Empirical Software Engineering (2023) 28:141
https://doi.org/10.1007/s10664-023-10372-1

A syntax-guided multi-task learning approach for Turducken-style code generation

Guang Yang[1] · Yu Zhou[1] · Xiang Chen[2] · Xiangyu Zhang[1] · Yiran Xu[1] · Tingting Han[3] · Taolue Chen[3]

Accepted: 27 July 2023

Abstract

Due to the development of pre-trained language models, automated code generation techniques have shown great promise in recent years. However, the generated code will not always adhere to syntactic constraints of the target language, especially in the case of Turducken-style code, where declarative code snippets are embedded within imperative programs. In this study, we summarize three significant challenges in regards to syntactic constraints: (1) the efficient representation of syntactic constraints, (2) the effective integration of syntactic information, and (3) the scalable syntax-first decoding algorithm. To address these challenges, we propose a syntax-guided multi-task learning approach TurduckenGen. Specifically, we first explicitly append the type information to the code tokens to capture the representation of syntactic constraints. Then we formalize code generation with syntactic constraint representation as an auxiliary task to enable the model to learn the syntactic constraints of the code. Finally, the syntactically correct code is selected accurately from the multiple candidates with the help of the compiler feedback. Extensive experiments and comprehensive analysis demonstrate the effectiveness and general applicability of our approach after being compared with six state-of-the-art baselines on two Turducken-style code datasets. Finally, we conducted a human study and found the code quality generated by our approach is better than baselines in terms of code readability and semantic similarity.

Keywords Syntactically-constrained code generation · Turducken-style code · Multi-task learning · CodeT5 · Abstract syntax tree

1 Introduction

Contemporary society is dependent on intricate software applications, and the development of such applications is a complex and prolonged process (Liu et al. 2023c). As the complexity of software increases, the development process becomes increasingly time-consuming and

Communicated by: Xin Xia

✉ Yu Zhou
zhouyu@nuaa.edu.cn

Extended author information available on the last page of the article

Published online: 14 October 2023

susceptible to errors. Moreover, the demand for acquiring proficiency in multiple programming languages is high, especially for novice developers, which further complicates software development (Xu et al. 2022). To mitigate these challenges, neural code generation has been proposed as a solution that aims to synthesize code snippets based on functional descriptions, with the potential to alleviate the burden of programmers during the development process.

Based on the type of target programming languages, the existing code generation tasks can be classified into imperative code generation and declarative code generation. Previous studies have demonstrated that existing methods can generate code with an accuracy of over 90% on datasets of declarative programs (Sun et al. 2020; Xuan et al. 2021), while the accuracy on datasets of imperative programs is less than 35% (Ahmad et al. 2021; Wang et al. 2021b). However, large software systems are rarely developed exclusively in a declarative language in practical software development. Instead, declarative programs are commonly embedded within imperative programs, which are normally referred to as *Turducken-style programs* (Liang et al. 2021). For example, in information management systems, developers often include SQL statements within imperative programs; in data mining systems, developers often incorporate regular expressions within imperative programs. A scientifically compelling and engineering-significant question is how to bring the good performance of automatic declarative program generation to real-world software development.

In general, automatic code generation can be beneficial in various aspects, such as increasing development efficiency, reducing potential faults in code, and enhancing code maintainability and readability. The automatic Turducken-style code generation method is primarily intended to assist programmers who need to simultaneously write declarative and imperative programs during the development process, thereby increasing development efficiency and productivity.

Turducken-style code is prevalent in real-world software development. Allamanis and Sutton (2013) observed that SQL is frequently used in conjunction with imperative programming languages, such as Java and Python. Moreover, we have retrieved Q&A posts on Stack Overflow with both keywords *SQL and Java* or *SQL and Python* in the past 10 years. The statistics are shown in Fig. 1 where we can find that the number of posts tagged by *SQL and Java* has been stable, while the number of posts tagged by *SQL and Python* has

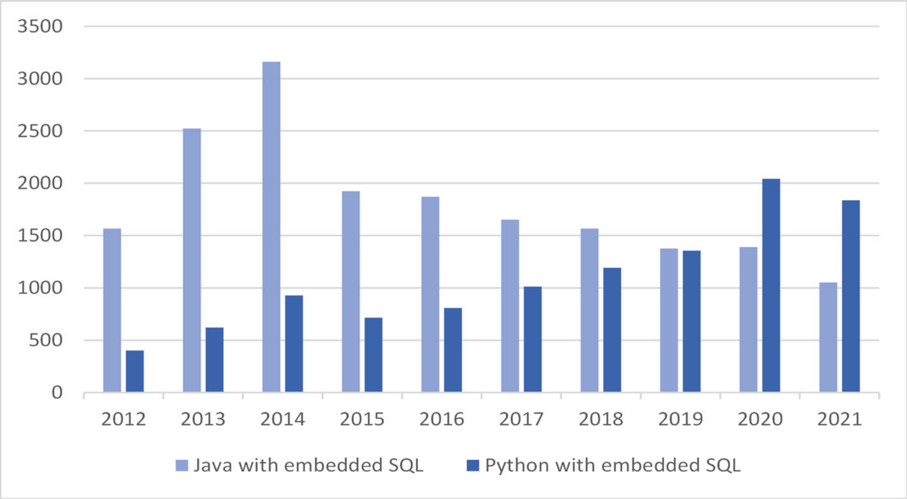


Fig. 1 The number of related posts in Stack Overflow by year

Table 1 A Post Related to Turducken-style Code Generation on Stack Overflow

Title	Using Spring JdbcTemplate to extract one string

Content	Can't seem to find a way to get one string from table using JdbcTemplate query.
	This is the table my sql returns:
	`ID  | STREET_NAME`
	`---------------------------`
	`1   | Elm street`
	Now how am I supposed to get the value of STREET_NAME. SQL always returns one row, so no need to worry about returning more than one row.
	But how can I extract "Elm street" from it using JdbcTemplate?

Tags	java, sql, hsqldb, jdbctemplate

increased steadily due to the growing popularity of Python. Table 1 presents a real-world post from Stack Overflow[1] In this post, the user had successfully constructed a SQL statement query, but encountered difficulties in embedding it into a Java program. This illustrates the challenges faced by developers who are not proficient in imperative programming languages when attempting to write Turducken-style code in the software development process.

In previous studies (Ahmad et al. 2021; Wang et al. 2021b; Niu et al. 2022; Chakraborty et al. 2022), a prevalent approach is to directly fine-tune pre-trained language models (PLMs) to generate code. However, this approach has a severe limitation, i.e., the generated code may not follow the syntactic rules of the targeted programming language (Dong et al. 2022), which can result in the failed compilation. This issue is particularly serious for domain-specific datasets. For example, in the field of exploit code generation, the two previous studies (Liguori et al. 2021; Yang et al. 2023) both exhibited that even the most advanced pre-training models can generate code that is lexically similar. Therefore, there is still room for improvement in terms of grammatical accuracy. In data science code generation, (Huang et al. 2022) found that about 8% of the errors in the generated code are due to syntax issues. Similarly, (Liang et al. 2021) have demonstrated the feasibility of using pre-trained language models to generate Turducken-style code but also admitted the less accurate syntactic conformation. The purpose of the automatic code generation model is to improve the efficiency of software development through automation and intelligent techniques. However, if the generated code has syntax problems, developers may need to spend a lot of time fixing the defects in the code. Therefore, generating code that conforms to syntactic constraints is still a significant area of research within the field.

Proposed solution. To address these issues, we propose a novel approach, i.e., TurduckenGen, which is based on CodeT5 (Wang et al. 2021b) and multi-task learning. TurduckenGen proposes corresponding solutions from three perspectives. (1) **The efficient representation of syntactic constraints.** TurduckenGen uses the syntax rule description language to represent syntactic constraints. TurduckenGen first parses the code into an abstract syntax tree, transforming it from a serialized text representation into a syntax tree with rich structural information. Then TurduckenGen proposes the syntax augmented traversal (SAT) algorithm, which can capture the representation of syntactic constraints by traversing the abstract syntax tree of the original code and explicitly appending the type information of the parent node to the code tokens. We refer to code with syntactically constrained representations as

[1] https://stackoverflow.com/questions/29286725/

syntax-guided code. (2) **The effective integration of syntactic information.** TurduckenGen formalizes syntax-guided code generation as an auxiliary task to enable the model to learn to generate code that adheres to syntactic constraints. Moreover, TurduckenGen employs a hard prompt method to facilitate the model's ability to distinguish the primary tasks and the auxiliary tasks. TurduckenGen employs the pre-training model CodeT5 to understand the user's functional description and learn both the target code and its corresponding code satisfying syntactic constraints. The encoder and decoder parameters are shared in the joint learning process, and the task-specific layers are customized in the final stage of mapping the semantic vector to the vocabulary. In order to incorporate the syntactic constraints learned in the auxiliary task into the primary task, TurduckenGen utilizes a gated linear unit for knowledge fusion. (3) **The scalable syntax-first decoding algorithm.** TurduckenGen proposes the syntax-first beam search (SF-Beam Search) method to maximize the ability of the model to generate syntax-correct code, where it can be easily integrated into existing pre-trained models. Specifically, the syntactically correct code can be selected accurately from the multiple candidates with the help of the compiler feedback.

To evaluate the effectiveness of TurduckenGen, we conduct experiments on two Turducken-style code datasets Lyra and Pisces, where Lyra (Liang et al. 2021) is obtained from GitHub repositories and annotated by human annotators, whereas Pisces is obtained by manually translating the python code in Lyra into the corresponding Java code through a crowd-sourcing approach. TurduckenGen is compared to six state-of-the-art baselines, including Transformer (Vaswani et al. 2017), CodeBERT (Feng et al. 2020), GraphCodeBERT (Guo et al. 2021), GPT (Radford et al. 2019), CodeGPT (Lu et al. 2021a), UniXcoder (Guo et al. 2022) and CodeT5 (Wang et al. 2021b), in terms of six automatic performance metrics (i.e., BLEU (Papineni et al. 2002), Weighted BLEU (Ren et al. 2020), Crystal BLEU (Eghbali and Pradel 2022), Code BLEU (Ren et al. 2020), Syntax Match (Ren et al. 2020), Syntax Exact Match (Liang et al. 2021), and Code Executable (Liang et al. 2021)). The comparison results show that TurduckenGen outperforms these baselines. Moreover, we design ablation studies to verify the effectiveness of multi-task learning and the method SF-Beam Search. Finally, from practitioners' perspectives on the generated Turducken-style code, we conduct a human evaluation to evaluate the quality of the generated code in terms of code readability and semantic similarity. The final results also show the competitiveness of our proposed approach.

The main contributions of our study can be summarized as follows.

- We propose a novel approach TurduckenGen for generating syntax-guided Turducken-style code. By utilizing CodeT5 and multi-task learning, we can effectively integrate syntax knowledge into the generated code, while our proposed SAT algorithm and SF-Beam Search method can improve the ability of the model to capture the representation of syntactic constraints and generate syntax-correct code.
- We conduct comprehensive experiments using both automatic evaluation metrics and human evaluation to assess the performance of TurduckenGen on two Turducken-style code datasets. The results of the evaluation indicate that TurduckenGen outperforms the state-of-the-art baselines.
- To facilitate the replication and reuse of TurduckenGen, we develop an Integrated Development Environment (IDE) plug-in and make our source code, trained models, as well as the datasets in the GitHub repository publicly available[2].

Structure The rest of the paper is organized as follows. Section 2 introduces the background of our work. Section 3 describes the framework of TurduckenGen and its key components.

[2] https://github.com/NTDXYG/TurduckenGen

Section 4 presents the experimental design and result analysis. Human Evaluation and potential threats to validity are given in Sections 6 and 7 respectively. Section 8 reviews the related work and emphasize the novelty of our study, and Section 9 concludes the paper.

2 Background

In this section, we provide an overview of the background of Turducken-style code, CodeT5, and multi-task learning.

2.1 Turducken-style Code

In the neural code generation task, programming languages are typically classified into two categories: declarative and imperative programming languages. Declarative programming is a paradigm that expresses the logic of a computation without describing its control flow (Lloyd 1994). It can simplify writing parallel programs (Bailey 2009). Examples of declarative languages include database query languages (e.g., SQL, XQuery), regular expressions, markup languages (e.g., HTML, XAML), functional programming (e.g. Haskell, Scheme), and configuration management systems (e.g., Json, YAML). On the other hand, imperative programming is a paradigm that focuses on describing how a program should accomplish a task without specifying all the details of how the program should achieve the result (Gifford and Lucassen 1986). Imperative programming focuses on describing how a program runs step-by-step, rather than on a high-level description of its intended outcome. Examples of imperative languages include C, C++, Java, Python, and so on.

Turducken-style code, which was first proposed by Liang et al. (2021), refers to a style of code where declarative programming is embedded within imperative programming. This type of code can be commonly found in real-world software systems. For example, SQL statements are often used with imperative programming in information management systems, and regular expressions are commonly used with imperative programming in data mining systems.

2.2 CodeT5

CodeT5 (Wang et al. 2021b) is an identifier-aware unified pre-trained encoder-decoder model for code understanding and generation, which is based on the Transformer architecture (Vaswani et al. 2017). It is pre-trained on the large-scale dataset CodeSearchNet (Husain et al. 2019) and BigQuery (Fernandes and Bernardino 2015), which includes eight programming languages (i.e., Go, Java, Javascript, PHP, Python, Ruby, C, and CSharp). To address out-of-vocabulary issues, CodeT5 utilizes a code-specific tokenizer based on the Byte-level BPE method. To take advantage of both bimodal and unimodal large-scale data, CodeT5 proposes four pre-training tasks: Masked Span Prediction (MSP), Identifier Tagging (IT), Masked Identifier Prediction (MIP), and Bimodal Dual Generation (BDG).

The MSP pre-training task takes a lexical perspective of the code, it utilizes a whole word span masking objective that randomly masks spans of arbitrary lengths and then predicts these masked spans combined with some sentinel tokens at the decoder. The IT and MIP pre-training tasks take a syntactic perspective of the code. It aims to notify the model with the knowledge of whether a code token is an identifier or not, and MIP masks all identifiers in the program language segment, which is inspired by obfuscation in the field of software engineering. The BDG pre-training task treats code generation and code summarization as dual tasks. To bridge the gap between the programming language and natural language, BDG leverages bimodal data to train the model for bidirectional conversion.

2.3 Multi-task Learning

Multi-task learning (MTL) is a machine learning paradigm. MTL aims to leverage useful information shared across multiple related tasks, which can improve the generalization performance on all tasks or enhance the model performance for a specific task by using auxiliary tasks (Liu et al. 2020a). MTL methods can be divided into hard or soft parameter sharing. In hard parameter sharing, the hidden layer is shared between all tasks, while keeping the output layer for several specific tasks. Intuitively, the more tasks that are learned simultaneously, the more representation of the tasks can be captured by the model, resulting in less risk of overfitting the primary task. In soft parameter sharing, each task has its own hidden layers and output layer. To ensure that the parameters of each task are similar, the distance between the parameters of each task is regularized.

In the Turducken-style code generation task, MTL has a natural advantage due to the limited labeled data. From a data augmentation perspective, MTL aggregates training samples from datasets of multiple tasks, which is especially beneficial for low-resource tasks whose labeled dataset is sometimes too small to sufficiently train a model. In most cases, the augmented training dataset alleviates the risk of overfitting and leads to more robust models (Sánchez-Cartagena et al. 2021). Furthermore, MTL provides additional performance gains compared to data augmentation approaches, due to the learned shared knowledge.

2.4 Prompt-based Learning

Prompt-based learning (Liu et al. 2023b) is a strategy to train large language models (LLMs) so the same model can be used for different downstream tasks without re-training. Traditional strategies for training large language models (such as GPT-3 and BERT) require the model to be pre-trained with unlabeled data and then fine-tuned for specific tasks with labeled data. In contrast, prompt-based learning models can autonomously tune themselves for different tasks by transferring domain knowledge (Wang et al. 2022a) introduced through prompts.

In general, a prompt is a snippet of natural language text that is added to unlabeled data during the pre-training phase (Gao et al. 2021). The art of writing useful prompts is called prompt engineering. According to the flexibility of the inserted prompt, prompt tuning techniques can be categorized into two types, i.e., hard prompt (Gu et al. 2022) and soft prompt (Li and Liang 2021). The hard prompt is a technique that modifies the model input by adding fixed natural language instruction (prompts), and the tokens in the soft prompt are continuous vectors that can be learned during the tuning stage.

3 Approach

Figure 2 illustrates the overall framework of TurduckenGen. It mainly consists of two phases: *Model Training Phase* and *Model Inference Phase*. *Model Training Phase* mainly includes prompt construction, syntax constraint representation, and syntax information integration. Specifically, we build a prompt that consists of the task description. We also represent syntax constraints using a syntax tree and integrate syntax information into the model, which is used

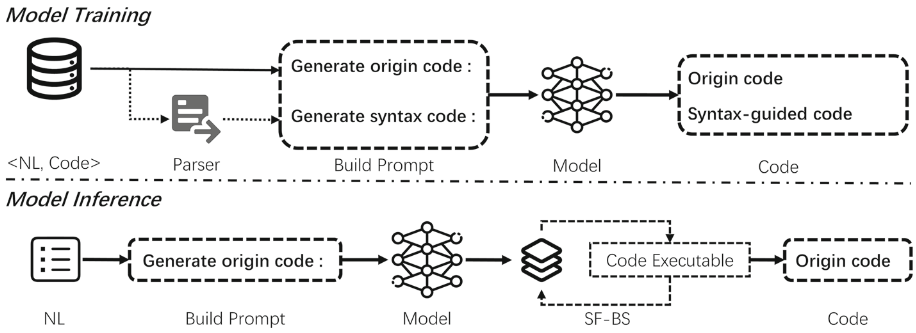


Fig. 2 Overview framework of our proposed approach

to guide the model to generate syntactically correct code. *Model Inference Phase* mainly introduces the scalable syntax-first decoding algorithm SF-Beam Search. This algorithm is used to generate code by considering the feedback provided by the compiler.

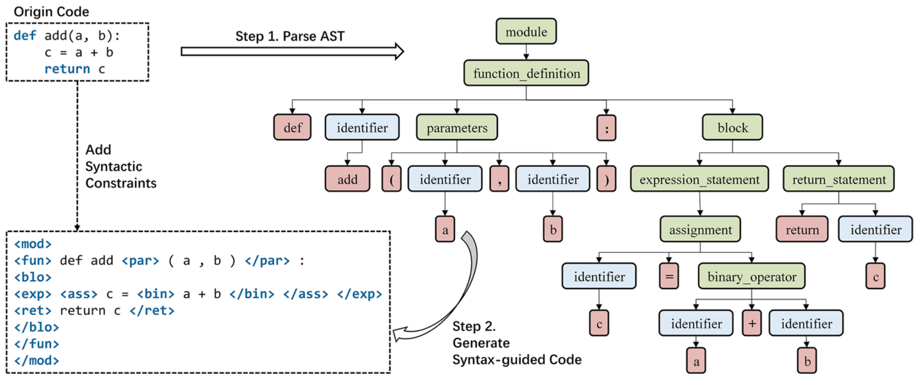
In the rest of this section, we provide more details on these two phases to better illustrate the novelty of our approach.

3.1 Model Training Phase

To train the model, we input the natural language functional descriptions of both the primary and auxiliary tasks. We use the template-based hard prompt method to allow the model to recognize and learn different tasks. Compared to continuous-based soft prompt methods, our method is prompted by a series of discrete tokens that are meaningful and understandable. For our task, we design the template by appending task-specific instructions as follows:

$$f_{prompt}([TASK], [X], [Y]) = \text{“}Generate\ [TASK]\ code\ :\ [X]\ [Y]\text{”} \qquad (1)$$

This template explains that the model is to generate the code [Y] according to functional description [X] under the task [TASK]. Specifically, the input of the auxiliary task is “Generate syntax code: [X]", and the output is a sequence that contains syntax information, which is referred to as “syntax-guided code". The input of the primary task is “Generate origin code:


Origin Code

```
def add(a, b):
    c = a + b
    return c
```

```
<mod>
<fun> def add <par> ( a , b ) </par> :
<blo>
<exp> <ass> c = <bin> a + b </bin> </ass> </exp>
<ret> return c </ret>
</blo>
</fun>
</mod>
```

Fig. 3 An example is used to illustrate how SAT traverses the AST of the source code

[X]", and the output is code sequence, which is referred to as “original code". Both the auxiliary task and the primary task use the model’s output and ground truth to calculate the loss function, and the model parameters are updated through backward propagation.

Figure 3 provides the process of the representation of syntactic constraints, and Fig. 4 provides an overview of the proposed network model architecture for the effective integration of syntactic information. In the rest of this section, we show the details of the representation of syntactic constraints (i.e., Syntax Augmented Traversal) and the integration of syntactic information (i.e., Model Architecture).

3.1.1 Syntax Augmented Traversal

To facilitate the pre-trained language model to generate code that adheres to the target programming language’s syntactic constraints, we propose a Syntax Augmented Traversal (SAT) algorithm for the efficient representation of syntactic constraints. This algorithm parses the AST of the source code and then converts it into a sequence of tokens annotated with syntactic information. The details of this algorithm can be found in Algorithm 1.

Step 1. From the root node, we use a pair of XML-like flags to annotate the syntax type.

Step 2. Then, we traverse the sub-trees of the root node by pre-order traversal. For a non-leaf node, a determination is made as to whether the node is a string, in which case the label ‘STR’ is utilized for the node. Conversely, the token value to the current node is employed. For a leaf node, its root node is incorporated into the XML-like flags.

Step 3. Traversing all sub-trees by recursion, this process continues until all nodes are traversed. The result of this process is a syntax-guided code sequence.

An illustrative example is shown in Fig. 3. Different colors are assigned to the nodes in the AST in the figure for illustrative purposes. Specifically, the red nodes indicate identifiers in the code, which embody the lexical information of the code. For this type of nodes, their value is retained. The blue nodes imply the type of these identifiers. For this type of nodes, their type information is ignored. The green nodes represent the complete syntax constraint information of the code (e.g., module, block, and return_statement), and the first three characters of its type information are extracted as its syntactic information. SAT appends these green nodes’ type information to the red nodes’ tokens explicitly to incorporate syntactic constraints. This approach enables the preservation of syntactic information and the ability to revert back to the original code.

3.1.2 Model Architecture

The architecture of TurduckenGen adheres to the Transformer (Vaswani et al. 2017) and pre-trained model CodeT5 (Wang et al. 2021b), which have been successfully utilized in code-related tasks. The architecture is primarily composed of three sub-modules. (1) **Encoder.** It aims to encode the functional description and employs multi-head self-attention to learn

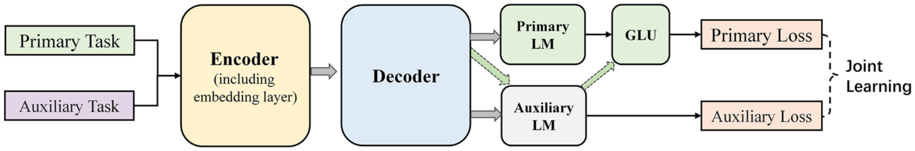


Fig. 4 The structure of the multi-task learning used in our neural network

Algorithm 1 Syntax Augmented Traversal (SAT) Algorithm.

```
Input  : AST $tree$ of Source Code Parsed by Tree-Sitter;
Output : Syntax-guided Code $syn\_code$;
$syn\_code \leftarrow \emptyset$;
if ! $tree.hasChild$ then
    if $tree.type$ == Identifier then
        if $tree.value$ is String then
            $syn\_code \leftarrow$ STR;
        else
            $syn\_code \leftarrow tree$.value;
    else
        $syn\_code \leftarrow tree$.value;
else
    $syn\_code \leftarrow< tree$.type>;
    for each $c \in tree.getChilds$ do
        $syn\_code \leftarrow syn\_code$ + SAT (c)
    $syn\_code \leftarrow</tree$.type>;
return $syn\_code$;
```

the sequential information of the functional description. (2) **Decoder.** It aims to generate Turducken-style code through the use of the self-attention layer and the encoder-decoder attention layer. (3) **Task-specific Layers.** They are designed to learn the mapping relationships of various tasks separately and to integrate the knowledge of the auxiliary task into the primary task using the Gated Linear Unit (GLU) network.

Encoder. TurduckenGen first generates embedding vectors that capture the semantic meaning of tokens and their position within a functional description. For a functional description X, TurduckenGen firstly tokenizes it into a sequence of sub-words (i.e., $X = x_1, \cdots, x_m$) by the BPE algorithm (Raffel et al. 2020), where m is the length of the tokenized sequence. For each sub-word, TurduckenGen generates an [1x768] embedding vector and combines it into a matrix to represent the meaningful relationship between a given token and the other tokens. In addition, to capture the position of each token within a functional description, TurduckenGen employs the relative position encoding technique (Raffel et al. 2020).

Encoder contains a stack of twelve layers of blocks, each block consists of two sub-components: a multi-head self-attention layer with relative position encoding and a feed-forward neural network. In contrast to the original Transformer, each sub-component in TurduckenGen is followed by a layer normalization and has a residual connection behind it. Given an input vector X, the first step is to create three main vectors (i.e., a query vector Q, a key vector K, and a value vector V). Further, TurduckenGen computes the relative positional information P, where P is an edge representation for the two inputs in dot-product operation to determine the positional information between tokens. P is supplied to the model as an additional component to the K and V, and the final attention score is computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q(K+P)^T}{\sqrt{d_k}}\right)(V+P) \tag{2}$$

Thus, for the embedding vector X, TurduckenGen first uses a layer normalization step and creates three vectors (Q, K, and V), which are then fed into Multi-head Attention Layer.

$$\text{Q, K, V} = \text{LayerNorm}(X) \tag{3}$$

Multi-head attention mechanisms obtain h different representations of (Q, K, and V). Then they concatenate the results and project the concatenation with a residual connection layer.

$$\mathrm{head}_i = \text{ Attention } \left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{4}$$

$$\mathrm{MultiHead}(Q, K, V) = \text{ Concat }_i\,(\mathrm{head}_i)\,W \tag{5}$$

$$\mathrm{X}_{\mathrm{atten}} = X + \mathrm{MultiHead}(Q, K, V) \tag{6}$$

Finally, TurduckenGen feeds $\mathrm{X}_{\mathrm{atten}}$ into the feed-forward layer (FFN) to generate the hidden state vector $\mathrm{X}_{\mathrm{hidden}}$.

$$\mathrm{X}_{\mathrm{hidden}} = \mathrm{LayerNorm}(\mathrm{X}_{\mathrm{atten}} + \mathrm{FFN}(X)) \tag{7}$$

Decoder In the decoder part, TurduckenGen also contains a stack of twelve layers of blocks, but each block consists of three sub-components: a masked multi-head self-attention layer with relative position encoding, a multi-head encoder-decoder attention layer with relative position encoding, and a feed-forward neural network. The feed-forward neural network is as same as that in the Encoder component, while the self-attention layer is similar to that in the encoder component except that it only deals with generated code tokens in the output sequence. Different from the self-attention layer, the encoder-decoder attention layer learns the relationship between the source functional description and the target code. The calculation of $\mathrm{Y}_{\text{cross-atten}}$ is similar to self-attention. The Queries matrix Q comes from the output of the self-attention layer and the Key matrix K and Values matrix V from the output of the encoder component $\mathrm{X}_{\mathrm{hidden}}$. Finally, TurduckenGen feeds $\mathrm{Y}_{\text{cross-atten}}$ into the FFN to generate the hidden state vector $\mathrm{Y}_{\mathrm{hidden}}$.

Task-specific Layers Task-specific output layers are employed to generate task-specific outputs, which contain Auxiliary LM layer, Primary LM layer, and GLU layer.

For the auxiliary task, we append the prompt 'Generate syntax code:' to the functional description, denoted as X_{aux}. Its corresponding decoder output vector is denoted as $\mathrm{Y}_{\text{aux-hidden}}$. The auxiliary LM layer is defined to produce the probability distribution of the syntax-guided code.

$$P_{aux} = Y_{\text{aux-hidden}} W^{aux} + b^{syn} \tag{8}$$

For the primary task, we append the prompt 'Generate origin code:' to the functional description, denoted as X_{pri}. Its corresponding decoder output vector is denoted as $\mathrm{Y}_{\text{pri-hidden}}$. Both the Primary LM layer and the Gated Linear Unit (GLU) network are defined to generate the probability distribution of the Turducken-style code.

$$P_{pri} = \left(Y_{\text{pri-hidden}} W^{pri} + b^{pri}\right) \otimes \sigma\left(Y_{\text{pri-hidden}} W^{aux} + b^{aux}\right) \tag{9}$$

The GLU employed here is intended to integrate the code syntax knowledge obtained from the Auxiliary LM layer into the Primary LM layer for the purpose of enforcing syntactic constraints. The GLU allows the network to selectively attend to the output of the semantic vectors in the Auxiliary LM and Primary LM layers for the purpose of knowledge interaction and selection. Furthermore, the GLU possesses non-linear characteristics while also maintaining a linear path for the gradient, thereby reducing the issue of vanishing gradients (Dauphin et al. 2017).

To learn these two tasks jointly, the parameters of TurduckenGen are trained to minimize the sum of the cross-entropy losses of the two tasks. The final loss function is presented as follows.

$$loss = \min_{\theta} \mathscr{L}_{Primary}(\theta) + \mathscr{L}_{Auxiliary}(\theta) \tag{10}$$

3.2 Model Inference Phase

During the model inference phase, the prompt template proposed in (1) is constructed to direct the trained model to generate the original code.

The model output is the probability of each token. In the decoding phase, we propose a scalable syntax-first beam search (SF-Beam Search) method to generate the Turducken-style code.

Recall that the beam search algorithm (Wiseman and Rush 2016) maintains a beam of k possible tokens $\left(token_t^i\right) i = 1^k$ at each time step t, where k represents the beam size. The possible tokens are updated as follows: for each token $token_t^i$, it adds each of the corresponding k most probable candidates, resulting in at most k Ã- k new tokens of size increased by 1. Then among these tokens, the k tokens with the highest likelihood are selected, thus obtaining the next step tokens $\left(token_{t+1}^i\right)_{i=1}^k$. However, the beam search algorithm aims at optimizing likelihood and ignores the code syntax, thus we cannot guarantee that the code decoded by beam search can be compiled and executed correctly.

Our proposed Syntax First Beam Search (SF-Beam Search) method is a simple yet effective solution that can be easily integrated into existing models. The SF-Beam Search method is based on our observation that there is a subtle gap between likelihood and code syntax. The likelihood of a code sequence indicates the maximum probability between tokens, but it does not ensure that the code is syntactically correct. To fill this gap, the SF-Beam Search method takes the k candidate code generated by the beam search algorithm and feeds them into a compiler tool in descending order of likelihood. If the current candidate code is executable, the SF-Beam Search method outputs it and terminates the loop. If all k candidate code are not executable, the SF-Beam Search method outputs the candidate code with the highest likelihood.

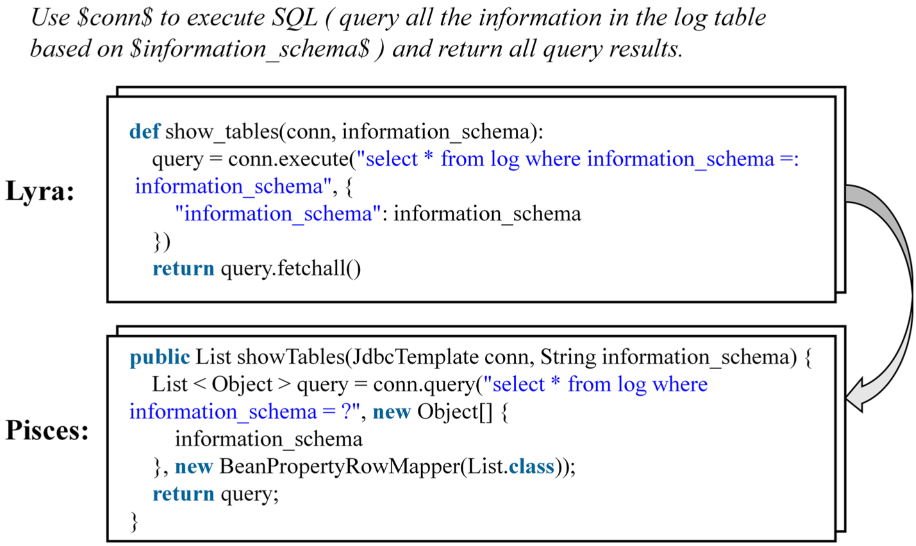
4 Experimental Setup

4.1 Research Questions

To evaluate the effectiveness of our proposed approach TurduckenGen, we want to investigate the following four research questions (RQs).

- **RQ1**: *Can our proposed approach TurduckenGen outperform the state-of-the-art baselines?*
- **RQ2**: *What is the contribution of the multi-task learning of TurduckenGen?*
- **RQ3**: *What is the benefit of using SF-Beam Search method of TurduckenGen?*
- **RQ4**: *What is the impact of different hard prompts of TurduckenGen?*

In RQ1, we want to compare TurduckenGen with previous pre-trained code generation methods in terms of automatic evaluation metrics. In RQ2, we want to adopt two variants of the multi-task learning methods to investigate the impacts of the multi-task learning method on TurduckenGen. In RQ3, we want to analyze the effectiveness between the SF-Beam Search method in TurduckenGen and other different decoding methods to demonstrate its benefit. In RQ4, we want to further explore the impact of different hard prompts on the performance of TurduckenGen.



Use $conn$ to execute SQL (query all the information in the log table based on $information_schema$) and return all query results.

```
def show_tables(conn, information_schema):
    query = conn.execute("select * from log where information_schema =:
 information_schema", {
        "information_schema": information_schema
    })
    return query.fetchall()
```

```
public List showTables(JdbcTemplate conn, String information_schema) {
    List < Object > query = conn.query("select * from log where
information_schema = ?", new Object[] {
        information_schema
    }, new BeanPropertyRowMapper(List.class));
    return query;
}
```

Fig. 5 An example in the Lyra and Pisces datasets

4.2 Datasets

The Lyra dataset (Liang et al. 2021) which is for mapping functional descriptions to Turducken-style code, only considers Python code with embedded SQL. To improve the diversity of Turducken-style code generation, we used a crowd-sourcing approach to translate the Python code in Lyra into its corresponding Java code and modified the functional description.

Finally, we construct a new high-quality dataset Pisces. The corpus contains 2,000 Turducken-style code and corresponding functional descriptions, which are in Java with embedded SQL.

In particular, the dataset Lyra is crawled from GitHub and rewritten code and comments by manual filtering. For the dataset Pisces, we hired two Java programmers with 3-5 years of development experience to label the dataset for 10 working days, with each of them spending 1-2 hours per day. They were paid a total of 1,200 Chinese yuan (CNY) as a reward. After the label work was completed, we used Maven to compile and check all the code to ensure the quality of the dataset. To ensure the quality of Lyra and Pisces, we also hired other six graduate student volunteers to review this dataset and discuss and fix the disputed data. Both Lyra and Pisces datasets have two different code styles (i.e., using native SQL and using Object Relational Mapper SQL). To ensure data consistency, ORM in Python uses SQLAlchemy[3], and ORM in Java uses JPA[4]. Figure 5 shows an example to illustrate our used datasets.

The statistical information (such as the count of code snippets, average tokens in the code, and average tokens in the functional descriptions) of the Lyra and Pisces datasets is shown in Table 2.

[3] https://www.sqlalchemy.org

[4] https://spring.io/projects/spring-data-jpa

Table 2 Statistical information of our used Lyra and Pisces datasets

Corpus	Type	Train	Valid	Test
Lyra	Count	1,600	200	200
	Avg. token in NL	47.18	47.42	47.27
	Avg. token in CODE	57.94	58.51	57.66
Pisces	Count	1,600	200	200
	Avg. token in NL	46.73	45.66	46.57
	Avg. token in CODE	79.15	89.20	84.93

4.3 Evaluation Metrics

To give a comprehensive evaluation, we use six metrics (i.e., *BLEU*, *Weight-BLEU*, *Crystal-BLEU*, *Syntax-Match*, *SyntaxExact-Match*, *Code-BLEU*, and *Code-Executable*), which have been widely used in previous related studies (Hussain et al. 2020, b, 2021; Yang et al. 2021b, 2022a), to automatically assess the quality of the generated code.

BLEU (Papineni et al. 2002), *Weight-BLEU* (Ren et al. 2020), and *Crystal-BLEU* (Eghbali and Pradel 2022) perform code similarity computation from the lexical perspective. Specifically, *BLEU* treats the generated code as the sentence in natural language, *Weight-BLEU* considers keywords in programming languages on the basis of *BLEU*, and *Crystal-BLEU* considers the inherent differences between source code and natural language, and optimizes the computation of N-grams in *BLEU*. *Syntax-Match* (Ren et al. 2020) and *SyntaxExact-Match* (Liang et al. 2021) perform code similarity computation from the syntax perspective. *Syntax-Match* (Liang et al. 2021) obtains all the sub-tree of the AST and then calculates the accuracy by comparing the candidate and reference sub-trees, while *SyntaxExact-Match* is the exact match of both AST and SQL statement, which also means the functional correctness of the generated code. *Code-BLEU* (Ren et al. 2020) is the mixed evaluation metric, which absorbs the strength of BLEU in the n-gram match and further injects code syntax via AST and code semantics via data-flow analysis.

In addition, *Code-Executable* (Wang et al. 2022b; Liang et al. 2021) is designed to calculate the executable rate of the generated code. Specially, for the Lyra dataset, we use Python 3.9[5] as the compiler and utilize the API provided by pylint[6] for automated checking. For the Pisces dataset, we use JDK 1.8[7] as compiler and utilize the command `mvn compile` provided by Maven[8] for automated checking.

To ensure the fairness of the experiment, we use the scripts provided by Ren et al. (2020) and Liang et al. (2021) to calculate the value of these metrics. These performance metrics range from 0 to 1, where larger values represent higher similarity.

4.4 Baselines

To our best knowledge, there are no models specifically designed for Turducken-style code generation. There exists only an empirical study (Liang et al. 2021) on Turducken-style code generation and we use their selected methods as our baselines. We evaluate the competi-

[5] https://www.python.org/downloads/release/python-390/

[6] https://github.com/PyCQA/pylint

[7] https://www.oracle.com/java/technologies/downloads/#java8

[8] https://maven.apache.org/

Table 3 Hyperparameter settings

Hyperparameter	Value	Hyperparameter	Value
Optimizer	AdamW	Seed	1234
Learning Rate	5e-5	Training batch size	16
Beam size	10	Validation batch size	16
Max input length	150	Max output length	256

tiveness of our proposed approach against seven state-of-the-art pre-trained code generation baselines. Specifically, we classify these baselines into four groups. The first group is the original Transformer (Vaswani et al. 2017), which trains the model from the scratch. The second group is Encoder-Only pre-trained models, including CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2021). The third group is Decoder-Only pre-trained models, including GPT (Radford et al. 2019) and CodeGPT (Lu et al. 2021a). The last group is Encoder-Decoder pre-trained model, including UniXcoder (Guo et al. 2022), CodeT5 (Wang et al. 2021b).

4.5 Experimental Settings

In our empirical study, Transformer is implemented by OpenNMT-py (Klein et al. 2017), other pre-trained models and corresponding tokenizers are loaded from the official repository Huggingface[9] and our used framework is Pytorch[10]. The hyper-parameters and their values used in our empirical study are summarized in Table 3, where optimizer, learning rate, and beam size are referenced to the parameters of CodeT5 (Wang et al. 2021b).

We implement TurduckenGen using PyTorch 1.8 and all the experiments run on a computer with an Intel(R) Xeon(R) Silver 4210 CPU and the Tesla V100 SXM2 GPU with 32 GB memory. The running OS platform is Linux OS.

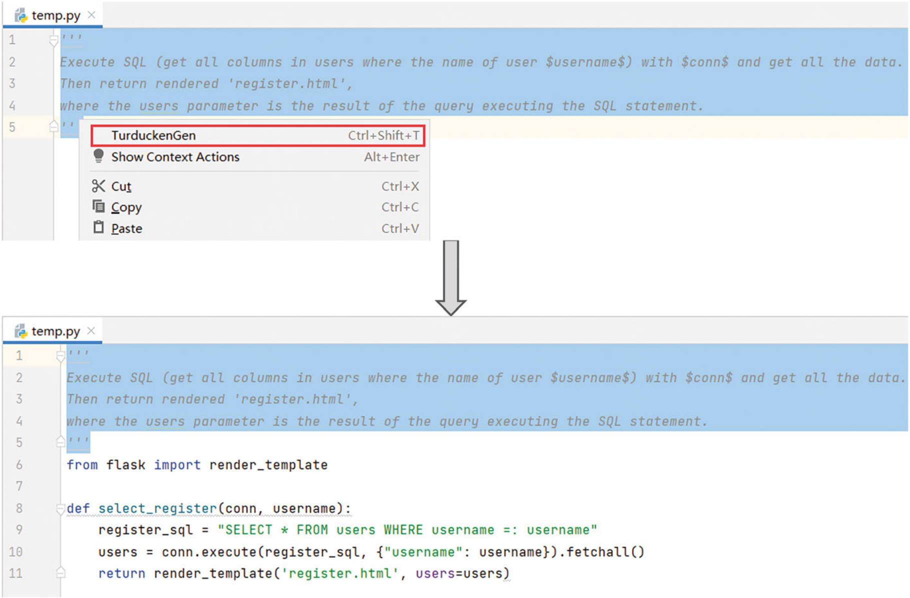
4.6 Tool Implementation

To help developers improve development efficiency, we have also developed an IDE plug-in based on our proposed approach. Our developed plug-in can be integrated into a range of IDEs under JetBrains[11] products (e.g. IDEA and PyCharm). We show the screenshot of our developed plug-in in Fig. 6. In our plug-in, developers first simply enter their functional requirements. Then they select these requirements with the mouse and right-click 'TurduckGen' in the shortcut menu. Finally, our plug-in can automatically generate the corresponding code.

[9] https://huggingface.co/models

[10] https://pytorch.org/

[11] https://www.jetbrains.com/



```
'''
Execute SQL (get all columns in users where the name of user $username$) with $conn$ and get all the data.
Then return rendered 'register.html',
where the users parameter is the result of the query executing the SQL statement.
'''
```

```
'''
Execute SQL (get all columns in users where the name of user $username$) with $conn$ and get all the data.
Then return rendered 'register.html',
where the users parameter is the result of the query executing the SQL statement.
'''
from flask import render_template

def select_register(conn, username):
    register_sql = "SELECT * FROM users WHERE username =: username"
    users = conn.execute(register_sql, {"username": username}).fetchall()
    return render_template('register.html', users=users)
```

Fig. 6 The screenshot of our developed IDE plug-in

Table 4 The comparison results between our proposed TurduckenGen and baselines

Corpus	Approach	*BLEU*	*Weight-B*	*Crystal-B*	*Code-B*	*S-M*	*SE-M*	*C-E*
Lyra	Transformer	38.97	39.52	10.00	46.46	48.46	0.00	15.00
	CodeBERT	63.29	63.93	43.70	68.77	74.04	6.50	59.50
	GraphCodeBERT	67.34	67.91	50.43	71.78	76.26	8.50	53.50
	GPT	72.76	73.17	60.88	77.26	81.36	21.50	92.50
	CodeGPT	73.23	73.65	63.51	77.41	80.37	24.00	96.00
	UniXcoder	70.45	70.33	62.10	70.17	70.73	18.50	70.00
	CodeT5	76.59	76.97	67.84	80.25	83.20	30.00	95.50
	TurduckenGen	**78.36**	**79.23**	**69.77**	**82.72**	**86.92**	**34.00**	**100.00**
Pisces	Transformer	41.66	42.12	10.91	49.35	52.86	1.00	74.00
	CodeBERT	57.65	58.48	32.67	61.05	61.81	0.00	93.50
	GraphCodeBERT	60.22	61.05	38.06	63.16	62.68	1.00	93.50
	GPT	61.88	62.87	45.60	64.50	62.64	1.00	89.50
	CodeGPT	62.75	63.64	47.28	65.32	63.08	1.50	89.50
	UniXcoder	61.27	62.25	46.35	64.45	62.94	1.50	94.50
	CodeT5	63.69	64.36	46.02	65.96	63.27	1.50	97.00
	TurduckenGen	**66.53**	**67.18**	**52.03**	**68.52**	**65.22**	**2.50**	**100.00**

5 Experimental Results

5.1 RQ1: Can our Proposed Approach TurduckenGen Outperform the State-of-the-Art Baselines?

We first apply our approach TurduckenGen and the baseline methods (i.e., Transformer, CodeBERT, GraphCodeBERT, GPT, CodeGPT, and CodeT5) on Lyra and Pisces, and compare their performance in terms of six automatic evaluation metrics. Then we use Wilcoxon signed-rank tests (Wilcoxon 1992) at the confidence level of 95% to check whether the performance differences are significant. Moreover, we use two samples to perform a qualitative analysis for our proposed approach.

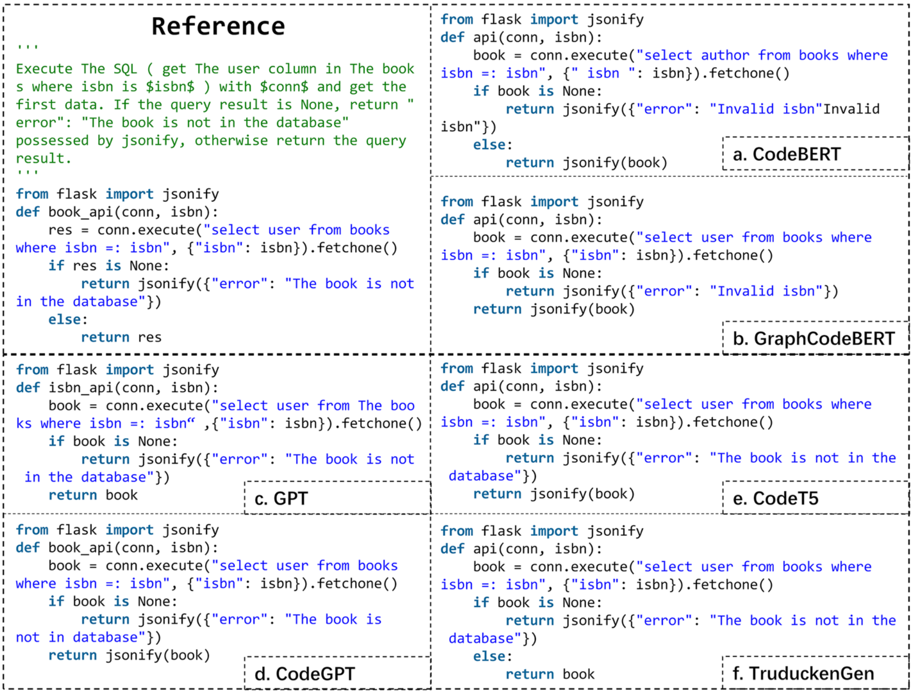
Table 4 shows the comparison results between TurduckenGen and the baselines. For both the Lyra and Pisces, TurduckenGen can achieve the best performance in terms of all the performance metrics. Specifically, in terms of *BLEU*, *Weight-BLEU*, *Crystal-BLEU*, and *Code-BLEU*, TurduckenGen can improve the performance by at least 2.31%, 2.94%, 2.84%, and 3.08% in Lyra, and by at least 4.46%, 4.38%, 10.05%, and 3.88% in Pisces. This demonstrates that TurduckenGen can generate more precise code at the lexical level. In terms of *Syntax-Match* and *SyntaxExact-Match* metrics, TurduckenGen can improve the performance by at least 4.47% and 13.33% in Lyra, and 3.08% and 66.67% in Pisces. This demonstrates that TurduckenGen is capable of generating code that not only matches syntax more accurately, but also with more similar semantics. In terms of *Code-Executable* metrics, TurduckenGen can achieve a 100% code execution rate on both Lyra and Pisces. This demonstrates that TurduckenGen generates more syntactically correct code.

In addition, we conduct a Wilcoxon signed-rank test (Wilcoxon 1992) with a significance level of 0.05 to assess the statistical significance of the performance differences between TurduckenGen and the state-of-the-art baseline CodeT5. The Wilcoxon signed-rank test is a two-sided test by default, which means that it tests the null hypothesis that there is no significant difference in performance between TurduckenGen and CodeT5, regardless of the direction of the difference. In our study, the null hypothesis is denoted as H_0, i.e., there is no significant difference between TurduckenGen and the state-of-the-art baseline in terms of metrics *BLEU*, *Weight-BLEU*, *Crystal-BLEU*, *Code-BLEU*, and *SyntaxMatch*. The results are shown in Table 5. In this table, we find all the p-values are smaller than $5e$-2. These statistical results lead to the rejection of the null hypothesis, which means that there exists a significant difference between our approach and baseline. Note that the results in Table 4 show that TurduckenGen can outperform other baselines. Therefore we can conclude that TurduckenGen achieves better performance than other baseline approaches significantly.

Finally, we show two examples with the ground-truth code and the code generated by TurduckenGen and all baselines in the Lyra dataset and the Pisces dataset respectively. As shown in Fig. 7, the code generated by CodeBERT contains syntax errors and fails to compile successfully. Additionally, the code generated by GraphCodeBERT deviates from

Table 5 The p-value between our proposed TurduckenGen and CodeT5 by using the Wilcoxon signed-rank test

Corpus	*BLEU*	*Weight-BLEU*	*Crystal-BLEU*	*Code-BLEU*	*Syntax-Match*
Lyra	2.6e-3	1.7e-4	4.2e-2	5.8e-4	1.4e-3
Pisces	3.0e-10	6.5e-11	3.1e-16	4.3e-11	7.2e-5



Reference

```
'''
Execute The SQL ( get The user column in The book
s where isbn is $isbn$ ) with $conn$ and get the
first data. If the query result is None, return "
error": "The book is not in the database"
possessed by jsonify, otherwise return the query
result.
'''
from flask import jsonify
def book_api(conn, isbn):
    res = conn.execute("select user from books
where isbn =: isbn", {"isbn": isbn}).fetchone()
    if res is None:
        return jsonify({"error": "The book is not
in the database"})
    else:
        return res
```

a. CodeBERT

```
from flask import jsonify
def api(conn, isbn):
    book = conn.execute("select author from books where
isbn =: isbn", {" isbn ": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "Invalid isbn"Invalid
isbn"})
    else:
        return jsonify(book)
```

b. GraphCodeBERT

```
from flask import jsonify
def api(conn, isbn):
    book = conn.execute("select user from books where
isbn =: isbn", {"isbn": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "Invalid isbn"})
    return jsonify(book)
```

c. GPT

```
from flask import jsonify
def isbn_api(conn, isbn):
    book = conn.execute("select user from The boo
ks where isbn =: isbn“ ,{"isbn": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "The book is not
 in the database"})
    return book
```

d. CodeGPT

```
from flask import jsonify
def book_api(conn, isbn):
    book = conn.execute("select user from books
where isbn =: isbn", {"isbn": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "The book is
not in database"})
    return jsonify(book)
```

e. CodeT5

```
from flask import jsonify
def api(conn, isbn):
    book = conn.execute("select user from books where
isbn =: isbn", {"isbn": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "The book is not in the
 database"})
    return jsonify(book)
```

f. TruduckenGen

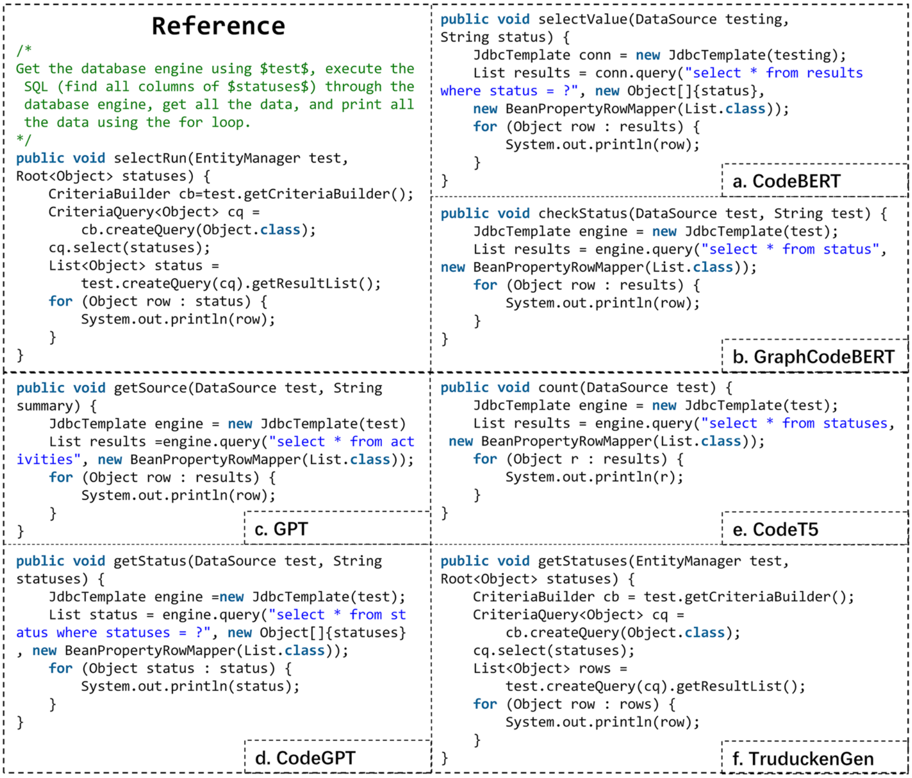
```
from flask import jsonify
def api(conn, isbn):
    book = conn.execute("select user from books where
isbn =: isbn", {"isbn": isbn}).fetchone()
    if book is None:
        return jsonify({"error": "The book is not in the
 database"})
    else:
        return book
```

Fig. 7 The example of Turducken-style code generated by TurduckenGen and baselines in the Lyra dataset

the user's specifications as indicated by error messages. An exception in the SQL statement is present in the code generated by GPT, specifically an extraneous `The`. Furthermore, the code generated by CodeGPT and CodeT5 may contain a bug, i.e., the generated code lacks the return branch of `else`, it will return `jsonify(None)` during code execution resulting in an exception. In contrast, the code generated by TurduckenGen can accurately reflects the functional requirements, though differing from the ground-truth code in terms of naming identifiers. As shown in Fig. 8, the code generated by CodeBERT, GraphCodeBERT, GPT, and CodeGPT all have problems in their SQL statements. While the code generated by CodeT5 is semantically consistent with the ground-truth code, it diverges in terms of code style, specifically in the utilization of native SQL statements as opposed to the ORM framework employed by the ground-truth code. Therefore, the code generated by TurduckenGen shows higher similarity to the ground-truth code, both in semantics and code style. Thus, we can observe that TurduckenGen can generate higher-quality code compared to baselines (Fig. 9).

Summary for RQ1

TurduckenGen can significantly outperform the baselines in terms of six performance metrics on both Lyra and Pisces.



Reference

```java
/*
Get the database engine using $test$, execute the
 SQL (find all columns of $statuses$) through the
 database engine, get all the data, and print all
 the data using the for loop.
*/
public void selectRun(EntityManager test,
Root<Object> statuses) {
    CriteriaBuilder cb=test.getCriteriaBuilder();
    CriteriaQuery<Object> cq =
        cb.createQuery(Object.class);
    cq.select(statuses);
    List<Object> status =
        test.createQuery(cq).getResultList();
    for (Object row : status) {
        System.out.println(row);
    }
}
```

```java
public void selectValue(DataSource testing,
String status) {
    JdbcTemplate conn = new JdbcTemplate(testing);
    List results = conn.query("select * from results
where status = ?", new Object[]{status},
    new BeanPropertyRowMapper(List.class));
    for (Object row : results) {
        System.out.println(row);
    }
}
```

a. CodeBERT

```java
public void checkStatus(DataSource test, String test) {
    JdbcTemplate engine = new JdbcTemplate(test);
    List results = engine.query("select * from status",
new BeanPropertyRowMapper(List.class));
    for (Object row : results) {
        System.out.println(row);
    }
}
```

b. GraphCodeBERT

```java
public void getSource(DataSource test, String
summary) {
    JdbcTemplate engine = new JdbcTemplate(test)
    List results =engine.query("select * from act
ivities", new BeanPropertyRowMapper(List.class));
    for (Object row : results) {
        System.out.println(row);
    }
}
```

c. GPT

```java
public void count(DataSource test) {
    JdbcTemplate engine = new JdbcTemplate(test);
    List results = engine.query("select * from statuses",
 new BeanPropertyRowMapper(List.class));
    for (Object r : results) {
        System.out.println(r);
    }
}
```

e. CodeT5

```java
public void getStatus(DataSource test, String
statuses) {
    JdbcTemplate engine =new JdbcTemplate(test);
    List status = engine.query("select * from st
atus where statuses = ?", new Object[]{statuses}
, new BeanPropertyRowMapper(List.class));
    for (Object status : status) {
        System.out.println(status);
    }
}
```

d. CodeGPT

```java
public void getStatuses(EntityManager test,
Root<Object> statuses) {
    CriteriaBuilder cb = test.getCriteriaBuilder();
    CriteriaQuery<Object> cq =
        cb.createQuery(Object.class);
    cq.select(statuses);
    List<Object> rows =
        test.createQuery(cq).getResultList();
    for (Object row : rows) {
        System.out.println(row);
    }
}
```

f. TruduckenGen

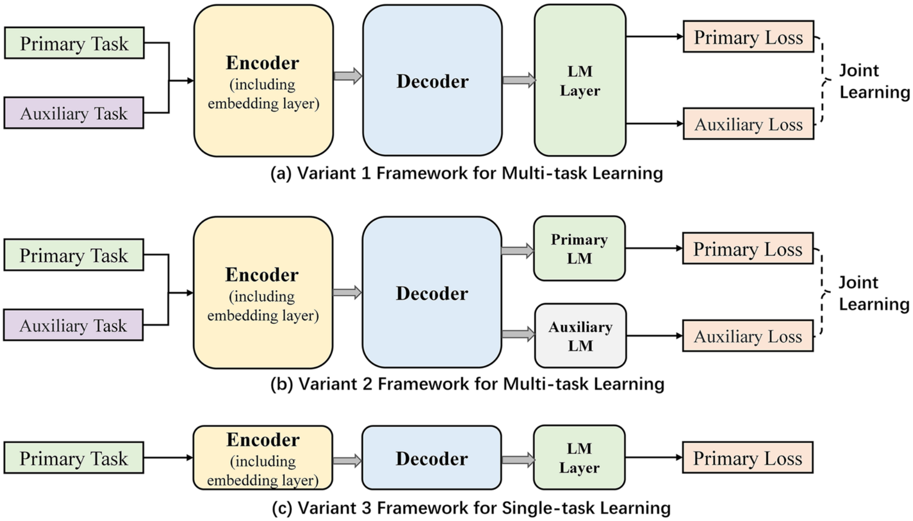
Fig. 8 The example of Turducken-style code generated by TurduckenGen and baselines in the Pisces dataset

5.2 RQ2: What is the Contribution of the Multi-task Learning of TurduckenGen?

To demonstrate the significance of each component in our proposed multi-task learning framework, we conduct a comparison of our approach with three of its variants:

- *Variant_1* does not introduce task-specific layers. This variant uses the same model and learns different tasks according to the prompt.
- *Variant_2* does not introduce additional GLU for knowledge interaction. In this variant, the task-specific layers of the two tasks are separated and do not affect each other.
- *Variant_3* does not introduce the auxiliary task and only consider the impact of the primary task. This variant model only contains CodeT5 with prompt and SF-beam search.

In Table 6, We can verify the effectiveness of task-specific layers and GLU by comparing TurduckenGen and variants. Similar to RQ1, we use *BLEU*, *Weight-BLEU*, *Crystal-BLEU*, *Code-BLEU*, *Syntax-Match*, *Syntax Exact-Match*, and *Code-Executable* to evaluate the effectiveness of our proposed approach and these three variants. As shown in Table 6, our proposed approach TurduckenGen shows superior performance compared to the other three variants on all metrics. Specifically, we can notice that when TurduckenGen removes only the GLU, there is a decrease in both lexical and syntactic similarity when compared to Variant 2 in Lyra and Pisces. On the contrary, Variant 1 shows comparable performance to that of Variant 2. When TurduckenGen removes the multitask learning framework and considers only the



(a) Variant 1 Framework for Multi-task Learning

(b) Variant 2 Framework for Multi-task Learning

(c) Variant 3 Framework for Single-task Learning

Fig. 9 The framework structure of the different variants

primary task, it can achieve lower performance than both variant 1 and variant 2 in terms of most metrics, especially on *Syntax-Match* and *Code-Executable*. These findings suggest that, within a multi-task learning framework, the inclusion or exclusion of task-specific layers does not have a significant impact on performance if there is no additional semantic interaction. Furthermore, we show the importance of multitask learning by comparing it with variant 3, which retains only prompt and SF-beam search compared to TurduckenGen, but is also can outperform CodeT5 with *Code-Executable*, which shows that the construction of prompt and compiler-based SF-beam search are also beneficial for CodeT5. Moreover, these results indicate that the incorporation of the GLU in TurduckenGen effectively embeds the grammatical knowledge acquired through the auxiliary task into the primary task, thereby improving the model's performance.

Table 6 The comparison results between our proposed TurduckenGen and two variants

Corpus	Approach	*BLEU*	*Weight-B*	*Crystal-B*	*Code-B*	*S-M*	*SE-M*	*C-E*
Lyra	Variant_1	77.67	78.86	69.11	80.96	85.45	29.00	99.00
	Variant_2	77.93	78.70	68.37	81.21	84.95	30.50	100.00
	Variant_3	76.97	77.32	68.10	81.03	84.25	30.50	97.50
	TurduckenGen	**78.36**	**79.23**	**69.77**	**82.72**	**86.92**	**34.00**	**100.00**
Pisces	Variant_1	64.84	66.51	50.74	66.48	64.52	2.00	99.50
	Variant_2	65.85	65.95	51.59	67.24	64.48	2.00	100.00
	Variant_3	64.04	65.29	49.75	66.30	63.86	2.00	98.00
	TurduckenGen	**66.53**	**67.18**	**52.03**	**68.52**	**65.22**	**2.50**	**100.00**

Table 7 The comparison results between SF-Beam Search and other decoding algorithms

Corpus	Approach	*BLEU*	*Weight-B*	*Crystal-B*	*Code-B*	*S-M*	*SE-M*	*C-E*
Lyra	Sampling Search	76.26	76.91	65.38	79.55	83.23	29.00	94.50
	Greedy Search	77.89	78.75	69.12	81.07	84.86	31.50	96.50
	Beam Search	78.03	78.94	69.17	81.44	86.08	32.50	97.00
	SF-Beam Search	**78.36**	**79.23**	**69.77**	**82.72**	**86.92**	**34.00**	**100.00**
Pisces	Sampling Search	64.52	64.48	49.48	66.23	63.98	1.50	96.50
	Greedy Search	65.12	66.04	50.98	66.94	64.50	2.00	97.00
	Beam Search	66.05	66.15	51.86	67.36	64.98	2.00	98.00
	SF-Beam Search	**66.53**	**67.18**	**52.03**	**68.52**	**65.22**	**2.50**	**100.00**

Summary for RQ2

Different from previous multi-task learning frameworks, we incorporate GLU after task-specific layers, which enables the incorporation of knowledge acquired through the auxiliary task into the primary task. Our study shows this setting can result in a positive impact on the performance of TurduckenGen.

5.3 RQ3: What is the Benefit of Using the SF-Beam Search Method of TurduckenGen?

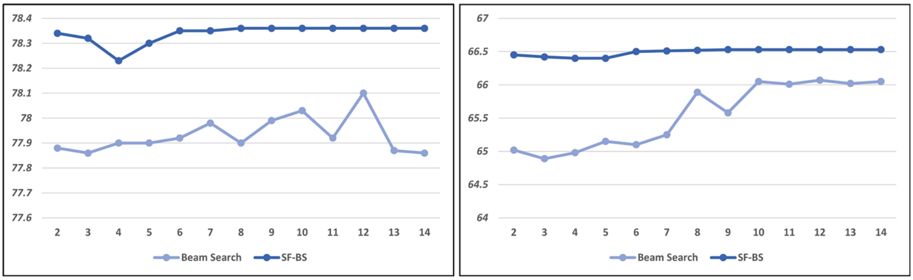
To investigate how the SF-Beam Search method affects the performance of TurduckenGen, we mainly consider three different decoding algorithms:

- *Sampling Search.* It is a sampling algorithm with randomness. Compared to the algorithm by probability, this sampling algorithm can introduce more randomness and is often present in dialogue generation task (Liu et al. 2020b).
- *Greedy Search.* It is a simple yet effective sampling algorithm that directly selects the token with the maximum probability for each output until a terminator appears or the maximum sentence length is reached.
- *Beam Search.* It is a heuristic graph search algorithm that keeps top-k token nodes with the highest probability at each step, which can reduce the space and time cost occupied by the search.

We show the comparison results in Table 7 and we find that the SF-Beam Search method works best both on Lyra and Pisces. Compared with Sampling Search, Greedy Search, and Beam Search, our SF-Beam Search method outperforms them in all metrics, especially in code execution rate. Therefore, the results show the effectiveness of our proposed GE-BS method.

Furthermore, as our SF-Beam Search method employs the beam search technique, the size of the beam size can have a direct impact on the performance of code generation. Thus, we conduct a comparison of the BLEU metric trends for code generated by the SF-Beam Search method and the beam search method for different beam sizes.

Later, we conduct a time cost analysis when considering program compiler cost during beam search in our SF-Beam Search method. Specifically, we find that without optimization,



(a) The hyper-parameter beam size in Lyra (b) The hyper-parameter beam size in Pisces

Fig. 10 The influence of the hyper-parameter beam size in Lyra and Pisces, where the horizontal axis denotes beam size and the vertical axis denotes the value of BLEU

the average time required for the program compiler is approximately 1∼2 seconds. This additional time cost resulted in a total time cost of approximately 15 seconds for each test sample when the beam search is set to 10. However, by using the multi-threading technology, the time cost for each test sample can be reduced to around 2∼3 seconds when the beam search is 10.

Finally, we conduct a series of experiments for the influence of the hyper-parameter beam size. As shown in Fig. 10, we can observe that the performance of code generated by the beam search method exhibits fluctuations as the beam size increases, while the code generated by the SF-Beam Search method demonstrates higher stability. Furthermore, the quality of code generated by the SF-Beam Search method is higher than that of the beam search method when analyzing all the results.

Summary for RQ3

In comparison to other decoding algorithms, the utilization of SF-Beam Search can result in a positive impact on the performance of Turducken-style code generation and demonstrates higher stability.

5.4 RQ4: What is the Impact of Different Prompt Methods of TurduckenGen?

In our study, we manually define the natural language tokens in hard prompt templates. To explore the impact of hard prompts and other prompt methods on TurduckenGen, we analyzed the performance of different hard prompts, soft prompts, and mixed prompts in terms of seven automatic metrics in this RQ:

- *[X] [Y]* means that the hard prompt is not used and relies only on the final LM layer for judgment.
- *[TASK] : [X] [Y]* means to use the name of the task as the hard prompt, without adding any other prompt token.
- *Generate Turducken-Style code under [TASK] : [X] [Y]* has the same meaning as the prompt used in TurduckenGen, but we add some tokens to make the prompt longer.
- *[SOFT]* $*$ *n : [X] [Y]* means to prepend several virtual tokens to the original input, which is referred as prefix-tuning (Li and Liang 2021) and is a typical method of soft prompt (Wang et al. 2022a).

- *[SOFT] ∗n+ Generate [TASK] code : [X] [Y]* means adding several virtual tokens before the hard prompt, which is referred as mixed prompt (Longpre et al. 2023).

We follow the setting of Wang et al. (2022a) and set the value of the parameter n in the soft prompts and mixed prompts to 4. We show the comparison results in Table 8. In this table, we can find that the different prompts have a slight impact on the performance, and using hard prompt templates for TurduckenGen can achieve the best performance on both Lyra and Pisces in terms of most metrics. More importantly, a well-designed hard prompt template is particularly important. A good prompt can stimulate the potential of the model, but this may require the experts to rely on their own experience for tuning. Moreover, our results indicate a similarity in the model performance between the incorporation of solely the soft prompt and the absence of any prompt. However, the introduction of a mixed prompt can lead to optimal model performance in several metrics. This result shows the significance of the hard prompt and the possibility of improving model performance through the implementation of the soft prompt as a precursor to a well-designed hard prompt.

Summary for RQ4

We demonstrate the importance of different prompt methods and find only well-designed prompts can have a positive impact on the performance of TurduckenGen.

6 Human Evaluation

In RQ1, we conducted performance comparisons automatically in terms of six performance metrics. However, in the absence of test cases, these automatic performance metrics may not truly reflect the semantic similarity between different code snippets. To alleviate this issue, we further conducted a human evaluation to verify the effectiveness of our proposed approach TurduckenGen. In our human study, we only compare TurduckenGen with CodeT5, which can achieve the best performance in all baselines.

We refer to the methodology used by Hu et al. (2022) and Yang et al. (2022c) to conduct the human evaluation in the code generation task. In the human study, we evaluate the quality of the generated code from three aspects:

- `Code Readability.` It evaluates the code readability of the generated code. For example, Java code should follow DRAFT[12] and Python code should follow PEP 8[13].
- `Semantic Similarity.` It evaluates the semantic similarity between the generated code and the reference code since the code snippets with the same semantics may differ at the lexical level.
- `User Preference.` It evaluates the users' preference score between the reference code and code generated by the two methods, which can represent the real preferences of the users and avoid the bias caused by the provided ground truth.

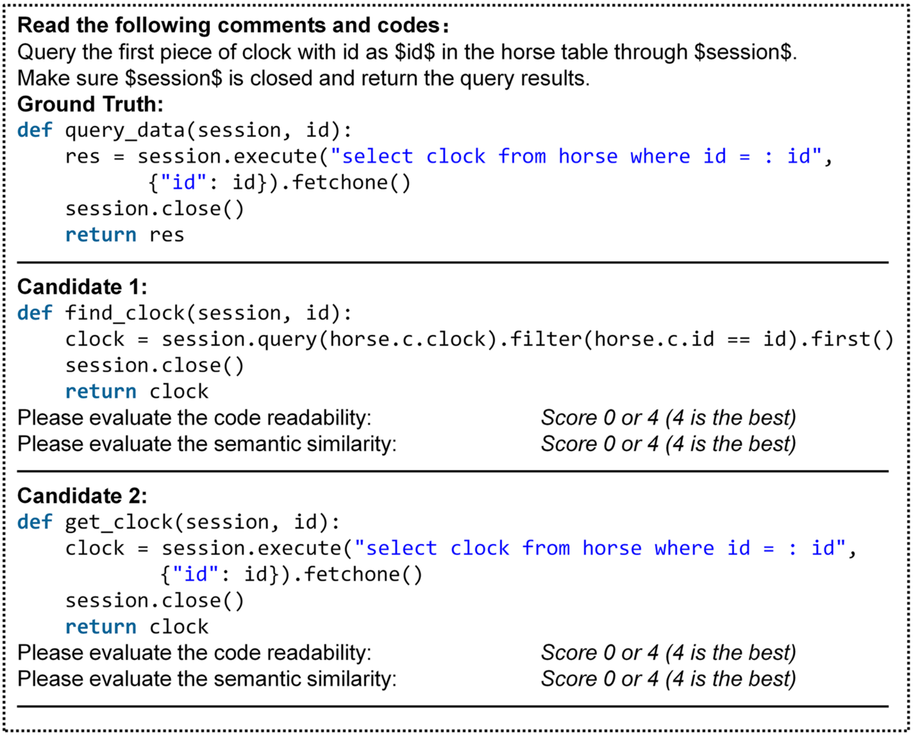
The scores of `Code Readability` and `Semantic Similarity` range from 0 to 4 (the higher the better) and all these scores are integers. In consideration of `User Preference`, we designed it as a choice question, where volunteers were asked to select the one or more codes they think best from three candidate codes (i.e., the reference code,

[12] http://cr.openjdk.java.net/~alundblad/styleguide/index-v6.html

[13] https://peps.python.org/pep-0008

Table 8 The comparison results between different prompts

Corpus	Approach	*BLEU*	*Weight-B*	*Crystal-B*	*Code-B*	*S-M*	*SE-M*	*C-E*
Lyra	*[X] [Y]*	78.16	78.89	69.54	82.45	84.97	29.00	**100.00**
	[TASK] : [X] [Y]	77.68	78.45	68.90	81.92	84.06	33.00	99.50
	Generate Turducken-Style code under [TASK] : [X] [Y]	78.28	79.06	69.16	82.57	85.42	**34.00**	99.00
	[SOFT] ∗ n : [X] [Y]	78.19	79.08	69.49	82.47	85.04	32.00	99.50
	[SOFT] ∗n+ Generate [TASK] code : [X] [Y]	78.32	79.24	69.69	**82.73**	86.78	**34.00**	**100.00**
	Generate [TASK] code : [X] [Y]	**78.36**	**79.23**	**69.77**	**82.72**	**86.92**	**34.00**	**100.00**
Pisces	*[X] [Y]*	65.84	66.23	51.48	67.93	64.66	2.00	**100.00**
	[TASK] : [X] [Y]	65.26	65.89	51.38	67.58	64.44	2.00	**100.00**
	Generate Turducken-Style code under [TASK] : [X] [Y]	66.04	66.76	51.23	67.94	64.50	**2.50**	99.50
	[SOFT] ∗ n : [X] [Y]	65.28	66.15	51.44	67.87	64.51	2.00	99.00
	[SOFT] ∗n+ Generate [TASK] code : [X] [Y]	**66.74**	67.12	**52.09**	68.12	65.12	**2.50**	99.00
	Generate [TASK] code : [X] [Y]	66.53	**67.18**	52.03	**68.52**	**65.22**	**2.50**	**100.00**



Read the following comments and codes:
Query the first piece of clock with id as id in the horse table through $session$.
Make sure $session$ is closed and return the query results.

Ground Truth:

```
def query_data(session, id):
    res = session.execute("select clock from horse where id = : id",
           {"id": id}).fetchone()
    session.close()
    return res
```

Candidate 1:

```
def find_clock(session, id):
    clock = session.query(horse.c.clock).filter(horse.c.id == id).first()
    session.close()
    return clock
```

Please evaluate the code readability: *Score 0 or 4 (4 is the best)*
Please evaluate the semantic similarity: *Score 0 or 4 (4 is the best)*

Candidate 2:

```
def get_clock(session, id):
    clock = session.execute("select clock from horse where id = : id",
            {"id": id}).fetchone()
    session.close()
    return clock
```

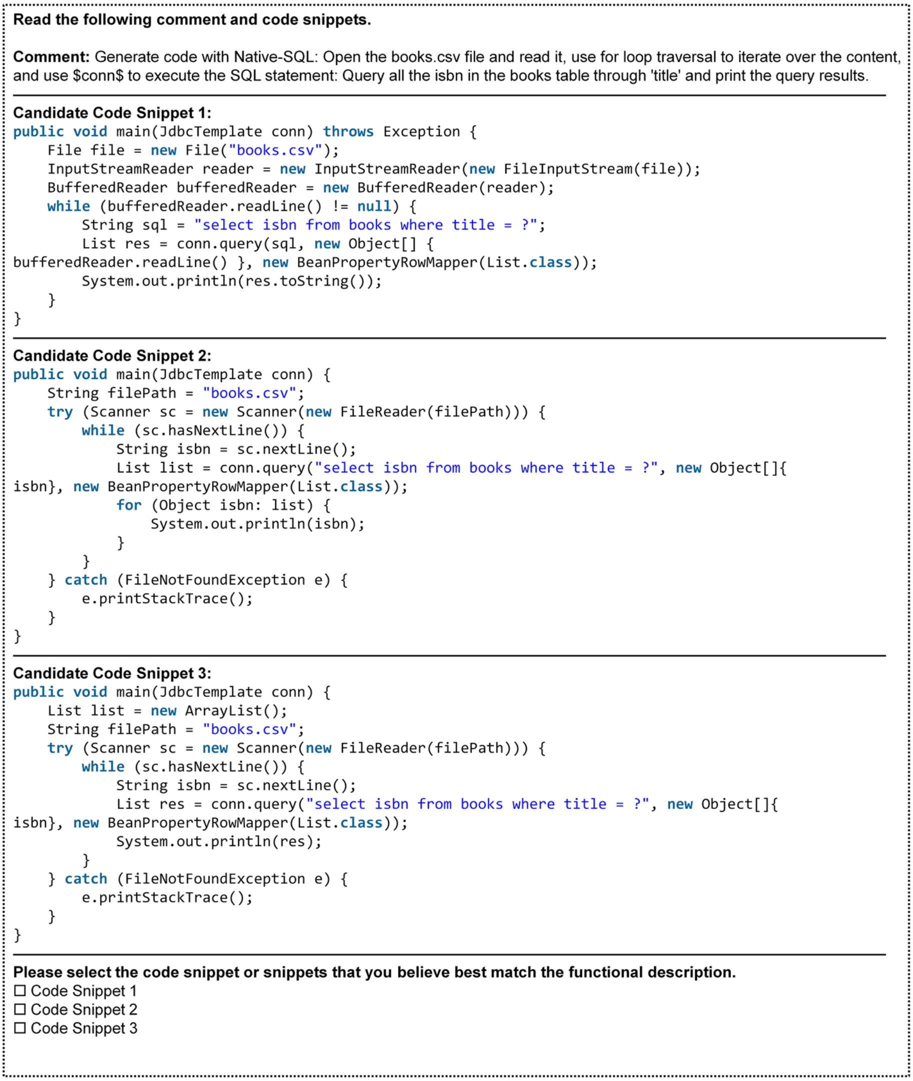
Please evaluate the code readability: *Score 0 or 4 (4 is the best)*
Please evaluate the semantic similarity: *Score 0 or 4 (4 is the best)*

Fig. 11 A questionnaire used to evaluate Code Readability and Semantic Similarity in our human evaluation

the code generated by CodeT5, and the code generated by TurduckenGen). We invite five volunteers, who have 3∼5 years of Java and Python experience and have good English reading ability. Then we paid a certain fee for each participant in their code evaluation work. Due to the high cost of manually analyzing all these samples in the testing set, we randomly select 100 Turducken-style code snippets and functional descriptions by following Hu et al. (2021). Specifically, we randomly selected 50 samples from the Lyra and 50 samples from the Pisces. To avoid any potential bias (i.e., answer leakage) caused by previously sampled individuals in the Code Readability and Semantic Similarity stages, we randomly selected new 50 samples for the User Preference questionnaire for the same group of volunteers.

For each code snippet, we generate a questionnaire for each participant, which is shown in Figs. 11 and 12.

In the questionnaire used to evaluate Code Readability and Semantic Similarity, there are two code snippets generated by TurduckenGen and the baseline CodeT5 respectively. Each participant is asked to score each code in terms of code readability and semantic similarity for two code snippets generated by TurduckenGen and the baseline CodeT5 respectively. In the questionnaire used to evaluate user preference, there are three code snippets, which contain the reference code, the code generated by TurduckenGen, and the code generated by CodeT5 respectively. Since there is the possibility of the exact match between these three code snippets, each participant is asked to choose one or more codes that best match the functional description.



Read the following comment and code snippets.

Comment: Generate code with Native-SQL: Open the books.csv file and read it, use for loop traversal to iterate over the content, and use $conn$ to execute the SQL statement: Query all the isbn in the books table through 'title' and print the query results.

Candidate Code Snippet 1:

```java
public void main(JdbcTemplate conn) throws Exception {
    File file = new File("books.csv");
    InputStreamReader reader = new InputStreamReader(new FileInputStream(file));
    BufferedReader bufferedReader = new BufferedReader(reader);
    while (bufferedReader.readLine() != null) {
        String sql = "select isbn from books where title = ?";
        List res = conn.query(sql, new Object[] {
bufferedReader.readLine() }, new BeanPropertyRowMapper(List.class));
        System.out.println(res.toString());
    }
}
```

Candidate Code Snippet 2:

```java
public void main(JdbcTemplate conn) {
    String filePath = "books.csv";
    try (Scanner sc = new Scanner(new FileReader(filePath))) {
        while (sc.hasNextLine()) {
            String isbn = sc.nextLine();
            List list = conn.query("select isbn from books where title = ?", new Object[]{
isbn}, new BeanPropertyRowMapper(List.class));
            for (Object isbn: list) {
                System.out.println(isbn);
            }
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Candidate Code Snippet 3:

```java
public void main(JdbcTemplate conn) {
    List list = new ArrayList();
    String filePath = "books.csv";
    try (Scanner sc = new Scanner(new FileReader(filePath))) {
        while (sc.hasNextLine()) {
            String isbn = sc.nextLine();
            List res = conn.query("select isbn from books where title = ?", new Object[]{
isbn}, new BeanPropertyRowMapper(List.class));
            System.out.println(res);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

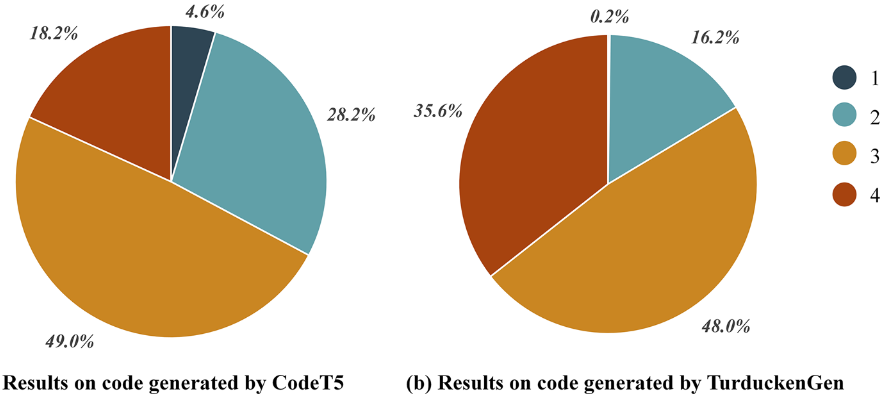
Please select the code snippet or snippets that you believe best match the functional description.

☐ Code Snippet 1
☐ Code Snippet 2
☐ Code Snippet 3

Fig. 12 A questionnaire used to evaluate User Preference in our human evaluation

During the code quality evaluation process, the participants can discuss and resort to external resources (e.g., Wikipedia and Q&A websites). To ensure the fairness of the comparison, the participants do not know which code is generated by which approach, and the order of questionnaires is different for different participants. To guarantee the code evaluation quality, we need each participant to review only 25 code snippets in half a day to avoid fatigue.

Figures 13 and 14 show the statistical results of this feedback in terms of code readability and semantic similarity. In these two figures, the left and right sub-figures show the votes for the Turducken-style code generated by CodeT5 and TurduckenGen, respectively. In terms of code readability, we find that 83% of the human ratings for the code generated by TurduckenGen are not less than 3, which means that they are considered to have good



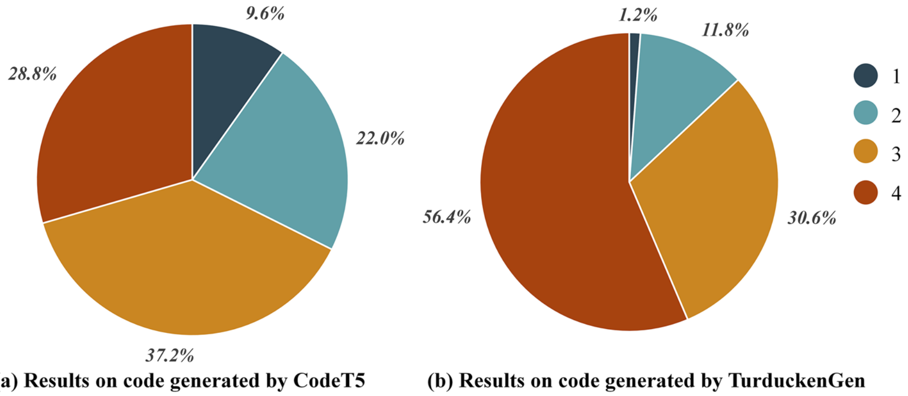
(a) Results on code generated by CodeT5 **(b) Results on code generated by TurduckenGen**

Fig. 13 Rating distribution of human evaluation in terms of code readability

programming specifications and can be easily comprehended by the user. In terms of semantic similarity, we find that 87% of the human ratings for the code generated by TurduckenGen are not less than 3, which means they are considered to have good quality and can be used as code without major modifications.

We also compute the average score of the participants' feedback and the results are shown in Table 9. In this table, values in parentheses indicate the degree of consistency of the scores marked by the participants using Kendall's coefficient of concordance (Legendre 2005). Notice the value of Kendall's coefficient of concordance ranges from 0 to 1, with larger values indicating higher concordance. Based on the results, we can find that for the Lyra, TurduckenGen can outperform the approach CodeT5 by 0.340 and 0.516 respectively in terms of code readability and semantic similarity. For example, in the corpus Pisces, TurduckenGen can outperform the approach CodeT5 by 0.424 and 0.720 respectively in terms of code readability and semantic similarity. In addition, all Kendall's coefficient of concordance in our human evaluation is larger than 0.5, which indicates their scores are about the same degree of concentration.

As for user preference, we first calculate the score for each volunteer, which is the proportion of times they chose the reference code, the code generated by CodeT5, and the code



(a) Results on code generated by CodeT5 **(b) Results on code generated by TurduckenGen**

Fig. 14 Rating distribution of human evaluation in terms of semantic similarity

Table 9 Results of our human study in terms of code readability, semantic similarity (values in parentheses indicate Kendall's coefficient of concordance in our human evaluation), and user preference

Dataset	Approach	Code Readability	Semantic Similarity	User Preference
Lyra	Reference	-	-	86.6%
	CodeT5	3.152 (0.533)	3.136 (0.788)	52.4%
	TurduckenGen	**3.492** (0.553)	**3.652** (0.843)	68.5%
Pisces	Reference	-	-	90.4%
	CodeT5	2.464 (0.771)	2.472 (0.884)	56.2%
	TurduckenGen	**2.888** (0.666)	**3.192** (0.756)	64.7%

generated by TurduckenGen over all questionnaires. Then, we average the scores of all volunteers to obtain the final score. Based on the results, we can find that regardless of Lyra or Pisces, volunteers tend to prefer the reference code. Secondly, volunteers generally believe that the code generated by TurduckenGen is better than the code generated by CodeT5, indicating that volunteers are more inclined towards the code generated by TurduckenGen. Therefore, our human study can further verify the competitiveness of TurduckenGen.

7 Threats to Validity

7.1 Construct Validity

The first construct validity concerns the appropriateness of our evaluation metrics. To mitigate this construct threat, we consider six evaluation metrics. Additionally, we also compute the p-value by utilizing the Wilcoxon signed-rank test to further verify the statistical significance of our proposed approach. The second construct is the semantic correctness of the code generated by TurduckenGen. Due to the lack of high-quality test cases, we mainly conduct a human evaluation and assess the effectiveness of TurduckenGen by taking into account the code readability and semantic similarity of the generated Turducken-style code.

7.2 Internal Validity

The first internal threat is the potential defects in the implementation of our proposed method. To alleviate this threat, we first check the code carefully and use mature libraries, such as PyTorch and Transformers. The second internal threat is the implementation of the baseline methods. To alleviate this threat, we try our best to fine-tune the pre-trained models (i.e., CodeBERT[14], GraphCodeBERT[15], GPT[16], CodeGPT[17][18], and CodeT5[19]). Moreover, the parameters for both CodeT5 and TurduckenGen are the same, as shown in Table 3. For the other baseline models, we followed the parameter setting of the previous study (Liang

[14] https://huggingface.co/microsoft/codebert-base

[15] https://huggingface.co/microsoft/graphcodebert-base

[16] https://huggingface.co/gpt2

[17] https://huggingface.co/microsoft/CodeGPT-small-java

[18] https://huggingface.co/microsoft/CodeGPT-small-py

[19] https://huggingface.co/Salesforce/codet5-base

et al. 2021). The third internal threat is that our proposed architecture is specifically designed for pre-trained encoder-decoder models, such as CodeT5. Our approach is not applicable to other pre-trained models, such as CodeBERT, GPT, GraphCodeBERT, etc. This is because CodeBERT-like models are only pre-trained on the encoder side, while our multi-task learning approach operates on the decoder side. On the other hand, GPT-like models are only pre-trained on the decoder side, while the prompt method proposed by our approach relies more heavily on the encoder side. Therefore, we believe that our method is only suitable for pre-trained encoder-decoder models, such as CodeT5.

7.3 External Validity

External validity refers to the extent to which the findings of a study can be generalized to other datasets. In our study, we mainly focused on Turducken-style code generation. The main difference between generating Turducken-style code and general code is the nested nature of Turducken-style code (i.e. declarative programs are embedded in imperative programs). In addition, it is more common for programmers to write Turducken-style code in real business development scenarios. However, our approach is essentially independent of specific programming languages, meaning that it has a certain degree of generality. By adjusting the training data and fine-tuning the model, our proposed approach can be applied to general code generation tasks. Meanwhile, the corresponding compiler needs to be replaced for a specific programming language in order to parse the AST and perform the SF-Beam Search decoding.

8 Related Work

In this section, we mainly summarize related studies for automatic code generation, multi-task learning for code intelligence, and abstract syntax tree traversal methods. After related work analysis, we emphasize the novelty of our study.

8.1 Automatic Code Generation

In the field of automatic code generation for declarative programming, most researchers focused on the automatic generation of SQL statements. Dahl et al. (1994) were the first to investigate this area and shared the ATIS dataset for the Airline Travel Query domain. Then, a growing number of researchers explored and constructed various datasets for automatic SQL generation. For example, some datasets (Zelle and Mooney 1996; Iyer et al. 2017) focus on a single domain and other datasets (Zhong et al. 2017; Yu et al. 2018b, 2019b, a) encompass multiple domains and cover diverse SQL types. Early studies (Popescu et al. 2003; Mahmud et al. 2015) primarily employed rule-based approaches. However, these methods required pre-defined templates for SQL statements and had poor scalability. With the advent of neural network models, the majority of work in this field has shifted towards generating more flexible SQL statements with a wider range of styles. Yu et al. (2018b) modeled SQL generation as a neural translation task and employed an RNN-based Seq2Seq model for this purpose. Yu et al. (2018a) additionally incorporated Schema Linking in the encoder and added Sketch information in the decoder. Bogin et al. (2019) then used GNN as the encoder and LSTM as the decoder and incorporated grammar information. With the emergence of Transformer (Vaswani et al. 2017), more researchers (Wang et al. 2020; Lin et al. 2020;

Huang et al. 2021; Rubin and Berant 2021) applied this technique to SQL generation tasks and considered domain-specific language information to improve performance. Recently, inspired by pre-trained language models, (Scholak et al. 2021) applied the T5 model and achieved state-of-the-art performance.

For the automatic code generation of imperative programming, (Mou et al. 2015) were the first to propose the use of a standard encoder-decoder architecture based on RNN to generate corresponding C++ code snippets from user functional descriptions. Ling et al. (2016) proposed two new corpora for card games code and employed the LSTM to construct the encoder-decoder architecture, which incorporates character-level softmax and pointer networks. Yin and Neubig (2018) proposed TRANX, a model based on ASDL, LSTM, the attention mechanism, and the copy mechanism for Python code generation. Hayati et al. (2018) proposed RECODE, which is based on TRANX and integrates information retrieval methods into the Seq2Seq model. Wei et al. (2019) and Yang et al. (2022b) formalized the code generation task and code summarization task as dual tasks and improved model performance through dual learning. Sun et al. (2019) designed a grammar-based structural convolutional neural network for code generation that generated a program by predicting the programming language's grammar rules and subsequently proposed a novel tree-based neural architecture TreeGen (Sun et al. 2020) based on Transformer. With the development of pre-trained language models for code-related tasks, (Liguori et al. 2021) and (Yang et al. 2022c) applied CodeBERT to the exploit code generation task. Lu et al. (2021b) proposed CodeGPT, (Ahmad et al. 2021) proposed PLBART, and (Wang et al. 2021b) proposed CodeT5 for improved Java code generation task. Recently, researchers (Wang et al. 2022b; Le et al. 2022) combined pre-trained language models and reinforcement learning to optimize pre-trained language models from the perspective of code compilability.

For the automatic Turducken-style code generation, to our best knowledge, there was only one study (Liang et al. 2021). They used Encoder-Only pre-trained models and Decoder-Only pre-trained models to conduct experiments on the Lyra dataset to demonstrate the potential possibility of Turducken-style code generation.

In contrast to previous studies, our goal is to generate syntactically correct Turducken-style code, and we consider the learning paradigm of multi-task learning to construct ingenious auxiliary tasks. Moreover, we propose a syntax-first decoding algorithm to satisfy the syntax of the generated code.

8.2 Multi-task Learning for Code Intelligence

Multi-task learning (MTL) is a machine learning technique that allows a model to learn multiple tasks simultaneously by sharing common features or representations. In the field of code intelligence, multi-task learning has been applied to various tasks, such as code understanding, code summarization, and code completion. Wang et al. (2021a) proposed a multi-task learning approach MulCode for source code understanding. This approach learns unified representation space for tasks, with the pre-trained BERT model for the token sequence and the Tree-LSTM model for abstract syntax trees. MulCode achieves promising performance on three downstream tasks: comment classification, author attribution, and duplicate function detection. For the code summarization task, (Xie et al. 2021) proposed a deliberation multi-Task learning approach DMACOS by exploiting method names to improve code summarization. They introduced the tasks of generation and informativeness prediction of method names as two auxiliary training objectives for code summarization. Then they incorporated a novel two-pass deliberation mechanism into their MTL architecture. For the code completion task, (Liu et al. 2020a) and (Liu et al. 2022) adopted multi-task learning to predict the token and its type jointly and utilize the predicted type to assist the token prediction. In summary, multi-task learning can be used to improve the performance of code intelligence tasks by allowing models to learn multiple tasks and share common features or representations.

Compared with previous multi-task learning studies that were used for code intelligence, our approach designs a syntax-guided auxiliary task using for code-based generation and uses a GLU network to incorporate the syntax knowledge learned from the auxiliary task into the primary task. Empirical results show the promising performance of our proposed approach.

8.3 Abstract Syntax Tree Traversal Methods

An Abstract Syntax Tree (AST) is a tree representation of the abstract structure of source code written in a programming language. Researchers usually use AST to extract syntactic knowledge of the code. In the field of code comment generation, (Yang et al. 2021a) proposed Sim_SBT, which uses pre-order traversal to flatten the type nodes of the AST. Hu et al. (2020) and Niu et al. (2022) proposed SBT and X-SBT, respectively. They also used pre-order traversal, along with brackets and xml-like tags to form structured data. In the field of code translation, (Liu et al. 2023a) captured the structural information of the AST by modeling the node distances and the node paths.

Different from previous AST traversal methods, our proposed SAT traversal method takes into account both the type and value of nodes in the AST. By incorporating the type information of the parent node into the code tokens explicitly, using our proposed SAT traversal method can help to generate code, which satisfy syntactic constraints.

9 Conclusion and Future Work

In this study, we propose a novel approach TurduckenGen for Turducken-style code generation, where TurduckenGen addresses the challenge of syntactic constraints on code generation from three perspectives. The results of the automated evaluation show that our proposed TurduckenGen outperforms the state-of-the-art baselines on Lyra and Pisces. Moreover, we also verify our SF-Beam Search decoding method and model components rationality of TurduckenGen by designing a set of ablation studies. Finally, We conduct a human study to evaluate the quality of the generated code in terms of code readability and semantic similarity from the practitioner's perspective.

In the future, we first want to further improve the performance of TurduckenGen by considering advanced code representation methods. We second want to explore more types of Turducken-style code generation tasks to promote the practical application of our study, such as generating programs with regular expressions embedded in Java or Python and JavaScript embedded in HTML.

Acknowledgements This work is supported by the National Natural Science Foundation of China (No. 62372232), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, the Open Project of Key Laboratory of Safety-Critical Software for Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology (No. NJ2020022), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. KYCX23_0396). T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), Birkbeck BEI School Project (EFFECT), and National Natural Science Foundation of China (No. 62272397).

Data Availibility Statement The datasets generated during and analyzed during the current study are available in the Github repository (https://github.com/NTDXYG/TurduckenGen).

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

References

Ahmad W, Chakraborty S, Ray B, Chang KW (2021) Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp 2655–2668

Allamanis M, Sutton C (2013) Why, when, and what: analyzing stack overflow questions by topic, type, and code. In: 2013 10th Working conference on mining software repositories (MSR). IEEE, pp 53–56

Bailey MW (2009) Workshop on declarative aspects of multicore programming (damp 2009) damp 2009

Bogin B, Berant J, Gardner M (2019) Representing schema structure with graph neural networks for text-to-sql parsing. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp 4560–4565

Chakraborty S, Ahmed T, Ding Y, Devanbu PT, Ray B (2022) Natgen: generative pre-training by "naturalizing" source code. In: Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering. pp 18–30

Dahl DA, Bates M, Brown MK, Fisher WM, Hunicke-Smith K, Pallett DS, Pao C, Rudnicky A, Shriberg E (1994) Expanding the scope of the atis task: The atis-3 corpus. In: Human language technology: proceedings of a workshop held at Plainsboro, New Jersey, March 8-11, 1994

Dauphin YN, Fan A, Auli M, Grangier D (2017) Language modeling with gated convolutional networks. In: International conference on machine learning. PMLR, pp 933–941

Dong Y, Jiang X, Liu Y, Li G, Jin Z (2022) Codepad: Sequence-based code generation with pushdown automaton. https://doi.org/10.48550/ARXIV.2211.00818. arXiv:2211.00818

Eghbali A, Pradel M (2022) Crystalbleu: precisely and efficiently measuring the similarity of code. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. pp 341–342

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020:1536–1547

Fernandes S, Bernardino J (2015) What is bigquery? In: Proceedings of the 19th International Database Engineering & Applications Symposium. pp 202–203

Gao T, Fisch A, Chen D (2021) Making pre-trained language models better few-shot learners. In: Joint conference of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing, ACL-IJCNLP 2021, Association for Computational Linguistics (ACL). pp 3816–3830

Gifford DK, Lucassen JM (1986) Integrating functional and imperative programming. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp 28–38

Gu Y, Han X, Liu Z, Huang M (2022) Ppt: Pre-trained prompt tuning for few-shot learning. In: Proceedings of the 60th annual meeting of the association for computational linguistics (Volume 1: Long Papers). pp 8410–8423

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S et al (2021) Graphcodebert: Pre-training code representations with data flow. In: ICLR

Guo D, Lu S, Duan N, Wang Y, Zhou M, Yin J (2022) Unixcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th annual meeting of the association for computational linguistics (Volume 1: Long Papers). pp 7212–7225

Hayati SA, Olivier R, Avvaru P, Yin P, Tomasic A, Neubig G (2018) Retrieval-based neural code generation. In: Proceedings of the 2018 conference on empirical methods in natural language processing. pp 925–930

Hu X, Li G, Xia X, Lo D, Jin Z (2020) Deep code comment generation with hybrid lexical and syntactical information. Empir Softw Eng 25(3):2179–2217

Hu X, Gao Z, Xia X, Lo D, Yang X (2021) Automating user notice generation for smart contract functions. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE). pp 5–17. https://doi.org/10.1109/ASE51524.2021.9678552

Hu X, Xia X, Lo D, Wan Z, Chen Q, Zimmermann T (2022) Practitioners' expectations on automated code comment generation. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022. ACM, Pittsburgh, PA, USA, May 25-27, 2022, pp 1693–1705. https://doi.org/10.1145/3510003.3510152

Huang J, Wang Y, Wang Y, Dong Y, Xiao Y (2021) Relation aware semi-autoregressive semantic parsing for nl2sql. arXiv:2108.00804

Huang J, Wang C, Zhang J, Yan C, Cui H, Inala JP, Clement C, Duan N, Gao J (2022) Execution-based evaluation for data science code generation models. arXiv:2211.09374

Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: Evaluating the state of semantic code search. arXiv:1909.09436

Hussain Y, Huang Z, Zhou Y, Wang S (2020) Codegru: Context-aware deep learning with gated recurrent unit for source code modeling. Inf Softw Technol 125:106309

Hussain Y, Huang Z, Zhou Y, Wang S (2020) Deep transfer learning for source code modeling. Int J Softw Eng Knowl Eng 30(05):649–668

Hussain Y, Huang Z, Zhou Y (2021) Improving source code suggestion with code embedding and enhanced convolutional long short-term memory. IET Softw 15(3):199–213

Iyer S, Konstas I, Cheung A, Krishnamurthy J, Zettlemoyer L (2017) Learning a neural semantic parser from user feedback. In: Proceedings of the 55th annual meeting of the association for computational linguistics (Volume 1: Long Papers). pp 963–973

Klein G, Kim Y, Deng Y, Senellart J, Rush AM (2017) Opennmt: Open-source toolkit for neural machine translation. In: Proceedings of ACL 2017, System Demonstrations. pp 67–72

Le H, Wang Y, Gotmare AD, Savarese S, Hoi SC (2022) Coderl: Mastering code generation through pretrained models and deep reinforcement learning. arXiv:2207.01780

Legendre P (2005) Species associations: the kendall coefficient of concordance revisited. J Agric Biol Environ Stat 10(2):226–245

Li XL, Liang P (2021) Prefix-tuning: Optimizing continuous prompts for generation. In: Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (Volume 1: Long Papers). pp 4582–4597

Liang Q, Sun Z, Zhu Q, Zhang W, Yu L, Xiong Y, Zhang L (2021) Lyra: A benchmark for turducken-style code generation. arXiv:2108.12144

Liguori P, Al-Hossami E, Orbinato V, Natella R, Shaikh S, Cotroneo D, Cukic B (2021) Evil: exploiting software via natural language. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp 321–332

Lin XV, Socher R, Xiong C (2020) Bridging textual and tabular data for cross-domain text-to-sql semantic parsing. Findings of the Association for Computational Linguistics: EMNLP 2020:4870–4888

Ling W, Blunsom P, Grefenstette E, Hermann KM, Kočiskỳ T, Wang F, Senior A (2016) Latent predictor networks for code generation. In: Proceedings of the 54th annual meeting of the association for computational linguistics (Volume 1: Long Papers). pp 599–609

Liu F, Li G, Zhao Y, Jin Z (2020a) Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering. pp 473–485

Liu F, Li G, Wei B, Xia X, Fu Z, Jin Z (2022) A unified multi-task learning model for ast-level and token-level code completion. Emp Softw Eng 27(4):1–38

Liu F, Li J, Zhang L (2023a) Syntax and domain aware model for unsupervised program translation. arXiv:2302.03908

Liu P, Yuan W, Fu J, Jiang Z, Hayashi H, Neubig G (2023) Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Comput Surv 55(9):1–35

Liu Q, Chen Y, Chen B, Lou JG, Chen Z, Zhou B, Zhang D (2020b) You impress me: Dialogue generation via mutual persona perception. In: Proceedings of the 58th annual meeting of the association for computational linguistics. pp 1417–1427

Liu Y, Tantithamthavorn C, Liu Y, Li L (2023c) On the reliability and explainability of automated code generation approaches. arXiv:2302.09587

Lloyd JW (1994) Practical advtanages of declarative programming. In: GULP-PRODE (1). pp 18–30

Longpre S, Hou L, Vu T, Webson A, Chung HW, Tay Y, Zhou D, Le QV, Zoph B, Wei J, et al (2023) The flan collection: Designing data and methods for effective instruction tuning. arXiv:2301.13688

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D et al. (2021a) Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement CB, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S

(2021b) Codexglue: A machine learning benchmark dataset for code understanding and generation. In: Vanschoren J, Yeung S (eds) Proceedings of the neural information processing systems track on datasets and benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual

Mahmud T, Hasan KA, Ahmed M, Chak THC (2015) A rule based approach for nlp based query processing. In: 2015 2nd International conference on electrical information and communication technologies (EICT). IEEE, pp 78–82

Mou L, Men R, Li G, Zhang L, Jin Z (2015) On end-to-end program generation from user intention by deep neural networks. arXiv:1510.07211

Niu C, Li C, Ng V, Ge J, Huang L, Luo B (2022) Spt-code: sequence-to-sequence pre-training for learning source code representations. In: Proceedings of the 44th international conference on software engineering. pp 2006–2018

Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the association for computational linguistics. pp 311–318

Popescu AM, Etzioni O, Kautz H (2003) Towards a theory of natural language interfaces to databases. In: Proceedings of the 8th international conference on intelligent user interfaces. pp 149–157

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I et al (2019) Language models are unsupervised multitask learners. OpenAI blog 1(8):9

Raffel C, Shazeer N, 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. J Mach Learn Res 21(1):5485–5551

Ren S, Guo D, Lu S, Zhou L, Liu S, Tang D, Sundaresan N, Zhou M, Blanco A, Ma S (2020) Codebleu: a method for automatic evaluation of code synthesis. arXiv:2009.10297

Rubin O, Berant J (2021) Smbop: Semi-autoregressive bottom-up semantic parsing. In: Proceedings of the 5th workshop on structured prediction for NLP (SPNLP 2021). pp 12–21

Sánchez-Cartagena VM, Esplà-Gomis M, Pérez-Ortiz JA, Sánchez-Martínez F (2021) Rethinking data augmentation for low-resource neural machine translation: A multi-task learning approach. In: Proceedings of the 2021 conference on empirical methods in natural language processing. pp 8502–8516

Scholak T, Schucher N, Bahdanau D (2021) Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In: Proceedings of the 2021 conference on empirical methods in natural language processing. pp 9895–9901

Sun Z, Zhu Q, Mou L, Xiong Y, Li G, Zhang L (2019) A grammar-based structural cnn decoder for code generation. Proceedings of the AAAI conference on artificial intelligence 33:7055–7062

Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L (2020) Treegen: A tree-based transformer architecture for code generation. Proc AAAI Conf Art Intell 34:8984–8991

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Adv Neural Inf Process Syst 30

Wang B, Shin R, Liu X, Polozov O, Richardson M (2020) Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In: Proceedings of the 58th annual meeting of the association for computational linguistics. pp 7567–7578

Wang C, Yang Y, Gao C, Peng Y, Zhang H, Lyu MR (2022a) No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering. pp 382–394

Wang D, Yu Y, Li S, Dong W, Wang J, Qing L (2021a) Mulcode: A multi-task learning approach for source code understanding. In: 2021 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 48–59

Wang X, Wang Y, Wan Y, Mi F, Li Y, Zhou P, Liu J, Wu H, Jiang X, Liu Q (2022) Compilable neural code generation with compiler feedback. Findings of the Association for Computational Linguistics: ACL 2022:9–19

Wang Y, Wang W, Joty S, Hoi SC (2021b) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 conference on empirical methods in natural language processing. pp 8696–8708

Wei B, Li G, Xia X, Fu Z, Jin Z (2019) Code generation as a dual task of code summarization. Adv Neural Inf Process Syst 32

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics. Springer, pp 196–202

Wiseman S, Rush AM (2016) Sequence-to-sequence learning as beam-search optimization. In: Proceedings of the 2016 conference on empirical methods in natural language processing. pp 1296–1306

Xie R, Ye W, Sun J, Zhang S (2021) Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In: 2021 IEEE/ACM 29th international conference on program comprehension (ICPC). IEEE, pp 138–148

Xu FF, Vasilescu B, Neubig G (2022) In-ide code generation from natural language: Promise and challenges. ACM Trans Softw Eng Methodol (TOSEM) 31(2):1–47

Xuan K, Wang Y, Wang Y, Wen Z, Dong Y (2021) Sead: End-to-end text-to-sql generation with schema-aware denoising. arXiv:2105.07911

Yang G, Chen X, Cao J, Xu S, Cui Z, Yu C, Liu K (2021a) Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In: 2021 8th International conference on dependable systems and their applications (DSA). IEEE, pp 30–41

Yang G, Zhou Y, Chen X, Yu C (2021b) Fine-grained pseudo-code generation method via code feature extraction and transformer. In: 2021 28th Asia-pacific software engineering conference (APSEC). IEEE, pp 213–222

Yang G, Chen X, Zhou Y, Yu C (2022a) Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning. arXiv:2202.09785

Yang G, Chen X, Zhou Y, Yu C (2022b) Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning. In: IEEE international conference on software analysis, evolution and reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022. IEEE, pp 361–372. https://doi.org/10.1109/SANER53432.2022.00052

Yang G, Zhou Y, Chen X, Zhang X, Han T, Chen T (2022c) Exploitgen: Template-augmented exploit code generation based on codebert. J Syst Softw 111577

Yang G, Zhou Y, Chen X, Zhang X, Han T, Chen T (2023) Exploitgen: Template-augmented exploit code generation based on codebert. J Syst Softw 197:111577

Yin P, Neubig G (2018) Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the 2018 conference on empirical methods in natural language processing: system demonstrations. pp 7–12

Yu T, Li Z, Zhang Z, Zhang R, Radev D (2018a) Typesql: Knowledge-based type-aware neural text-to-sql generation. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers). pp 588–594

Yu T, Zhang R, Yang K, Yasunaga M, Wang D, Li Z, Ma J, Li I, Yao Q, Roman S et al. (2018b) Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In: Proceedings of the 2018 conference on empirical methods in natural language processing. pp 3911–3921

Yu T, Zhang R, Er H, Li S, Xue E, Pang B, Lin XV, Tan YC, Shi T, Li Z et al. (2019a) Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In: Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP). pp 1962–1979

Yu T, Zhang R, Yasunaga M, Tan YC, Lin XV, Li S, Er H, Li I, Pang B, Chen T et al (2019b) Sparc: Cross-domain semantic parsing in context. In: Proceedings of the 57th annual meeting of the association for computational linguistics. pp 4511–4523

Zelle JM, Mooney RJ (1996) Learning to parse database queries using inductive logic programming. In: Proceedings of the national conference on artificial intelligence. pp 1050–1055

Zhong V, Xiong C, Socher R (2017) Seq2sql: Generating structured queries from natural language using reinforcement learning. arXiv:1709.00103

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Guang Yang[1] · Yu Zhou[1] · Xiang Chen[2] · Xiangyu Zhang[1] · Yiran Xu[1] · Tingting Han[3] · Taolue Chen[3]

Guang Yang
novelyg@outlook.com

Xiang Chen
xchencs@ntu.edu.cn

Xiangyu Zhang
zhangx1angyu@nuaa.edu.cn

Yiran Xu
behumble@nuaa.edu.cn

Tingting Han
t.han@bbk.ac.uk

Taolue Chen
t.chen@bbk.ac.uk

[1] The College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

[2] The School of Information Science and Technology, Nantong University, Nantong, China

[3] Department of Computer Science, Birkbeck, University of London, London, UK