency in pretrained language models. Trans Assoc Comput Linguist 9:1407. https://doi.org/10.1162/tacl_x_00455
4. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, vol 30
5. Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L (2018) Deep contextualized word representations. ArXiv arXiv:1802.05365
6. Howard J, Ruder S (2018) Universal language model fine-tuning for text classification. In: Annual meeting of the association for computational linguistics. https://api.semanticscholar.org/CorpusID:40100965
7. Raffel C, Shazeer NM, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. ArXiv arXiv:1910.10683
8. Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L (2019) Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461
9. Rothe S, Narayan S, Severyn A (2020) Leveraging pre-trained checkpoints for sequence generation tasks. Trans Assoc Comput Linguist 8:264–280
10. LeClair A, Jiang S, McMillan C (2019) A neural model for generating natural language summaries of program subroutines. In: Proceedings of the 41st international conference on software engineering. ICSE '19. IEEE Press, pp 795–806. https://doi.org/10.1109/ICSE.2019.00087
11. Gad W, Alokla A, Nazih W, Salem AB, Aref M (2021) Dlbt-Deep learning-based transformer to generate pseudo-code from source code. CMC 70:3117–3123. https://doi.org/10.32604/cmc.2022.019884
12. Alokla A, Gad W, Nazih W, Aref M, Salem AB (2022) Retrieval-based transformer pseudocode generation. Mathematics 10(4):604. https://doi.org/10.3390/math10040604
13. Kaur P, Kumar H, Kaushal S (2023) Technology-assisted language learning adaptive systems: a comprehensive review. Int J Cogn Comput Eng 4:301–313. https://doi.org/10.1016/j.ijcce.2023.09.002
14. Javidpanah M, Javadpour A, Rezaei S (2021) ROOA: CloudIDE framework for extension development. Int J Cogn Comput Eng 2:165–170. https://doi.org/10.1016/j.ijcce.2021.09.003
15. Moss A, Muller H (2005) Efficient code generation for a domain specific language. In: Glück R, Lowry M (eds) Generative programming and component engineering. Springer, Berlin, pp 47–62
16. Guizzo G, Zhang J, Sarro F, Treude C, Harman M (2023) Mutation analysis for evaluating code translation. Empir Softw Eng 29:19
17. Athiwaratkun B, Gouda SK, Wang Z, Li X, Tian Y, Tan M, Ahmad WU, Wang S, Sun Q, Shang M, Gonugondla SK, Ding H, Kumar V, Fulton N, Farahani A, Jain S, Giaquinto R, Qian H, Ramanathan MK, Nallapati R, Ray B, Bhatia P, Sengupta S, Roth D, Xiang B (2023) Multi-lingual evaluation of code generation models. arXiv preprint arXiv:2210.14868
18. Dahal S, Maharana A, Bansal M (2021) Analysis of tree-structured architectures for code generation. In: Findings of the association for computational linguistics: ACL-IJCNLP 2021, pp 4382–4391
19. Qin P, Tan W, Guo J, Shen B, Tang Q (2021) Achieving semantic consistency for multilingual sentence representation using an explainable machine natural language parser (mparser). Appl Sci 24:11699
20. Tang Z, Shen X, Li C, Ge J, Huang L, Zhu Z, Luo B (2022) Ast-trans: code summarization with efficient tree-structured attention. In: 2022 IEEE/ACM 44th international conference on software engineering (ICSE), pp 150–162
21. Shin R, Lin CH, Thomson S, Chen C, Roy S, Platanios EA, Pauls A, Klein D, Eisner J, Van Durme B (2021) Constrained language models yield few-shot semantic parsers. arXiv preprint arXiv:2104.08768
22. Dong L, Lapata M (2016) Language to logical form with neural attention. arXiv preprint arXiv:1601.01280
23. Yin P, Neubig G (2017) A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696
24. Rabinovich M, Stern M, Klein D (2017) Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535
25. Yin P, Neubig G (2018) Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. arXiv preprint arXiv:1810.02720
26. Yin P, Neubig G (2019) Reranking for neural semantic parsing. In: Proceedings of the 57th annual meeting of the association for computational linguistics
27. Shin EC, Allamanis M, Brockschmidt M, Polozov A (2019) Program synthesis and semantic parsing with learned code idioms. In: Advances in neural information processing systems, vol 32
28. Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L (2020) Treegen: a tree-based transformer architecture for code generation. In: Proceedings of the AAAI conference on artificial intelligence, vol 34, pp 8984–8991
29. Xu FF, Jiang Z, Yin P, Vasilescu B, Neubig G (2020) Incorporating external knowledge through pre-training for natural language to code generation. arXiv preprint arXiv:2004.09015
30. Lano K, Xue Q (2023) Code generation by example using symbolic machine learning. SN Comput Sci 4:1–23
31. Le THM, Chen H, Babar MA (2020) Deep learning for source code modeling and generation. ACM Comput Surv (CSUR) 53:1–38
32. Norouzi S, Tang K, Cao Y (2021) Code generation from natural language with less prior knowledge and more monolingual data. In: Proceedings of the 59th Annual meeting of the association for

computational linguistics and the 11th international joint conference on natural language processing (volume 2: short papers), pp 776–785

33. Orlanski G, Gittens A (2021) Reading stackoverflow encourages cheating: adding question text improves extractive code generation. arXiv preprint arXiv:2106.04447
34. Beau N, Crabbé B (2022) The impact of lexical and grammatical processing on generating code from natural language. arXiv preprint arXiv:2202.13972
35. Wang Z, Cuenca G, Zhou S, Xu FF, Neubig G (2022) Mconala: a benchmark for code generation from multiple natural languages. arXiv preprint arXiv:2203.08388
36. Kusupati U, Ailavarapu VRT (2022) Natural language to code using transformers. ArXiv arXiv:2202.00367
37. Al-Hossami E, Shaikh S (2022) A survey on artificial intelligence for source code: a dialogue systems perspective. ArXiv arXiv:2202.04847
38. Ni P, Okhrati R, Guan S, Chang VI (2022) Knowledge graph and deep learning-based text-to-graphQL model for intelligent medical consultation chatbot. Inf Syst Front 2022:1–20
39. Kamath A, Das R (2018) A survey on semantic parsing. ArXiv arXiv:1812.00978
40. Gu J, Lu Z, Li H, Li VOK (2016) Incorporating copying mechanism in sequence-to-sequence learning. ArXiv arXiv:1603.06393
41. Iyer S, Konstas I, Cheung A, Zettlemoyer L (2018) Mapping language to code in programmatic context. In: Conference on empirical methods in natural language processing. https://api.semanticscholar.org/CorpusID:52125417
42. Xiao C, Dymetman M, Gardent C (2016) Sequence-based structured prediction for semantic parsing. In: Annual meeting of the association for computational linguistics. https://api.semanticscholar.org/CorpusID:16911296
43. Krishnamurthy J, Dasigi P, Gardner M (2017) Neural semantic parsing with type constraints for semi-structured tables. In: Conference on empirical methods in natural language processing. https://api.semanticscholar.org/CorpusID:1675452
44. Ling W, Blunsom P, Grefenstette E, Hermann KM, Kociský T, Wang F, Senior AW (2016) Latent predictor networks for code generation. ArXiv arXiv:1603.06744
45. Iyer S, Cheung A, Zettlemoyer L (2019) Learning programmatic idioms for scalable semantic parsing. In: Conference on empirical methods in natural language processing. https://api.semanticscholar.org/CorpusID:125969731
46. Nye M, Hewitt LB, Tenenbaum JB, Solar-Lezama A (2019) Learning to infer program sketches. ArXiv arXiv:1902.06349
47. Dong L, Quirk C, Lapata M (2018) Confidence modeling for neural semantic parsing. In: Annual meeting of the association for computational linguistics. https://api.semanticscholar.org/CorpusID:13686145
48. Chaurasia S, Mooney RJ (2017) Dialog for language to code. In: International joint conference on natural language processing. https://api.semanticscholar.org/CorpusID:217279086
49. Andreas J, Bufe J, Burkett D, Chen CC, Clausman J, Crawford J, Crim K, DeLoach J, Dorner L, Eisner J, Fang H, Guo A, Hall DLW, Hayes KD, Hill K, Ho D, Iwaszuk W, Jha S, Klein D, Krishnamurthy J, Lanman T, Liang P, Lin CH, Lintsbakh I, McGovern A, Nisnevich A, Pauls A, Petters D, Read B, Roth D, Roy S, Rusak J, Short BA, Slomin D, Snyder B, Striplin S, Su Y, Tellman Z, Thomson S, Vorobev AA, Witoszko I, Wolfe J, Wray AG, Zhang Y, Zotov A (2020) Task-oriented dialogue as dataflow synthesis. Trans Assoc Comput Linguist 8:556–571
50. Polozov O, Gulwani S (2015) Flashmeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications
51. Parisotto E, Mohamed A, Singh R, Li L, Zhou D, Kohli P (2017) Neuro-symbolic program synthesis. ArXiv arXiv:1611.01855
52. Bhupatiraju S, Singh R, Mohamed Ar, Kohli P (2017) Deep api programmer: Learning to program with apis. ArXiv arXiv:1704.04327
53. Balog M, Gaunt AL, Brockschmidt M, Nowozin S, Tarlow D (2017) Deepcoder: learning to write programs. ArXiv arXiv:1611.01989
54. Devlin J, Uesato J, Bhupatiraju S, Singh R, Mohamed Ar, Kohli P (2017) Robustfill: Neural program learning under noisy i/o. ArXiv arXiv:1703.07469
55. Xu Y, Dai L, Singh U, Zhang K, Tu Z (2019) Neural program synthesis by self-learning. ArXiv arXiv:1910.05865
56. Polosukhin I, Skidanov A (2018) Neural program search: Solving data processing tasks from description and examples. In: ICLR 2018
57. Li T, Zhang S, Li Z (2023) Sp-nlg: a semantic-parsing-guided natural language generation framework. Electronics 12:1772
58. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. arXiv preprint arXiv:2005.14165
59. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: A robustly optimized bert pretraining approach. ArXiv arXiv:1907.11692
60. Devlin J, Chang MW, Lee K, Toutanova K (2019) Bert: Pre-training of deep bidirectional transformers for language understanding. ArXiv arXiv:1810.04805
61. Soliman AS, Hadhoud MM, Shaheen SI (2022) Mariancg: a code generation transformer model inspired by machine translation. J Eng Appl Sci 69:1–23
62. Sanh V, Debut L, Chaumond J, Wolf T (2019) Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. CoRR arXiv:1910.01108
63. Clark K, Luong MT, Le QV, Manning CD (2020) Electra: Pre-training text encoders as discriminators rather than generators. ArXiv arXiv:2003.10555
64. Yamada I, Asai A, Shindo H, Takeda H, Matsumoto Y (2020) Luke: deep contextualized entity representations with entity-aware self-attention. In: EMNLP
65. Ross SI, Martinez F, Houde S, Muller M, Weisz JD (2023) The programmer's assistant: Conversational interaction with a large language model for software development. https://doi.org/10.1145/3581641.3584037
66. Poldrack RA, Lu T, Beguš G (2023) Ai-assisted coding: Experiments with gpt-4