Non-Autoregressive Line-Level Code Completion

FANG LIU, School of Computer Science and Engineering, State Key Laboratory of Complex & Critical Software Environment, Beihang University, Beijing, China
ZHIYI FU, School of Computer Science, Peking University, Beijing, China
GE LI, School of Computer Science, Peking University, Beijing, China
ZHI JIN, School of Computer Science, Peking University, Beijing, China
HUI LIU, Beijing Institute of Technology, Beijing, China
YIYANG HAO, Silicon Heart Tech Co., Beijing, China
LI ZHANG, School of Computer Science and Engineering, State Key Laboratory of Complex & Critical Software Environment, Beihang University, Beijing, China

Software developers frequently use code completion tools to accelerate software development by suggesting the following code elements. Researchers usually employ AutoRegressive (AR) decoders to complete code sequences in a left-to-right, token-by-token fashion. To improve the accuracy and efficiency of code completion, we argue that tokens within a code statement have the potential to be predicted concurrently. In this article, we first conduct an empirical study to analyze the dependency among the target tokens in line-level code completion. The results suggest that it is potentially practical to generate all statement tokens in parallel. To this end, we introduce SANAR, a simple and effective syntax-aware non-autoregressive model for line-level code completion. To further improve the quality of the generated code, we propose an adaptive and syntax-aware sampling strategy to boost the model's performance. The experimental results obtained from two widely used datasets indicate that our model outperforms state-of-the-art code completion approaches of similar model size by a considerable margin, and is faster than these models with up to 9× speed-up. Moreover, the extensive results additionally demonstrate that the enhancements achieved by SANAR become even more pronounced with larger model sizes, highlighting their significance.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**;

Additional Key Words and Phrases: Code completion, neural networks, non-autoregressive generation

ACM Reference Format:
Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. 2024. Non-Autoregressive Line-Level Code Completion. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 120 (June 2024), 34 pages. https://doi.org/10.1145/3649594

This work is supported by the National Natural Science Foundation of China Grants Nos. 62302021, 62177003, and 62192731, and the Self-determined Research Funds of State Key Laboratory of Complex & Critical Software Environment SKLSDE-2023ZX-15.

Authors' addresses: F. Liu and L. Zhang (Corresponding author), School of Computer Science and Engineering, State Key Laboratory of Complex & Critical Software Environment, Beihang University, 37 Xueyuan Road, Haidian District, Beijing, 100191, China; e-mails: fangliu@buaa.edu.cn, lily@buaa.edu.cn; Z. Fu, G. Li, and Z. Jin, Key Laboratory of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, 100871, China; e-mails: fuzhiyi1129@gmail.com, lige@pku.edu.cn, zhijin@pku.edu.cn; H. Liu, School of Computer Science and Technology, Beijing Institute of Technology, No. 5 Zhongguancun South Street, Haidian District, Beijing, 100081, China; e-mail: liuhui08@bit.edu.cn; Y. Hao, Silicon Heart Tech Co., No. 113, Zhichun Road, Haidian District, Beijing, 100086, China; e-mail: haoyiyang@nnthink.com.

ACM 1049-331X/2024/06-ART120
https://doi.org/10.1145/3649594

1 INTRODUCTION

Code completion is one of the most useful features in **Integrated Development Environments (IDEs)**, improving the software development efficiency by suggesting future code snippets. In recent years, with the development of deep learning technologies and easy-to-acquire open-source codebases, researchers have started to tackle code completion by learning from large-scale code corpora. Various **Language Models (LM)** have been employed for code completion, including N-gram [20, 46], RNN [22, 26], and Transformer based models [13, 27, 51] to predict a single token at a time.

More recently, both academia and industry began to explore the probability of completing code sequences instead of a single next token [1, 3, 6, 53]. However, when a longer code snippet is recommended, the possibility of containing errors increases. Developers have to pay extra effort to read code, locate errors, and revise them, which defends the purpose of improving development efficiency. For this reason, statement/line-level completion is most widely accepted by developers in recent years [5]. Wang et al. [49] and Svyatkovskiy et al. [43] proposed line-level code completion approaches based on language models. Given a partially completed code snippet, they built a Transformer-based LM to complete the entire line of code. Several popular industry code completion tools, e.g., Tabnine [45], aiXcoder [1], and GitHub Copilot [6], also rely on LMs to provide suggestions of code statements in a token-by-token manner. Although such code completion models/tools can suggest longer code snippets, they intrinsically suffer from inefficiency because of the autoregressive decoding process.

For the paired data (X, Y), where $X = \{x_1, x_2, \ldots, x_m\}$ is the source input and $Y = \{y_1, y_2, \ldots, y_n\}$ is the target sequence, the training objective of an **Auto-Regressive (AR)** model is to minimize the following loss:

$$\mathcal{L}_{AR} = \sum_{t=1}^{n} \log p(y_t | y_{<t}, X; \theta) \tag{1}$$

where θ is the parameters of the model, y_t is the generated token at the current time step t, and $y_{<t}$ are the tokens generated in previous $t-1$ decoding steps. These conditional probabilities are parameterized using a neural network, which predicts the next token conditioned on all previously generated tokens. In other words, the training of AR models adopts auto-regressive factorization in a left-to-right manner. Consequently, the quality of the generated results is guaranteed with the help of contextual dependencies, and the generation is not parallelizable and thus particularly slow. Some of these tools received negative feedback about the completion latency [48]. However, the efficiency of the generation process is of great importance for code completion considering the developers' expectation to get instant responses on their own devices. Besides, the autoregressive decoding process can also result in error accumulation at test time [37], leading to poorer results with a longer target code sequence to generate.

We argue that tokens within a statement have the potential to be predicted concurrently, although existing code completion algorithms predict them one by one (from left to right) by exploiting all tokens predicted previously. For example, multiple arguments of the same method are independent, and thus they could be predicted independently and concurrently. In Python code, you can even change the order of the arguments if a function with the "keyword arguments" is called. For example: `Func(arg1=a, arg2=b)` is equivalent to `Func(arg2=b, arg1=a)`. We also argue that even if there exists strong dependency among tokens within a statement, the dependency is not necessarily left-to-right. A typical example is "one line IF-ELSE statement" in Python, formed as:

```
value_1 if condition else value_2
```

In this case, `value_1` is dependent on the following condition (on its right-hand side), instead of the verse. It is more reasonable to consider both the condition and the value concurrently when generating such code statements, where the left-to-right generation manner cannot fit in well. We also perform an in-depth analysis in Section 2 to verify our assumption. The analysis results suggest that left-to-right is not always the optimal order for code completion. The quantitative dependency among the generated code tokens in code completion is smaller than that in **Neural Machine Translation (NMT)** tasks. Based on those observations, we conclude that generating tokens in parallel for code completion is possible and reasonable.

Non-autoregressive (NAR) models [15] have been successfully applied to NMT to generate tokens non-autoregressively through a parallel decoding process. NAR models use the conditional independent factorization for prediction, and the objective is to minimize the loss:

$$\mathcal{L}_{NAR} = \sum_{t=1}^{n} \log p(y_t|X;\theta) \tag{2}$$

where n is the length of the target sequence. It is worth noting that the target position index is omitted in the equation, which is also an input and is added to the embeddings of the decoder input. The position information can help the model to predict different tokens at different positions. Compared with AR models, it is obvious that the conditional tokens $y_{<t}$ are removed for NAR models. Thus, they can perform parallel translations without auto-regressive dependencies, allowing an order of magnitude faster speed during inference. There are two kinds of NAR models: Fully-NAR models [15, 31, 36] and Iterative-NAR models [12, 23]. Fully-NAR models can generate output sequences in parallel with one decoding pass. To improve the accuracy of the Fully-NAR models, Iterative-NAR models are proposed, which sacrifice the speed-up by incorporating an iterative refinement process [12, 23]. Although existing NAR models can speed up the inference process of NMT significantly compared with autoregressive models, NAR models are hard to train without specially designed training strategies as the left-to-right inductive bias among target tokens is ignored during decoding.

Considering the feasibility of parallel generation of code and inspired by the success of non-autoregressive generation in NLP, in this article, we propose a **Syntax-Aware Non-AutoRegressive** model (**SANAR**) for line-level code completion, which achieves parallel code generation with only a single decoding pass. Specifically, we propose an adaptive and syntax-aware sampling strategy to boost the training process of SANAR and improve the accuracy of the code completion, which dynamically glances some code tokens from the target sequence according to their difficulties and token types. The better the model is trained, the fewer code snippets would be glanced. Once the model is well-trained, it can generate the whole line of tokens in a single pass. The experimental results show that SANAR outperforms existing state-of-the-art code completion approaches of similar model size on both Python and Java datasets and is significantly faster than these autoregressive baseline models, achieving 5 $\sim$ 6$\times$ speed-up on average and 9$\times$ for long targets completion.

To summarize, the major contributions of our work are as follows:

— An empirical study on the dependency among the generated code tokens in code completion, whose results suggest that it is potentially practical to predict code tokens in parallel.
— A novel approach for line-level code completion. To the best of our knowledge, it is the first non-autoregressive approach to code completion. We boost the approach with an adaptive and syntax-aware sampling strategy that is specially designed for source code.
— A large-scale evaluation of the proposed approach whose results suggest that our approach outperforms state-of-the-art code completion approaches of similar model size and significantly reduces the inference time.

2 EMPIRICAL STUDY

We argue that the dependency among the generated code tokens is not as strong as in NMT (where NAR has been successfully applied), and the left-to-right generation order is not always optimal for code completion. To verify those assumptions, in this section, we conducted an empirical study to analyze the dependency among target tokens in code completion and answer the following questions:

> *RQ1: Is the left-to-right generation process always optimal for code completion?*
> *RQ2: How much do the generated tokens depend on each other?*

The answers to these questions provide the empirical foundation for whether the parallel generation of code is feasible and reasonable. We perform experiments on two high-quality benchmark datasets (Python [38] and Java [2]).[1] Prior to conducting the dependency analysis, we conducted a preliminary experiment to evaluate different model architectures. This experiment was crucial in determining the appropriate architecture to be used in SANAR, as well as guiding the dependency analysis experiments.

2.1 Model Architecture Design Choice

There are two alternative architectures for performing full-line code completion: the encoder-decoder architecture and the decoder-only architecture. These two architectures differ in the following ways:

(1) **Training process**: The training process of the decoder-only model is the same as language models, that is, the model is trained to predict probability distributions of the next token given the previous context. Every token in the training programs is used to train the model in a supervised manner. While the encoder-decoder model is trained to predict the target code sequence given the source sequence, the cross entropy loss will only be built upon the target tokens. In other words, the decoder-only model has a denser supervision signal.
(2) **Cross-attention**: Except for the self-attention, the encoder-decoder model also contains a cross-attention, which introduces information from the input sequence to the decoding layers.

In this section, we conducted a preliminary experiment to investigate the suitability of different architectures for line-level code completion and their performance with varying target lengths. The results are shown in Table 1. **ES** and **EM** represent the **Edit Similarity score** and **Exact Match accuracy**, respectively. A comprehensive explanation of these metrics can be found in Section 4.2. The results indicate that the decoder-only architecture performs better when completing shorter code sequences with a maximum length of 10 in all the metrics. However, as the target length increases, the encoder-decoder model outperforms the decoder-only model significantly. Moreover, the EM scores for both architectures show a significant decrease as the length of the sequences increases, clearly indicating the challenge in accurately completing longer sequences. Completing longer code sequences is more challenging as it requires a full understanding of the semantic information in the contextual code sequence to accurately predict the target token. With an explicit cross-attention mechanism, the encoder-decoder model can effectively model the dependency between the contextual tokens and the target tokens. On the other hand, completing shorter code sequences, which are comparatively easier, benefits from the decoder-only model as

[1]The datasets are also used for evaluating our model. The detailed information of the datasets is presented in Section 4.

Table 1. Results of Different Model Architectures

Target Length	Model	Python			Java		
		BLEU	EM	ES	BLEU	EM	ES
$L \leq 10$	En-De	19.89	17.14	62.42	22.27	29.01	63.34
	Decoder	29.39	18.89	66.30	25.39	31.82	69.86
$L > 10$	En-De	36.80	6.90	68.28	45.89	9.54	73.02
	Decoder	28.61	6.41	59.40	36.63	8.20	65.46

En-De stands for encoder-decoder architecture.

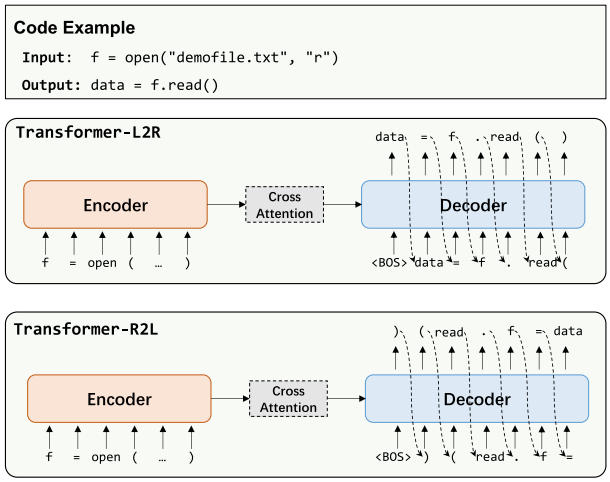

Fig. 1. Code completion with reverse generation orders.

it is trained more extensively with a larger number of training data points (every token in the training programs is used).

Based on the results and observations, we ultimately decided to employ the encoder-decoder architecture for both the empirical study and SANAR. Hopefully, this choice will not only improve efficiency but also enhance the quality of code completion, particularly for longer code sequences.

2.2 Reversing the Order of Code Generation

To answer the first question, we conduct an experiment to analyze the impact of different generation orders. Following existing work [43, 49], we employ autoregressive Transformer architecture to perform line-level code completion using two reverse orders: left-to-right (Transformer-L2R) and right-to-left (Transformer-R2L). The model architecture and configurations remain the same as in Transformer-base [47] and our approach (Section 4). Specifically, we utilize a 6-layer encoder and a 6-layer decoder with a model size of 512 and an intermediate size of 2,048, along with 64 attention heads per block. The model is trained from scratch on our data.

As shown in Figure 1, these two models have the same architecture, and the only difference is the generation order of the target code sequence, where Transformer-L2R conducts completion in a left-to-right manner, and Transformer-R2L conducts completion in a right-to-left manner. Transformer-L2R predicts code sequences from left to right, capturing the left-to-right dependency in the target code snippets. Oppositely, Transformer-R2L captures the right-to-left dependency in

Table 2. Results of Different Generation Orders

Dataset	Model	BLEU	EM	ES
Python	Transformer-L2R	21.93	16.24	62.73
	Transformer-R2L	24.52	16.07	62.82
Java	Transformer-L2R	25.73	30.33	63.95
	Transformer-R2L	26.09	30.01	63.57

EM stands for exact match accuracy, BLEU is the BLEU-4 score, and ES stands for edit distance similarity score.

Table 3. The Percentage of Generating High-quality Code

Dataset	Model	Metrics	Percentage
Python	only Transformer-L2R	EM	2.65%
		ES>50	6.57%
	only Transformer-R2L	EM	2.47%
		ES>50	7.21%
	Both	EM	13.59%
		ES>50	60.39%
Java	only Transformer-L2R	EM	3.28%
		ES>50	6.71%
	only Transformer-R2L	EM	3.86%
		ES>50	8.22%
	Both	EM	26.15%
		ES>50	60.43%

the target. From an empirical standpoint, we argue that the left-to-right order is not always the optimal coding order for programmers, and therefore may not be the optimal decoding order for the decoder. Different generation orders result in distinct conditional contextual information, thereby affecting the difficulty of correctly predicting subsequent tokens. With this in mind, this experiment aims to determine if there are differences in difficulty when learning the dependency in different directions, by comparing the performance of various generation orders.

We use Python [38] and Java [2] benchmark datasets to evaluate their performance, where the model is employed to predict the next line of code given previous (up to) 10 lines. We adopt the BLEU-4 score, Exact Match accuracy (EM), and Edit Similarity score (ES) as metrics to evaluate the quality of the generated code. Table 2 shows the completion results of each model. As seen from the results, the overall performances of completing code with different generation orders are comparable. Surprisingly, the right-to-left order outperforms the left-to-right order in terms of BLEU-4 score. The results suggest that the standard left-to-right generation process is not always optimal for code completion. Other generation orders or manners are also feasible.

Furthermore, we present the percentage of target code lines that can only be correctly predicted by each model in Table 3. For Python language, 2.65% and 2.47% of programs can only be correctly generated by L2R and R2L, respectively; 6.57% and 7.21% of the programs can only be approximately generated (edit similarity >50%) by L2R and R2L, respectively. Also, there are more than 60% target code lines that can be approximately generated by both L2R and R2L. The results on the Java dataset are similar. These results further confirm that the standard left-to-right generation process is not optimal for completing correct code lines. In this article, we take the first step to explore completing code in parallel, i.e., generating code tokens non-autoregressively.

2.3 Quantitative Dependency among Code Tokens

To answer the second question, following Ren et al. [39], we build a **Dependency Analysis Model (DAM)** to measure the quantitative dependency among the generated tokens by attention density ratio and compare the measured dependency among code tokens against that among natural language tokens. Considering that NAR models have been successfully applied to NMT tasks, we select NMT for comparison.

To measure the dependency among target tokens, and compare it with the dependency on source tokens, we have the following considerations in the design of DAM: (1) Predicting the current masked target token based on bi-directional target context and source tokens; (2) Ensure the dependency on source and target tokens to be comparable. However, neither the encoder nor the decoder of the encoder-decoder architecture can fulfill this target:

— Encoder: cannot attend to target tokens.
— Decoder: although cross-attention and self-attention are available (attend to both source tokens and previous target tokens), they cannot attend to the right context of the target sequence, and also are hard to ensure the dependency on source and target tokens to be comparable.

For this reason, we build a variant of the Transformer encoder following Ren et al. [39], which applies mix-attention to calculate the attention weights on both source and target tokens in a single softmax function. Specifically, we build a Transformer encoder model, which takes the whole source sequence and the partially masked target sequence as its input. Then we make it learn to predict the masked target tokens. DAM utilizes a mix-attention [18] mechanism, where source tokens can only attend to source tokens while target tokens can attend to all source and target tokens. By learning to predict masked target tokens based on source context and target context, DAM learns to allocate different ratios of attention weights to source tokens and target tokens in the mix-attention. After convergence, we measure the dependency among target tokens using the trained DAM. This is done by calculating the ratio of attention density α_i in the target context to that in the full context when predicting a specific target token y_i. It is defined as follows:

$$\alpha_i = \frac{\frac{1}{N}\sum_{j=1}^{N} A_{i,j}}{\frac{1}{N}\sum_{j=1}^{N} A_{i,j} + \frac{1}{M}\sum_{j=N+1}^{N+M} A_{i,j}} \tag{3}$$

where $A_{i,j}$ denotes the attention weights from token i to token j in mix-attention, $j \in [1, N]$ represents the target token, and $j \in [N+1, N+M]$ represents the source token. M and N are the lengths of the source and target input, respectively. $\sum_{j=1}^{N+M} A_{i,j} = 1$. It is worth noting that the attention score of the multi-heads is aggregated through averaging. According to existing study [8, 50, 54], Transformer has been loosely shown to encode local syntax (e.g., attend more to adjacent tokens) in the lower layers, and more information about semantic knowledge and task-specific knowledge in the higher layers. In this experiment, we aim to analyze the token dependency from a semantical perspective, instead of just focusing on shallow information like the lexical relationship, thus we decided to compute the attention from the last layer.

Following Ren et al. [39], we modified the attention mask in the Transformer model to change the attention scope and applied the mask on all the attention layers and heads. DAM is trained to predict the masked tokens and counts α_i for those masked tokens. Different masking ratios p might give different statistical results. For a given masking probability p, the final attention density ratio $R(p)$ is calculated by averaging α_i on all test data:

$$R(p) = \text{avg}\left(\frac{1}{|\mathcal{M}^p|}\sum_{i \in \mathcal{M}^p} \alpha_i\right) \tag{4}$$

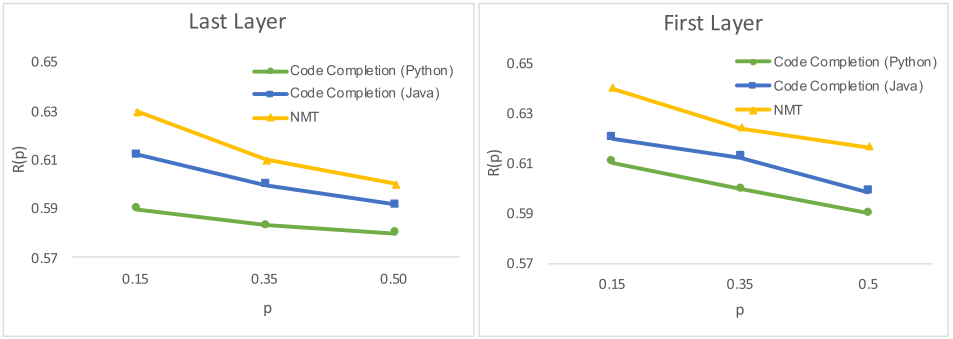

Fig. 2. The attention density ratio R(P) under different masking probability p in code completion and NMT.

where $\mathcal{M}^p$ indicates the masked token set under masking probability p. Since α_i denotes the attention density ratio on the target context when predicting the target token i, a higher value of α_i indicates that the prediction of token i places more emphasis on the target context, implying a stronger dependency among the target context elements. The function $R(p)$ represents the average attention density ratio α_i across all the predicted tokens in the test set, with a masked probability of p. A larger value of $R(p)$ signifies that, on average, there is a greater dependency on the target context when predicting the target tokens across the entire test dataset. **Thus, a bigger attention density ratio $R(p)$ indicates a larger dependency among target tokens**. In the extreme case of $p = 1$, DAM reads an all-masked target sequence as input and makes predictions based only on the source sequence, resembling the Fully-NAR model.

We employ DAM to conduct experiments on Python [38] and Java [2] datasets for the line-level code completion task. Additionally, we apply DAM to the IWSLT 2014 **German-English (De-En)** translation dataset[2] for the NMT task. Since a larger p naturally gives a lower $R(p)$ because of the limited target context, which further diminishes the difference of $R(p)$ between tasks, we make statistics on relative low masking probabilities {0.15, 0.3, 0.5}.

The results are shown in Figure 2. The subfigures on the left and right sides correspond to the results obtained from the last and first layers, respectively. For the last layer, we found that the attention density ratio for NMT is bigger than code generation for all masking probability p, which demonstrates the dependency among the target tokens in code completion is weaker than NMT. For the code completion task, we have observed a notable distinction in the attention density ratio between Java and Python. Specifically, the attention density ratio for Java is higher compared to Python. This finding implies that the dependency among the target tokens in Python is comparatively weaker than that in Java. We suspect that this disparity could be attributed to Python being a dynamic language, where the overall relevance of tokens in Python code tends to be lower compared to Java. As a result, the interdependence between tokens may be less pronounced in Python, leading to a lower attention density ratio in the code completion task. Also, the $R(p)$ on code completion is closer to 0.5, which means that DAM pays more balanced attention to both source and target sides compared with NMT.

Regarding the results of the first layer, we observe that the overall attention density ratio scores computed from the first layer are consistently higher than those from the last layer across all three tasks. This suggests that the inter-dependency among target tokens is greater in the lower layers, as the attention weights tend to give more importance to the adjacent tokens. Furthermore, we

[2]https://wit3.fbk.eu/mt.php?release=2014-01

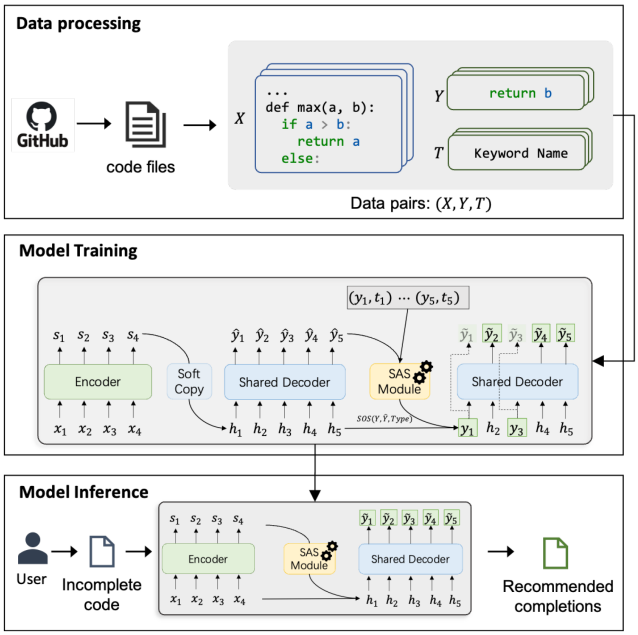

Fig. 3. The workflow of our model.

observed a consistent order of scores for the three tasks in both the first and last layers. These findings reinforce our previous results and provide additional evidence of the distribution of attention throughout the model.

Based on the above results, we argue that it is possible to predict code tokens in parallel. To this end, we propose SANAR, a Non-autoregressive model for statement-level code completion.

3 SANAR

In this section, we present our model SANAR in detail. The workflow of our approach is summarized in Figure 3.

We first employed a sliding context window of 10 lines to create the training data pairs, and then use these data to train our model following a two-state decoding strategy. After the model is well-trained, given an incomplete code, our model can generate the statement in one parallel decoding process. We first introduce our backbone model, we name it as Parallel Transformer, which contains a non-autoregressive decoder that can predict target sequence in parallel. Next, we introduce our proposed adaptive syntax-aware sampling strategy, which allows SANAR to generate syntax-accurate code sequences in a single pass during inference. Finally, we introduce the training and inference procedure of SANAR.

3.1 Parallel Transformer

We build SANAR to perform line-level code completion in parallel, which uses conditional independent factorization for the target tokens. Formally, given the contextual code sequence $X = \{x_1, x_2, \ldots, x_m\}$, the non-autoregressive full-line code completion task is to predict a sequence of tokens $Y = \{y_1, y_2, \ldots, y_n\}$ that form a complete statement in a non-autoregressive way:

$$P(y_1, \ldots, y_n | x_1, x_2, \ldots, x_m) = \prod_{t=1}^{n} P(y_t | x_1, x_2, \ldots, x_m) \tag{5}$$

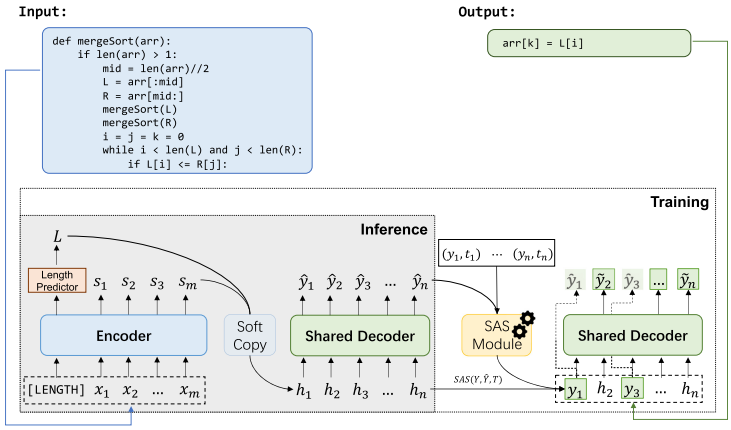

Fig. 4. The architecture of SANAR.

Figure 4 shows the architecture of our model. Specifically, SANAR adopts the encoder-decoder transformer architecture: a context encoder that does self-attention, and a non-autoregressive code-generation decoder that has one set of attention heads over the encoder's output and another set (self-attention) for the generated code, which generates the tokens of the target sequence non-autoregressively through a parallel decoding process.

To achieve this, SANAR performs decoding twice during training with only one shared decoder. As shown in Figure 4, the first decoding is performed in a fully-NAR way, where the input to the decoder $H = \{h_1, h_2, \ldots, h_n\}$ are copied from the encoder output using soft-copy [52], which maps the encoder embeddings $S = \{s_1, s_2, \ldots, s_m\}$ into target length $H = \{h_1, h_2, \ldots, h_n\}$ depending on the distance relationship between the source position i and the target position j. Specifically, the input embedding of the target at position j is computed as a weighted sum of the source embeddings:

$$\begin{aligned} y_i &= \sum_{i=0}^{m} W_{ij} s_i \\ W_{ij} &= W_j W_i (i + s_i) \end{aligned} \tag{6}$$

where W_i, W_j, W_{ij} are the weights. W_{ij} depends on the distance relationship between the source position i and the target position j. All the weights are trained along with the model training process. The initially predicted tokens $\hat{Y}$ are predicted as:

$$\hat{Y} = f_{\text{dec}}(\text{soft-copy}(f_{\text{enc}}(X; \theta')); \theta) \tag{7}$$

The prediction accuracy indicates the difficulty of fitting the current target.

In the second decoding, we propose an adaptive **Syntax-Aware Sampling (SAS)** strategy to dynamically glance code snippets from the target sequence depending on its difficulty and the tokens' syntax types, aiming to incorporate the explicit syntactic information of the program. Here, "glance" means select (sample) part of the tokens from ground truth as the input for the second-pass decoding pass. The selected tokens can provide "correct" information from the ground-truth target, therefore it helps train the decoder to predict the rest non-selected tokens. "Dynamically" means the sampling number N is decided based on the current trained model's capability, which can be measured by the difficulty of correctly generating the target token sequence via the initial decoding pass.

SANAR samples words of the targets as the extra decoding input by SAS sampling according to the first decoding results and the syntax information of the target tokens and learns to predict the rest of the words that are not selected. It is important to note that only the second decoding will update the model parameters. The training objective is to maximize the following loss function:

$$\mathcal{L}_{SAS} = \sum_{y_t \in Y \setminus \mathbb{SAS}(Y, \hat{Y}, T)} \log P(y_t | X, \mathbb{SAS}(Y, \hat{Y}, T); \theta) \tag{8}$$

where $\hat{Y}$ is the initially predicted tokens in the first decoding process via a fully-NAR decoding way, T is the syntax type sequence of the target sequence, and $\mathbb{SAS}(Y, \hat{Y}, T)$ is a subset of tokens selected via the adaptive SAS strategy. Meanwhile, the length of the target sequence is predicted jointly via the length predictor.

3.2 Syntax-Aware Sampling Strategy

Syntax-Aware Sampling (SAS) strategy is an essential component of SANAR, which adaptively selects the positions of tokens from the target sequence depending on its difficulty and the syntax types of the tokens. The selected tokens can provide correct information from the ground-truth target, thus reducing the burden of the decoder to predict the rest non-selected tokens in the training phase. SAS samples more tokens for SANAR to glance at the beginning and then reduces the sampling number gradually, which helps SANAR to learn, eventually, how to predict the whole line code snippet without seeing any ground truth tokens in one pass.

Formally, given the context code sequence X, its predicted token sentence $\hat{Y}$, the ground truth Y, and the token's syntax type sequence of the target T, the goal of SAS strategy $\mathbb{SAS}(Y, \hat{Y}, T)$ is to obtain a subset of tokens sampled from Y. Specifically, there are two steps:

(1) **Deciding the sampling numbers** N depending on the difficulty of correctly generating the target token sequence:

$$N = \lambda \cdot dis(Y, \hat{Y}) \tag{9}$$

N is computed by comparing the difference between $\hat{Y}$ and Y. We adopt the Hamming Distance as the metric following [36], $dis(Y, \hat{Y}) = \sum_{t=1}^{T} (y_t \neq \hat{y_t})$. λ is the sampling ratio, which is a hyper-parameter to flexibly control the number of sampled tokens. More tokens will be selected and fed as input for the second-pass decoding if the network's initial prediction is less accurate, i.e., $dis(Y, \hat{Y})$ is big. Thus, the sampling number can be adaptively decided considering the current trained model's prediction capability and the training sample's complexity. As the model's performance improved during the training, it will adaptively reduce the percentage of sampling, making sure that the well-trained model could learn to generate the final code sequence in one pass.

(2) **Sampling N tokens from the target sequence**. The most direct method is randomly selecting N tokens from Y like in Qian et al. [36]. However, we argue that considering the syntactic information of the code explicitly during token selection will be beneficial for understanding the programs and can help train the decoder to predict the rest tokens more precisely. In programs, keywords, identifiers, and operators contain more symbolic and syntactic information than other tokens (for example, literals and separators in Java, like ';', '{', '}', etc.). Glancing at these tokens will help a lot.

To capture the syntactic information of the code during the sampling procedure, we present a **Hybrid-Syntax-Guided (HSG)** sampling strategy. Specifically, $1 - p$ of the time, the sampling is performed randomly, where the tokens are randomly selected from the target sequence; and

p of the time, tokens are selected depending on their syntax-types,[3] where we randomly select $K \leq N/2$ keywords, $I \leq N/4$ identifiers, and $O \leq N/4$ operators from Y. $K + I + O \leq N$.[4] In order to determine the appropriate thresholds, we performed several preliminary experiments using different sets of thresholds: [N/3, N/3, N/3], [N/2, N/4, N/4], [N/4, N/2, N/4], and [N/4, N/4, N/2]. After assessing the results, we ultimately chose the configuration that produced the most favorable outcomes, specifically N/2 for keywords, N/4 for identifiers, and N/4 for operators. This chosen setting was then applied in our final experiments. The reason for introducing randomness (randomly sampling for $1 - p$ of the time) in the sampling process is to enable SANAR to explore more inter-dependency among target tokens. Compared with the pure random sampling strategy, our HSG sampling strategy can increase the probability of glancing at keywords, operators, and identifiers, which can help the SANAR capture the program's syntactic and semantic information better. We also analyze the performance of different token selection strategies in Section 5.3.5.

Finally, we can obtain a subset of tokens sampled from Y:

$$\begin{aligned} \mathbb{SAS}(Y, \hat{Y}, T) &= HSG(Y, N, p) \\ N &= \lambda \cdot dis(Y, \hat{Y}) \end{aligned} \tag{10}$$

3.3 Training Process

In the typical training procedure of autoregressive decoder, all the ground truth tokens $y = \{y_1, y_2, \ldots, y_n\}$ (instead of using the generated output from the previous time step) are used as the input for the decoder, also known as "teacher forcing". That is, each token of the target sequence $\widetilde{y}_i$ is predicted based on its previous (ground truth) tokens $y_1, \ldots, y_{i-1}$ during the training. However, during inference, when utilizing the autoregressive decoder, the generated tokens from the previous time step $(\widetilde{y}_1, \widetilde{y}_2, \ldots, \widetilde{y}_{i-1})$ are used as the input. SANAR deviates from autoregressive decoding by sampling parts of the ground truth as input, and trains the decoder to learn to predict the remaining tokens in parallel based on these input tokens as shown in Figure 4.

During training, the number of sampling tokens is gradually decreased throughout the training process. This progressive reduction facilitates explicit modeling of word interdependencies, contributing to the decoder's ability to generate whole sequences simultaneously. Consequently, the model becomes more adept at capturing the underlying data structure and dependencies. As training progresses, the sampling strategy gradually decreases the number of sampled words ($N \rightarrow 0$), aiding the model in learning the parallel generation of complete sentences. Ultimately, SANAR becomes proficient at generating entire sentences without relying on any targeted token sampling, i.e., SANAR can generate the whole sequence based solely on the encoder inputs. Thus, during the inference time, the decoder can work effectively when the ground truth is unavailable.

3.4 Inference

As SANAR is well-tuned, it will adaptively reduce the percentage of sampling, making sure that the trained model could learn to generate the whole line of code in one pass. Thus, the inference of SANAR is fully parallel with only a single pass. In traditional left-to-right code completion, where the target sequence is predicted token by token, the length of the sequence can be determined by predicting a special **EOS (end of sentence)** token. However, as all target tokens are generated in parallel in NAT models, there is no such special token or target information to guide the termination of decoding. NAT models must know the target length in advance and then generate the content based on it. Therefore, how to predict the correct length of the target sentence is critical

[3] p is a hyper-parameter to control the probability of syntax-guided sampling, the best setting is $p = 30\%$.
[4] The number of these specific tokens might not be enough.

for NAT models. Many methods for target length prediction have been proposed [12, 14, 15]. In SANAR, we follow Ghazvininejad et al. [12] to add an additional `[LENGTH]` token to the source input, and the encoder output for the `[LENGTH]` token is used to predict the length, formed as a *max-target-length* classification task. We use 1,000 as the max length. The loss is added to the main cross-entropy loss from the target sequence.

4 EVALUATION

For evaluation, we seek to answer the following research questions:

RQ3: Performance Comparison. *How accurate is SANAR compared with the state-of-the-art line-level code completion approaches?*
RQ4: Effectiveness of Syntax-aware Training Strategy. *Is our proposed training strategy necessary to improve the quality of the generated code?*
RQ5: Sensitivity Analysis. *How do various factors affect SANAR's performance, such as generated code length, model architecture, model size, completing start point, sampling strategy, and sampling probability?*
RQ6: Quality Analysis. *How useful is the code generated by SANAR in practice?*

4.1 Datasets

We conduct experiments on two program benchmarks crawled from Github repositories: PY150 [38] contains 150,000 Python files, split into a training set of 100,000 files and test set of 50,000 files, GitHub-Java [2] contains 14,317 projects, we follow Hellendoorn and Devanbu [19] and Karampatsis et al. [22] to split validation and test set, and randomly sample 1,000 projects from the rest of the projects as the training set. We use the Python official library tokenizer[5] and Javalang[6] to split Python and Java programs into lines of tokens, and extract their syntax types. Then we employ a sliding context window of 10-lines to create the data pairs, that is, the context code sequence contains at most 10 lines of code tokens, and the next line is considered as the target code sequence.

As shown in Figure 5, we process each code file by the following steps. Initially, we select the first 10 lines (lines 1∼10) of code tokens as the input, with the subsequent line (line 11) serving as the target code statement. We then proceed to slide the window down, generating the next input (lines 2∼11) and the corresponding target code statement (line 12), and so forth. This process continues until the window reaches the last $N-10 \sim N-1$ lines, with the last line serving as the final target. This allows us to cover the entire current code file. For code files that have fewer than 11 lines, we directly utilize the first $N-1$ (<10) lines as the input, with the last line being considered as the target. Hence, the context code sequence contains a maximum of 10 lines of code tokens, with the subsequent line being the target code sequence. By applying this method to all the files in the dataset, we obtain a comprehensive collection of input-output pairs. The detailed information of the datasets is shown in Table 4.

It is worth noting that SANAR can predict code with arbitrary length given the context, and we focus on line-level code completion in this article. To simplify the problem, we define 10 lines as a source and the next line as a target. Actually, SANAR can be seamlessly applied to other completion situations. For example, adding more source-target combinations (e.g., half-line as target) into the training data or directly applying our training strategy to the decoder-only model when the completion point and context length are uncertain.

[5] https://docs.python.org/3/library/tokenize.html
[6] https://github.com/c2nes/javalang

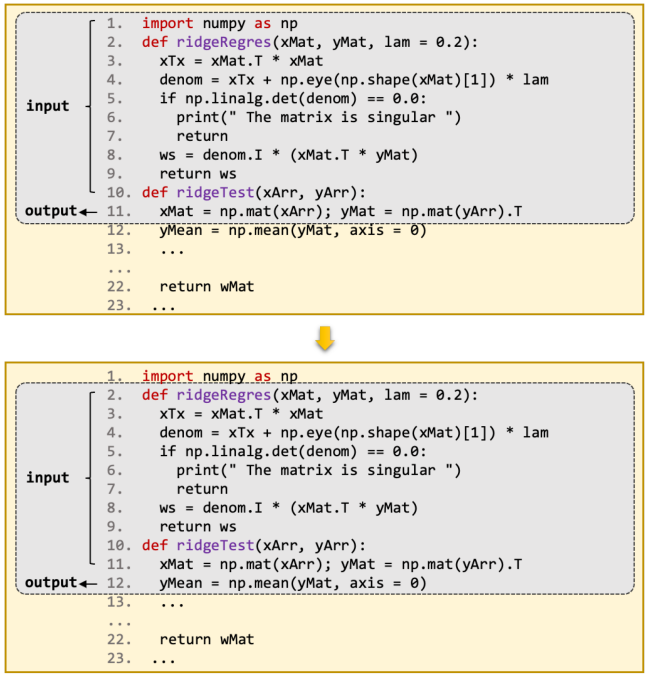

Fig. 5. Illustration of the input-output data preparation process.

Table 4. Dataset Statistics

	Python	Java
# of training pairs	7,531,208	12,993,112
# of testing pairs	3,693,213	671,236
Avg. tokens in cxt	90.8	71.2
Avg. tokens in tgt	9.4	6.9

4.2 Metrics

We use the following metrics to evaluate the performance of our approach:

— BLEU: We use the BLEU-4 [35] score to measure the n-gram similarity between the target code sequence and the generated code sequence. It is computed as:

$$BLEU = BP \cdot \exp\left(\sum_{n-1}^{N} w_n \log p_n\right) \tag{11}$$

where p_n is the precision of n-grams, that is, the ratio of n-grams in the candidate that are also in the reference:

$$p_n = \frac{\#\text{n-grams}_{overlap}}{\#\text{n-grams}_{candidate}} \text{ for n=1, ..., N} \tag{12}$$

where N is the maximum number of grams we consider, and we set N to 4. Each $w_n = \frac{1}{N}$ is the weight of p_n. BP is a brevity penalty that penalties the short candidates.

Table 5. Model Configuration Information of SANAR

Hyperparameter	Value
Encoder layer	6
Decoder layer	6
Model size	512
Intermediate size	2,048
Vocabulary size	50,000
Sampling ratio λ	0.3
Dropout rate	0.1
Optimizer	Adam, $\beta = (0.9, 0.999)$
Training epoch	18
Batch size	16k tokens

— Edit Similarity (ES): We use the character-level edit distance similarity of the predicted output $\hat{Y}$ and the target output Y, which is computed as:

$$ES = 1 - \frac{Lev(\hat{Y}, Y)}{|\hat{Y}| + |Y|} \tag{13}$$

where Lev is Levenshtein distance.
— Exact Match Accuracy (EM): We compare the exact matching accuracy between the generated code sequence and the ground truth.
— Latency: The inference latency is computed as the time to decode a single target code sentence without mini batching (batch size = 1), averaged over the whole test set. We measure the decoding latency in PyTorch on a single NVIDIA Tesla V100 for all models.

4.3 Implementation Details

The detailed model hyper-parameter settings of SANAR are provided in Table 5. For SANAR, we use a 6-layer encoder and a 6-layer decoder with the model size of $d_{\text{model}} = 512$ and intermediate size of $d_{\text{ff}} = 2{,}048$. We set the vocabulary size to 50,000 for both Python and Java datasets. UniXcoder uses 12-layer of Transformer with model size of $d_{\text{model}} = 768$, and the intermediate size $d_{\text{ff}} = 3{,}072$. UniXcoder has a pre-defined vocabulary consisting of 51,416 sub-tokens. Since UniXcoder and SANAR employ different data-processing techniques, to make the comparison fair, we use our vocabulary and data processing rule to replace the number, string, and out-of-vocabulary tokens in the UniXcoder generated code with <NUM>, <STR>, <unk>, respectively. It is worth noting that UniXcoder is a pre-trained model and the released checkpoint has around twice the parameters as our model and other baselines. The sampling ratio λ is set to 0.3. For the hyper-parameter p in the HSG sampling strategy, we use 30% as the final value, which can achieve the best results. The specific syntax types used for guiding the sampling include *keyword*, *operator*, and *identifier*.

The model parameters of SANAR are updated solely during the second decoding process, and the training loss is calculated based on the un-sampled tokens. As both decoding stages predict tokens in parallel and the loss is only calculated for a portion of the target tokens, each training step incurs a smaller cost compared to traditional transformers. Since each step only focuses on fitting a few tokens, it is necessary to cover more training steps (epochs) compared to traditional transformers. Specifically, we trained both the SANAR and baseline models with batches of 16k tokens on 2 V100 GPUs. Training SANAR took 12.6 hours over 18 epochs, while the baseline AR transformer took 10.5 hours over 10 epochs. **Thus, the training time and cost of SANAR and the baseline model are comparable**. The beam size for all AR baselines is set to 5. In SANAR, we

Table 6. Results against State-of-the-art Models on Python Dataset

Model	BLEU	EM	ES	Latency	Speedup
Transformer	21.93	16.24	62.73	121ms	1.0×
AR 12-1	18.99	15.08	61.36	51ms	2.4×
UniXcoder	23.79	**21.11**	61.56	181ms	0.7×
SANAR	**29.07**	18.35	**66.90**	**20ms**	6.1×

Table 7. Results against State-of-the-art Models on Java Dataset

Model	BLEU	EM	ES	Latency	Speedup
Transformer	25.73	30.33	63.95	101ms	1.0×
AR 12-1	22.64	28.38	62.23	49ms	2.1×
UniXcoder	21.59	**33.74**	65.85	161ms	0.6×
SANAR	**32.41**	30.57	**66.98**	**19ms**	5.3×

do not use beam search (beam size is 1) and keep the top 1 generated sample as the final output. We set the dropout rate to 0.1 and use Adam optimizer with $\beta = (0.9, 0.999)$. The replication package is publicly available.[7]

5 EMPIRICAL RESULTS

5.1 Comparison with State-of-the-art Line-level Code Completion Models (RQ3)

To answer this research question, we compare SANAR with the following state-of-the-art line-level code completion baselines, and all of them generate tokens in an autoregressive manner.

- Transformer [43]: We reproduce Intellicode compose [43], an autoregressive Transformer-based (GPT-2) line-level code completion approach using our datasets under comparable parameter settings with SANAR.[8]
- AR 12-1 [24]: Kasai et al. [24] found that deep-shallow (deep encoder, shallow decoder) autoregressive models can outperform non-autoregressive models with comparable inference speed. We also use a Transformer architecture with a 12-layer encoder and a 1-layer autoregressive decoder as our baseline.
- UniXcoder [17]: It is a unified cross-modal pre-trained Transformer architecture, which considers semantic and syntax information from code comments and ASTs, and can support both code understanding and generation tasks. When applied to code completion, UniXcoder works in an autoregressive manner. We fine-tune the released unixcoder-base checkpoint[9] on our Java and Python datasets for comparison.

The main results on Python and Java benchmarks are presented in Tables 6 and 7. When compared with the first two baselines which are trained from scratch (not pre-trained), SANAR outperforms these baselines on all evaluation metrics, especially on BLEU and ES scores. When compared with the powerful pre-trained baseline UniXcoder [17], our model outperforms it in terms of BLEU and ES and performs a little worse on EM. The reason mainly lies in that UniXcoder is pre-trained by several language modeling objectives with a larger corpus of six programming languages. Also, the model size of UniXcoder is much larger compared to SANAR, as described in Section 4.3.

[7] https://github.com/LiuFang816/SANAR
[8] Since their datasets, trained model and source code are not publicly available, and our computing resources are also limited, we trained a GPT-2 model using our dataset under comparable parameter settings with SANAR.
[9] https://huggingface.co/microsoft/unixcoder-base

We can observe that the improvements in BLEU and Edit Similarity are more significant than in Exact Match accuracy. As a recommendation tool, the code completion add-in serves as developers' cooperator, and developers can accept to make a few edits to correct the recommended code. Besides, different code snippets can have identical functionality. For example, these two Python return statements have the same functionality:

```
return True if a==0 else False vs return False if a!=0 else True
```

Only evaluating the quality of the generated code by comparing the exact match with the ground truth is too strict. Thus, BLEU and Edit Similarity, which calculate the token overlapping and the minimum number of operations required to transform the predicted code sequence into the target sequence, can evaluate the generated code more thoroughly. Thus, the significant improvements on BLEU and ES further demonstrate that SANAR can generate a code sequence that meets the developers' expectations more.

It is also worth noting that the Python results of SANAR and other baselines in Table 6 are worse than those in UniXcoder article. The reason lies in that the input-output pair construction and data pre-processing procedure in SANAR are different from UniXcoder's origin setting. The line-level completion task in UniXcoder/CodeXGLEU aims at completing an unfinished line from arbitrary positions. While in our experiments, the whole line is completed from the beginning. Thus, the results of UniXcoder article and our article are not directly comparable. For the Java dataset, the results of this article and the UniXcoder article are comparable. We hypothesize there are two main reasons: (1) Compared with Java, the Python training set is smaller; (2) Due to the different characteristics between Python and Java (for example, dynamic vs. static), the relevance of the context lines in Python code is lower than Java. Thus, completing a whole line from scratch in Python is harder than in Java.

For the inference latency, SANAR can significantly reduce the inference time within $5 \sim 6\times$ speed-up compared with the AR-based Transformer model. Among all the AR baselines, the deep-shallow transformer architecture (AR 12-1), i.e., 12 encoder layers and 1 decoder layer, can achieve $2.1 \sim 2.4\times$ speed-up compared with 6 encoder layers and 6 decoder layers. However, the generation results become worse. Considering the performance degradation and the modest speed-up, the deep-shallow architecture is not a good choice to improve the efficiency of the AR model upon code completion. UniXcoder is much slower than Transformer, as the number of the decoder's layer is more and the model size is bigger, thus needing more inference time. Compared with neural machine translation tasks, the target sequence in full-line code completion is shorter, thereby the overall speed-up is not as large as in NMT, but is still significant. We also present the latency comparison for completing lines of ≥ 10 tokens, which account for about 30% of the test set. The results on the Java dataset are shown in Table 8. As SANAR can generate tokens in parallel, thus the latency will not be affected by the target lengths, and still keeps comparable with previous results. However, for AR models, as the number of generated tokens increases, the latency increases a lot, and SANAR can achieve $9\times$ speed-up. The results indicate that when completing longer token sequences, the speed-up of SANAR will be more significant.

5.2 Effectiveness of Syntax-aware Training Strategy (RQ4)

To evaluate the effectiveness of our proposed syntax-aware training strategy, which is specially designed for source code, we compare SANAR with the following two strong NAR models in the NLP field.

— CMLM [12]: A strong **Iterative-NAR** model, which adopts a multi-pass decoding strategy to refine the predicted tokens.

Table 8. Speed-up for Completing Long Sequences on Java Dataset

Model	Latency	Speed up
Transformer	171ms	1.0×
UniXcoder	191ms	0.9×
AR 12-1	115ms	1.5×
SANAR	19ms	9.0×

Table 9. Comparison Results against NAR Models

Model	Python			Java		
	BLEU	EM	ES	BLEU	EM	ES
CMLM	23.98	13.44	63.82	28.97	28.36	63.66
GLAT	26.78	16.95	66.36	30.56	28.63	65.80
SANAR	**29.07**	**18.35**	**66.90**	**32.41**	**30.57**	**66.98**

— GLAT [36]: A strong **Fully-NAR** model, which can generate a high-quality target only with single-pass parallel decoding.

We apply these techniques to the line-level code completion task. The results are shown in Table 9. The results show that SANAR outperforms both Iterative- and Fully-NAR baselines. Both GLAT and SANAR are Fully-NAR models, which generate tokens with a single pass, and can achieve better performance than CMLM on full-line code completion. CMLM iteratively decodes the target tokens by repeatedly masking out and regenerating the subset of words that the model is least confident about during inference. Each pass of decoding is pre-trained using the masked language model. These results demonstrate that compared with incorporating an iterative refinement process during inference, a gradual training strategy, which starts from learning the generation of sequence fragments and gradually moving to whole sequences, is more suitable for modeling the token's inter-dependencies in non-autoregressive code completion. During training, GLAT randomly chooses reference words for glancing. Different from GLAT, we propose a syntax-aware training strategy that is specially designed for source code. It dynamically glances code snippets from the target sequence depending on its difficulty and the tokens' syntax types. In this way, SANAR can incorporate the explicit syntactic information of the program and substantially improves the code completion quality.

Besides, it is also interesting to note that these NAR-based models can achieve better performance than the AR-based Transformer models on most metrics. These results are quite different from NMT task, where the NAR models can achieve worse or comparable results to the AR model. The results further verify our assumptions in Section 2, that is, the non-autoregressive manner can fit code completion better than NMT, and can boost the performance on both efficiency and quality for code completion.

5.3 Sensitivity Analysis (RQ5)

5.3.1 Performance of Completing Code of Different Lengths. We further analyze the performance of our model and baselines when completing code lines of different lengths. We divide the test set into three categories according to the length of the target code sequence: $L \leq 5$, $5 < L \leq 10$, and $L > 10$. L means the number of tokens within the target code sequence. Figure 6 presents the results of SANAR and baseline models for completing codes of varying lengths.

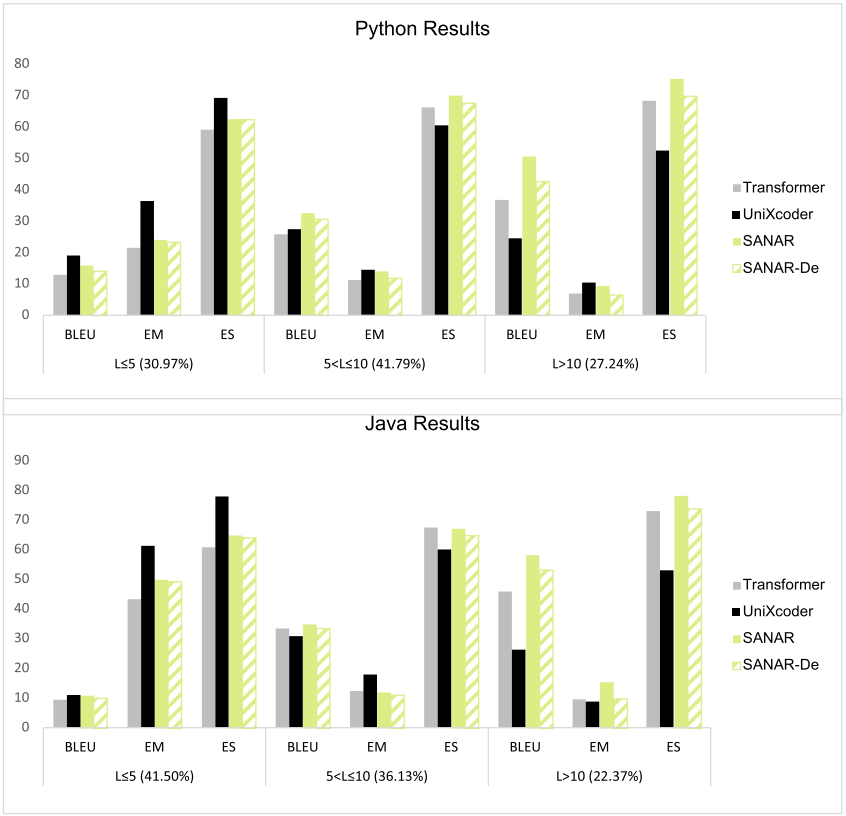

Fig. 6. Performance of completing code of different lengths. SANAR-De illustrates the results of applying SANAR to the Decoder-only model.

As seen from the results, SANAR achieves the top performance in terms of BLEU and Edit Similarity scores when completing the code of longer lengths ($L > 5$). We also find that although UniXcoder performs best when $L \leq 5$, its superiority shrinks dramatically as the length increases. To figure out why UniXcoder performs much better for the short target codes, we make further statistics on the generated results. We find that when completing a short code sequence, UniXcoder prefers to generate short targets compared to SANAR. Specifically, for target code sequences with a length of not more than 5 in the Python test set, the average token number in this set is 3.22. In this particular set, UniXcoder generated results with an average token number of 3.79, while SANAR produced results with an average token number of 4.27. The results on the Java dataset are also similar. Tending to generate short targets will become a disadvantage when the model needs to generate long code lines.

The reason why SANAR recommends longer targets may be attributed to the following factors. UniXcoder generates tokens one at a time, where each token is conditioned on the previously generated tokens. This sequential approach often leads to shorter predictions as the model may make more conservative choices to ensure accuracy. On the contrary, SANAR generates all output tokens in parallel, without relying on the sequential generation process of autoregressive models. Besides, SANAR also introduces randomness (random sampling for $1 - p$ of the time) in the sampling process to enable SANAR to explore more inter-dependency among target tokens. This uncertainty can lead to the model generating additional tokens, resulting in longer outputs.

As the target length increases, the improvement of SANAR becomes more significant. When generating long sequences, except for the generation efficiency, the quality of the generated code of baseline models is also poor due to error accumulation. Our model can generate the tokens in parallel. Thus, the advantages of both generation efficiency and quality are more prominent.

5.3.2 Performance of using Different Architectures in SANAR. To investigate whether SANAR can improve the performance of decoder-only models, we adapt our decoding strategy to Transformer decoder architecture. The following differences between the encoder-decoder and decoder-only models need to be considered when applying SANAR:

- **The attention scope for the input sequence/encoder**. The encoder-decoder models have an explicit encoder to summarize the input sentence with a bi-directional attention matrix, where the semantics of context can be well captured. For the decoder-only model, the input sequence is modeled from left to right.
- **Cross-attention.** Encoder-decoder models have a cross-attention mechanism, which explicitly models the dependencies between the input representation produced by the bi-directional encoder and the target tokens. For the decoder-only model, each token is predicted based on the representation of the previous sequence. Since the non-autoregressive decoding is heavily dependent on the input sequence, when applying SANAR to the decoder-only model, each decoder layer still needs to attend to both input sequence representation and the previous layer of decoder representation.
- **Soft-copy mechanism.** For the first decoding stage of SANAR, which is performed in a fully-NAR way, the input to the decoder $H = \{h_1, h2, \ldots, h_n\}$ are soft-copied from the encoder output. The results of the first decoding are used to guide the sampling for the second decoding procedure.
- **Target length prediction.** In non-autoregressive decoding, one necessary process is to predict the target length based on the input sequence.

Considering the above differences, in order to utilize SANAR with "decoder-only" models that employ unidirectional attention, we introduce specific modifications to our original encoder-decoder architecture. These changes are implemented to approximate the decoder-only mode. (1) Firstly, the attention matrix for the original encoder is transformed into a triangular form to ensure that the input sequence dependencies are modeled from left to right. We name the modified encoder as the "contextual decoder". During decoding, the attention between the decoder layer and the input sequence representation is still maintained. (2) In the initial decoding stage, the input $H = \{h_1, h_2, \ldots, h_n\}$ is soft-copied from the "contextual decoder". (3) Unlike the encoder-decoder architecture, the input sequence representation is computed from left to right. Therefore, the [LENGTH] symbol, which is used for target length prediction, is appended at the end of the input sequence instead of the beginning, since the attention operates from left to right. To summarize, accounting for the non-autoregressive decoding dependency on the input sequence and the inherent mechanisms of SANAR, **the architecture is not entirely the same as typical decoder-only models**. The results of applying SANAR to the decoder-only model, i.e., SANAR-De, are shown in Figure 6. According to the results, we found that the performance of SANAR-De is comparable or slightly lower than SANAR when completing the shorter sequences (L <= 5). As the target length increases, the performance becomes poor, and the discrepancy with SANAR becomes large. Nonetheless, SANAR-De outperforms Transformer in most cases.

5.3.3 Impact of Model Size. Recall that our model contains around half the parameters compared to the UniXcoder. To investigate how the model size impacts the completion performance, we conduct experiments by increasing both the encoder and decoder layers of SANAR to 12, we

Table 10. Results against State-of-the-art Large Models on Python Dataset

Model	BLEU	EM	ES	Latency
UniXcoder	23.79	21.11	61.56	181ms
InCoder	38.75	31.71	69.50	2005ms
CodeGen	34.09	24.25	67.52	1052ms
SANAR	29.07	18.35	66.90	20ms
SANAR-large	33.21	24.33	68.05	26ms

Table 11. Results against State-of-the-art Large Models on Java Dataset

Model	BLEU	EM	ES	Latency
UniXcoder	21.59	33.74	65.85	161ms
InCoder	41.40	38.80	69.89	1846ms
CodeGen	35.91	35.19	68.12	853ms
SANAR	32.41	30.57	66.98	19ms
SANAR-large	35.75	34.46	67.88	24ms

name it as SANAR-large. Besides, we also add two popular **Large Language Model (LLM)**-based code generation approaches, CodeGen-350M-multi [33] and InCoder-1B [11], as new baselines. Both of these two models are large-scale generative code language models, pre-trained on a huge corpus encompassing Python and Java programs. These models can be directly used for line or block completion tasks. In our experiments, we execute the inference code using the officially released model checkpoints and accompanying scripts. Since these models were primarily designed for generating code statements, rather than focusing solely on single lines, to ensure a consistent and fair comparison, we keep the first line of the results in the evaluation. Note that CodeGen-305M-muti and InCoder-1B are around 2 × and 10 × larger than SANAR-large, respectively. Both CodeGen and InCoder use a huge number of Python and Java programs collected from GitHub repositories to pre-train.[10] Consequently, it is highly likely that they have encountered some of the programs in our test set during their pre-training phase. Therefore, it is crucial to understand that the results obtained from these models may not be directly comparable to others that have not had such exposure during pre-training. The results for Python and Java datasets are shown in Tables 10 and 11.

As seen from the results, when comparing SANAR-large with SANAR, we observe a substantial improvement in performance as the model size increases in both Python and Java datasets. SANAR also outperforms UniXcoder in all the evaluation metrics. However, it's worth noting that the larger model size led to a longer inference time compared to SANAR. Nevertheless, thanks to our parallel decoding technique, SANAR-large still demonstrates much faster inference than other AR baselines. Thus, the significance of the enhancements achieved by SANAR becomes even more pronounced in such scenarios with bigger model sizes. When comparing SANAR-large with CodeGen (350M), we find that their performance was comparable, SANAR-large even outperforms CodeGen in EM and ES scores in the Python dataset. It is worth noting that despite being trained with a larger code corpus and having a bigger model size (approximately 2 times larger than SANAR-large), the statement completion performance of CodeGen does not show a significant improvement. Additionally, the larger model size results in longer inference times. Furthermore, when comparing with

[10]The training dataset of both the two models is not publicly available.

Table 12. Performance of Different Completion Start Points

Strategy	Model	Python			Java		
		BLEU	EM	ES	BLEU	EM	ES
Random	SANAR	43.97	**41.77**	**73.31**	**33.96**	**32.91**	**68.08**
	UniXcoder	**44.22**	41.35	71.40	32.79	32.03	65.97
Middle	SANAR	**44.85**	**40.28**	**75.80**	**33.65**	33.31	67.71
	UniXcoder	43.89	39.65	73.31	32.06	**33.50**	**68.01**
Beginning	SANAR	**29.07**	18.35	**66.90**	**32.41**	30.57	**66.98**
	UniXcoder	23.79	**21.11**	61.56	21.59	**33.74**	65.85

the much larger model InCoder (1B), which had a sharp increase in model size (around 10 times larger), both performance and inference time shows significant changes, that is, the performance greatly improved, but unfortunately, the inference time also increased accordingly.

To sum up, as the model size becomes larger (≥ 1B), the performance would get an obvious promotion, and the inference time also increases sharply accordingly. The significance of the enhancements achieved by our non-autoregressive decoding strategy will become more pronounced in such scenarios. Since larger models require significant computational resources, including powerful hardware and large amounts of memory. Due to our limited resources, it may not be feasible for us to train such large models by applying our approach. Nonetheless, our approach has shown promising results on the Transformer model under a smaller size. We believe it can bring benefit for the bigger models.

5.3.4 Impact of Completing Start Point. Following previous work and tools [6, 49], we decide to predict the whole line from the start in our experiments. However, this completion approach introduces more ambiguity compared to when the model is provided with a few tokens about the line to be completed. It is important and interesting to explore the impact of predicting code from different positions within a line. In this section, we conduct experiments with two different design choices for completing start point, including predicting from the middle of a line and predicting from a random point of a line, and compare the results with UniXcoder. Specifically, we modify the data pair construction strategy to update our dataset, and use it to tune the model with 1 epoch. For the random point scenario, we randomly split the original target line into two sections and incorporate the former portion into the input data. Similarly, for the middle point scenario, we divide the line at its midpoint and merge the preceding section with the input data. The results are shown in Table 12. We also list the results of our initial completing strategy, i.e., predicting the whole line from the beginning, in the last two rows of the table.

As seen from the results, the performance of completing a code line from the middle and random position is better than from the beginning, especially in the Python dataset. As we previously mentioned, due to the inherent differences between Python and Java, such as their dynamic and static nature respectively, the relevance of context lines in Python code is generally lower compared to Java. As a result, completing a whole line in Python poses greater challenges than in Java. When the model is provided with a few tokens about the line to be completed, the model can realize the developer's detailed intent and provide targeted recommendations. Furthermore, we have observed that SANAR exhibits more significant performance improvements in these two scenarios. SANAR consistently outperforms UniXcoder across most metrics on both datasets. These results further suggest that when provided with more definitive contextual information, SANAR is capable of making superior decisions during the inference process.

To sum up, compared with completing a whole line, predicting from the middle or from a random point takes the established context into account, potentially offering more accurate and

Table 13. Performance of Reference Word Selection Strategies

Strategy	Python			Java		
	BLEU	EM	ES	BLEU	EM	ES
Random	26.78	16.95	65.94	30.56	28.63	65.80
Most-confident	24.35	16.00	65.32	31.21	29.68	66.45
Most-inconfident	24.53	15.35	64.51	31.01	30.44	66.01
HSG	**29.07**	**18.35**	**66.90**	**32.41**	**30.57**	**66.98**

contextually appropriate completions. However, it also needs additional processing time to analyze the context and find appropriate completions. While predicting from the beginning could be perceived as more user-friendly since it offers immediate suggestions and assists users in early error prevention. However, if the dependencies are unclear in the contextual lines, there is a possibility of introducing a higher degree of uncertainty and irrelevant suggestions. Therefore, the choice of code completion strategy depends on various factors, including the nature of the code being written, the specific use case, desired user experience, and the like. Determining a superior strategy tends to be a challenging task. Exploring a combination of strategies that harness the individual strengths of each approach may unlock the potential for the ultimate code completion experience. This calls for further investigation and experimentation, which will be an integral part of our future work.

5.3.5 Impact of Sampling Strategy. HSG sampling strategy is used for selecting reference tokens as extra decoding input given the sampling number N, depending on the model's current performance and the target tokens' syntax types. To evaluate the effectiveness of the proposed HSG sampling strategy, we conduct experiments with different token selection strategies. In HSG, the tokens are selected depending on their syntax types. Besides the HSG strategy, we also introduce three selection approaches for comparison. The first one is the random select strategy, which assumes all the words in the reference are equally important and randomly chooses N reference words for glancing. Inspired by CMLM [12], the other two strategies relate to model's prediction confidence: "most-confident" and "most-inconfident". We use the probability distribution score of the output softmax layer to evaluate the model's prediction confidence. For the most-confident strategy, we choose N positions with a higher prediction confidence score as extra decoding input. For the most-inconfident strategy, we choose the positions with a lower prediction confidence score for glancing. The results for different selection approaches are shown in Table 13.

As seen from the results, our proposed HSG sampling strategy outperforms other token selection strategies in all the datasets. The results indicate that the tokens with specific syntax types (keyword, operator, and identifier) are more critical for glancing in training, which can enable SANAR to capture more syntactic and symbolic features of the target code tokens. Among all the comparison strategies, the random strategy performs a little better than the two confidence-based strategies in the Python dataset but performs a little worse in the Java dataset. This result demonstrates that introducing more randomness in sampling is more beneficial to Python programs. As a dynamic programming language, the flexibility of the Python code is greater. Thus, more diverse inter-dependency among target words needs to be explored.

5.3.6 Impact of Sampling Probability. To further analyze how the sampling probability p of our HSG sampling strategy affects the model's performance, we conduct experiments with different ps, where a larger p indicates more tokens are sampled depending on their syntax types. When p is zero, the sampling will become pure random. The results for different ps are listed in Table 14. When p is increased from 0 to 0.3, the performance becomes better, demonstrating the effectiveness

Table 14. Performance of Full-line Code Completion with Different Sampling Probabilities in HSG Sampling Strategy

p	Python			Java		
	BLEU	**EM**	**ES**	**BLEU**	**EM**	**ES**
0	26.78	16.95	65.94	30.56	28.63	65.80
0.15	28.20	16.96	66.36	32.16	29.94	66.81
0.3	**29.07**	**18.35**	**66.90**	**32.41**	**30.57**	**66.98**
0.5	27.37	17.31	66.30	31.29	29.82	66.43

Table 15. Error Location Analysis Results

	Python	**Java**
AR	0.84	0.57
SANAR	0.63	0.55

of the syntax-guided sampling. When p is further increased, where the randomness decreases, the performance began to drop, but still outperforms the random strategy ($p = 0$). This suggests that it is also necessary to introduce randomness during the syntax-aware sampling, which can help SANAR explore more inter-dependency among target tokens.

5.3.7 Error Location Analysis. Error accumulation is a known issue of autoregressive decoding, which is a significant cause of its poor performance, especially in long sequence generation. We argue that decoding in a non-autoregressive manner could mitigate this problem. We analyze the error locations of the code generated by SANAR and AR baseline under long sequence generation scenario ($L > 10$). To figure out whether AR model tends to make mistakes at the latter of the sequence, we calculate the average error indexes started from the 5-th position for each sample as follows:

$$\tilde{L} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{E_i}\sum_{j=5}^{L_i}\frac{\text{isError}(j)}{L_i} \tag{14}$$

where N is the sample number, E_i is the total number of the error tokens for i-th sample, L_i is the sequence length of i-th sample, j is the token index of each sample, and $\text{isError}(j) = 1$ if the j-th token does not occur in the target code sequence, or $\text{isError}(j) = 0$. The results are shown in Table 15, where **the large value indicates that the errors tend to occur at the latter part of the sequence**. As seen from the results, the average error indexes of the AR model are bigger than our NAR model for both datasets, and the difference is more obvious in Python. The results verified our hypothesis that decoding in parallel could mitigate the error accumulation to a certain extent. Thus, SANAR can perform better in long sequence generation situations in terms of both efficiency and quality.

5.4 User Study (RQ6)

Since automatic metrics do not always agree with the actual quality of the results, in this section, we first performed a manual verification to evaluate the quality of the automatically generated code by SANAR and the powerful AR baseline UniXcoder, mainly focusing on measuring the similarity between the generated code and the ground truth and the semantic consistency between the generated code and context. Then, we further compare SANAR with GitHub Copilot and OpenAI ChatGPT.

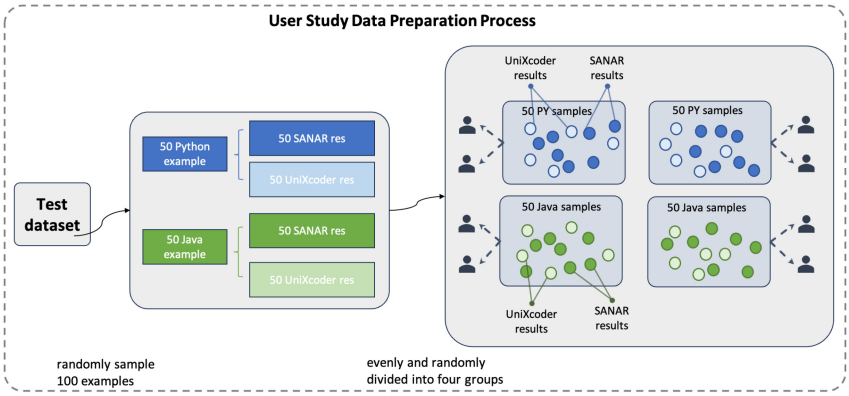

Fig. 7. Illustration of the user study data preparation process.

5.4.1 Human Evaluation. We inviteight participants for 2 hours each to evaluate the generated code in a survey-based study. The participants are computer science Ph.D and Master students and are not co-authors. They have Java and Python programming experience ranging from 2 to 5 years. Figure 7 illustrates the data preparation process. Initially, we randomly selected 100 examples from the test set, with 50 pertaining to Python and 50 to Java. Each sample included code generated by both SANAR and UniXcoder, resulting in a total of 200 data points. Next, we mixed the data points from both models for each language. We then divided the examples equally and randomly into four groups, with two groups assigned to each language. In order to evaluate the results, we enlisted the participation of two assessors for each group. Importantly, every participant assessed the completions from both models, and they were unaware of which approach generated the code being evaluated. Each group is evaluated by two participants. Participants are allowed to search the Internet for related information and unfamiliar concepts. Then we make a questionnaire for each group.

We ask participants to give a score between 0 to 5 to measure the similarity between the generated code and the ground truth and the semantic consistency between the generated code and context. The score criterion is shown as follows:

0: The generated code contains syntax errors.
1: The generated code is syntactically correct but is semantically different from the ground truth and is not consistent with the context.
2: The generated code looks similar to the ground truth but has different meanings.
3: The generated code has different semantics with ground truth but is semantically consistent with the context.
4: The generated code and ground truth are functionally equivalent but with different implementations.
5: The generated code and ground truth are identical.

Finally, we obtained 400 scores from our human evaluation, of which 200 are scores for SANAR, and the other 200 are scores for UniXcoder. We regard a score of 0 and 1 as low quality, a score of 2 and 3 as medium quality, and a score of 4 and 5 as high quality. Table 16 illustrates the results of our user study. The proportion of high-quality code generated by SANAR is substantially higher than UniXcoder, and the proportion of low-quality code generated by SANAR is much lower. Moreover, the mean score of SANAR is higher than that of UniXcoder. These results show that the code generated by SANAR is more reliable and semantically consistent with the context and human-written target code.

Table 16. User Study Results

Approach	Low	Medium	High	Mean Score
UniXcoder	39%	38.5%	22.5%	2.365
SANAR	21.5%	43%	35.5%	2.995

Input:
```
...
self._update_cluster_view()
self.cluster_view.select(selected, do_emit=False)
self.cluster_view.next()
if similar :
   self.similarity_view.next()
def attach(self, gui ) :
   self.gui = gui
   self._create_actions(gui)
   gui.add_view(self.cluster_view, name=<STR>)
   if self.quality :
```

Ground Truth:
```
self.cluster_view.add_column(self.quality,
name=self.quality.__name__,)
```

UniXCoder:
```
for i in range(<NUM>) :
```

SANAR:
```
return self.quality
```

Copilot:
```
gui.add_view(self.similarity_view, name=<STR>)
```

Input:
```
...
self.assertTrue(model.fm.shape==(nn, <NUM>))
self.assertTrue(model.fs2.shape==(nn, <NUM>))
def test_GPR(self) :
     print(<STR>)
     model = pyGPs.GPR()
     m = pyGPs.mean.Zero()
     k = pyGPs.cov.RBF()
     model.setPrior(mean=m, kernel=k)
     model.setOptimizer(<STR>, num_restarts=<NUM>)
     model.optimize(X, y)
```

Ground Truth:
```
model.predict(self.zr)
```

UniXCoder:
```
model.predict(z)
```

SANAR:
```
model.predict(self.<unk>)
```

Copilot:
```
self.assertTrue(model.lik.shape==(nn, <NUM>))
```

Fig. 8. Qualitative examples in Python.

5.4.2 SANAR vs GitHub Copilot. GitHub Copilot [6] is a powerful code completion plugin, which uses the OpenAI Codex [3] to suggest code snippets. We sample and run 20 qualitative examples with its extension in VSCode. To feed the data into Copilot, we input the context into IDE and then get the recommended code. There are two primary reasons why we opted not to report the overall accuracy of Copilot. Firstly, Copilot is integrated into an IDE, requiring us to input the code context into our IDE and obtain a response. However, due to potential network issues or other factors, we did not always receive a response. Running the entire test set for accuracy evaluation would also be a time-consuming and labor-intensive task. Secondly, Copilot's backbone model, Codex [3], is pre-trained using an extensive collection of source code from GitHub, which aligns with our training set. This poses a risk of data leakage, making a fair comparison challenging for us. Nevertheless, we maintain a level of curiosity regarding the performance of these powerful models. Consequently, we provide several examples of Copilot's output in our article to showcase its capabilities. It is worth noting that Codex is a very large-scale model based on GPT-3, thus it is much more powerful than SANAR. In most of cases, the recommended statements of both Copilot and SANAR look semantically consistent with the context. In this section, we present several examples where both SANAR and Copilot perform not well, and try to seek future improvement direction.

Figure 8 shows two cases in Python language. In the first case, the generated results of all three models are different, and also different from the ground truth. However, the statement generated by Copilot seems more relevant to the ground truth than SANAR and UniXCoder, that is, "add something". And the statement looks similar to the 9-th line in the input (i.e., `gui.add_view(self.cluster_view, name=<STR>)`. In the second case, both UniXCoder and SANAR make mistakes in predicting the parameter name. However, the Copilot's recommendation is totally different from the ground truth, and just imitates the first two lines of the input.

Figure 9 shows two cases in Java language. In the first case, the generated results of SANAR and UniXCoder look very similar, and are close to the ground truth. The mistake occurs in the data

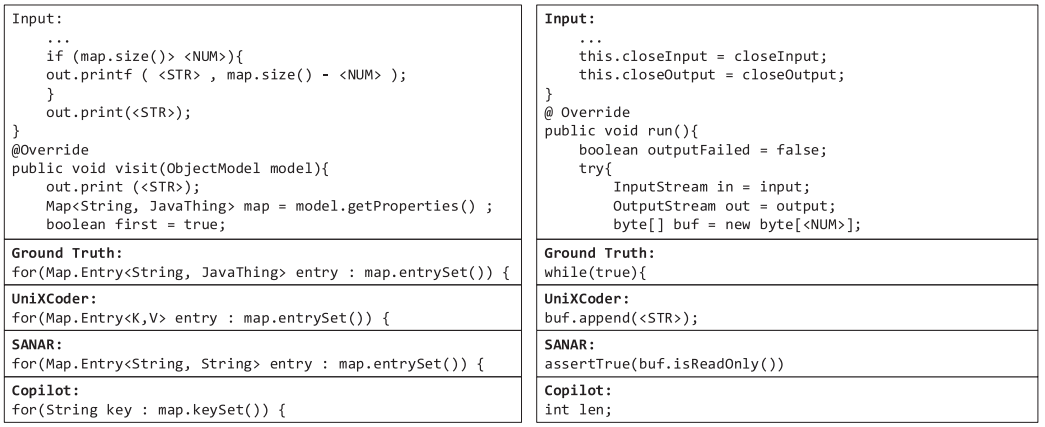

Fig. 9. Qualitative examples in Java.

Table 17. Comparison Results against ChatGPT

Model	Python-Challenging			Python-Common			Java-Challenging			Java-Common		
	BLEU	EM	ES	BLEU	EM	ES	BLEU	EM	ES	BLEU	EM	ES
UniXcoder	11.05	8	50.46	**24.43**	34	69.34	10.92	8	40.94	32.96	34	69.81
ChatGPT	**15.16**	6	49.35	22.56	**36**	69.56	**16.90**	**14**	**57.61**	**39.21**	**38**	68.00
SANAR	13.44	**12**	**53.92**	23.30	26	**72.10**	11.71	4	38.62	36.77	30	**73.81**

type of the key and value. For Copilot, the element type is totally different from the ground truth. In the second case, the generated statements of the three models are totally different, and also different from the ground truth. The recommendations of both UniXCoder and SANAR are related to "buf" object, while Copilot defines a new variable. It is hard to tell which recommendation is better.

To sum up, code completion is a challenging and controversial task. Since the users' intent is ambiguous and hard to infer from the context, even if the prediction is different from the ground truth, it can still be semantically consistent with the context, and also be acceptable sometimes. Besides, these models (including the powerful Copilot) still struggle with the parameter name and data type recommendation, which are more specific to the local context. Thus, clarifying the developers' intent and introducing more local information is necessary, which can be the focus of future research.

5.4.3 SANAR vs OpenAI ChatGPT. Since OpenAI's ChatGPT [34] is powerful, we compared our model with GPT-3.5[11] on a small challenging test set, taking into consideration the price, API request limitations, and the potential issue of data leakage where the test data may have been seen during training. To conduct the comparison, we randomly sampled 200 examples from both Python and Java test sets. We then asked two computer science Master students to identify 50 challenging examples for each language, where they believed completing the task would be difficult given the provided input. To make a comparison, we also ask them to identify 50 common tasks for each language. The results are shown in Table 17.

As seen from the results, the overall performance of the common tasks is better compared to the results of challenging tasks in both Python and Java languages. Besides, the performance of the

[11] https://platform.openai.com/docs/models/gpt-3-5

challenging tasks is much worse compared to the results obtained on the entire test set (Table 6 and 7). This is expected since the selected examples were intentionally more challenging. For the Python-Challenging tasks, ChatGPT demonstrates the best performance in terms of BLEU score. However, SANAR achieves the highest scores in EM and ES, and UniXcoder also outperforms ChatGPT in these two metrics. Overall, ChatGPT shows the ability to correctly predict several n-grams, resulting in a higher BLEU score. However, due to its tendency to generate longer statements, the ES score is lower. Regarding the Python-Common tasks, ChatGPT achieves the best EM score, indicating its superior performance in generating exact matches. On the other hand, SANAR performs best in ES, and UniXcoder achieves the highest BLEU score. Despite these differences, the overall variation in performance is relatively small.

For Java-Challenging tasks, ChatGPT outperforms the other two approaches. This can be attributed to the fact that contextual lines in Java code have stronger relevance, making it easier for ChatGPT to infer the exact intent of the context. On the other hand, completing a whole line from scratch in Python is more challenging, even for ChatGPT, compared to Java. When it comes to Java-Common tasks, ChatGPT displays superior performance compared to the other two approaches in terms of both BLEU and EM scores. However, it falls behind in ES, where SANAR achieves the highest score. This discrepancy can also be attributed to ChatGPT's tendency to generate longer statements, which ultimately leads to a lower ES score.

In summary, ChatGPT demonstrates excellent performance in generating exact matches across common and challenging tasks. However, due to its tendency to generate longer statements, it may result in a relatively lower ES score in most cases. Additionally, ChatGPT performs exceptionally well in Java, particularly in challenging tasks. Conversely, in Python, all three approaches encounter difficulties in both common tasks and challenging tasks.

6 THREATS TO VALIDITY

Threats to internal validity include the influence of the model hyper-parameter settings and the architectural choice. We chose the hyper-parameters through a mix of small-range random grid search and manual selection. Therefore, we believe that there is only a minor threat to the hyper-parameter choosing, and there might be room for further improvement. However, current settings have achieved a considerable performance increase. For architectural choice, SANAR employs an Encoder-Decoder architecture to perform line-level code completion. Another alternative is adopting a Decoder-only architecture. Regarding the results outlined in Section 5.3.2, it is apparent that applying SANAR to the Decoder-only model is indeed possible. However, the performance achieved is not as impressive as that of the Encoder-Decoder model. This outcome can be attributed to the fact that non-autoregressive decoding heavily relies on the input sequence and the intrinsic mechanisms of SANAR. Therefore, we employ seq2seq architecture due to its superior performance compared with the LM-Decoder.

Threats to external validity include the quality and representativeness of the datasets. We evaluate our approach on two widely-used code completion datasets that have been used in previous work [19, 26, 27]. The programs in the datasets are collected from top-ranked and popular GitHub repositories. Thus, most of the programs are expected to be of high-quality. However, there exist cases that do not have a single answer, as shown in our user study. A customized evaluation benchmark might be needed to confirm and improve the usefulness of our model. Besides, further studies are also required to validate and generalize our findings to other programming languages. Additionally, the findings from our empirical study are indeed influenced by the specific datasets used. In order to minimize potential biases and ensure broader applicability of our conclusions, we utilized datasets encompassing both Python and Java languages. Nonetheless, it is crucial to recognize that despite our diligent efforts, there may still exist inherent limitations that could

impact the generalizability of our results. Further evaluation is required to validate and enhance the overall generalizability of the results.

Threats to construct validity relate to the suitability of our evaluation measure. We adopted the popular automatic metrics used by the previous work [49], which measured BLEU-4 score, exact match accuracy, and character-level edit distance similarity. Besides, to prevent the automatic metrics do not always agree with the actual quality of the results, we also performed a manual verification to evaluate the quality of the generated code.

7 RELATED WORK

7.1 Token-level Completion

Previous LM-based code completion models mainly perform the next token completion. Robbes and Lanza [40] proposed to utilize the change history data to improve the results offered by code completion tools. Later, N-gram models are widely used. Hindle et al. [20] first built an N-gram model for code completion. In recent years, with the success of deep learning, deep neural network-based language models have been applied to source code modeling. Li et al. [26] proposed a code completion model based on LSTM and Pointer Network. Karampatsis et al. [22] proposed an open-vocabulary GRU-based LM, which utilized BPE and beam search algorithm to address the OOV problem in code completion. Liu et al. [27] used transformer-XL as the base model to perform code completion and adopted multi-task learning to predict the next token type and value jointly. Kim et al. [25] leveraged the syntactic structure of code by modeling the paths in AST. Liu et al. [28] presented the first pre-trained language model for code completion. Liu et al. [29] merged sequence features with structural features through the utilization of an extended attention mechanism. This integration resulted in an improvement in the completion performance.

7.2 Statistical Line-level Code Completion

Compared with token-level completion, completing an entire line of code can further improve development efficiency, which has attracted many researchers' attention in recent years. Wang et al. [49] first proposed a Transformer-based model to perform line-level code completion. Later, many pre-trained models have been proposed to promote the development of code intelligence [10, 17, 21, 28, 30, 43], and some of them have been used in line-level code completion. Svyatkovskiy et al. [43] proposed Intellicode compose, a pre-trained code completion tool that is capable of completing an entire line of code. They trained a GPT-2 model, an autoregressive pre-trained transformer model, using 1.2 billion LOC written in Python, C#, TypeScript, and JavaScript. Ciniselli et al. [4] conduct an empirical study to explore the capabilities of Transformer-based models in code completion at different levels, including token-level, (multiple)statement-level, and block-level. Izadi et al. [21] presented a multi-token code completion model to perform both single-token and multi-token prediction via multi-task learning. Ding et al. [9] proposed a framework for statement-level completion, which incorporates in-file and cross-file context into existing code language models. Guo et al. [17] proposed UniXcoder, a pre-trained multi-layer Transformer model for source code, which incorporated semantic and syntax information from code comments and ASTs. The experimental results show that UniXcoder obtains significant improvement on many downstream tasks, including line-level code completion. In this article, we compare with powerful state-of-the-art line-level completion approaches, including Intellicode compose [43] and UniXcoder [17].

CodeGen [33] is an expansive open-source code language model designed specifically for code generation tasks. With an impressive parameter count of up to 16.1B, it undergoes pre-training on a diverse set of both natural language and programming language data. The authors have released multiple versions of the pre-trained CodeGen models to cater to different requirements. These variations include CodeGen-NL, CodeGen-Mono, and CodeGen-Multi, which are distinguished

based on their training corpus and parameter initialization. InCoder [11], on the other hand, is another powerful generative code language model with a high parameter count of up to 6.7B. It possesses the capability to infill arbitrary sections of code, enabling it to support a wide range of tasks, such as code generation, code comment generation, variable re-naming, and more. Both of these models exhibit the ability to generate code statements in an auto-regressive manner. Additionally, they can be effectively utilized for line-level completion by retaining the first line of the generated output.

7.3 Efficient Code Completion

Svyatkovskiy et al. [44] present a code completion framework that allows combining different neural components, offering and exploring trade-offs in terms of memory, speed, and accuracy. They found that compared with the RNN/CNN-based models, Transformer-based models are the slowest. A **static analysis-based provider (StAn)** yields accurate and informative completions compared to the original vocabulary-based provider. They use different neural component combinations to deal with the efficiency problem. Besides, there are also a few works that perform completion not in a left-to-right manner, for example, by filling spaces or recommending edits to source code based on the bidirectional context. Cvitkovic et al. [7] form code completion as a fill-in-the-blank task. Specifically, they consider a single usage of a variable in the source code as the target, and replace it with a <FILL-IN-THE-BLANK> token. Then they build the model to predict what variable should have been there. Nguyen et al. [32] proposed an API recommendation tool, LUPE, which can provide a sequence of cohesive API calls. LUPE is an LSTM-based encoder-decoder architecture, where the decoder is auto-regressive. To improve the prediction performance, they trained LUPE by reversing the sequence order of the encoder input following an empirical work [42], which is different from our preliminary empirical study experiment. We aim to figure out how the decoding order impacts the code generation performance by reversing the decoder's input sequence order. Codex also releases an "edit" functionality.[12] Given some code and the instruction for how to modify it, the "code-davinci-edit-001" model will attempt to edit it accordingly. The above approaches can improve the completion efficiency, directly or indirectly, from different aspects. Nonetheless, the left-to-right generation is still widely used in existing code completion and generation research. In this article, we focus on boosting the left-to-right decoding procedure with a syntax-aware non-autoregressive decoding approach.

7.4 NAR Models of Text Generation

Non-AutoRegressive (NAR) models, which generate all the tokens in a target sequence in parallel and can speed up inference, are widely explored in natural language processing tasks such as neural machine translation (NMT). There are two kinds of NAR models: Fully-NAR models [15, 36] and Iterative-NAR models[12, 16].

7.4.1 Fully-NAR. The Fully Non-AutoRegressive model consists of the same encoder as Transformer and a parallel decoder [15]. During training, it uses conditional independent factorization for the target sentence, aiming to decode target tokens in parallel to speed up the generation. A line of work introduced latent variables to reduce the model's burden of learning the dependencies among target tokens [15, 31]. Gu et al. [16] proposed Levenshtein Transformer, which generated a sentence by explicitly predicting edit actions, such as insertion and deletion. Recently, Qian et al. [36] proposed GLAT, which can achieve parallel text generation with only a single decoding pass by gradual training.

[12] https://platform.openai.com/docs/guides/code/editing-code

7.4.2 Iterative-NAR. The conditional independence assumption in Fully-NAR does not hold in general, resulting in inferior performance. To improve the Fully-NAR model, Iterative-NAR approaches are proposed. Ghazvininejad et al. [12] proposed to pre-train a **conditional masked language model (CMLM)**, which first predicts all of the target words non-autoregressively and then repeatedly masks out and regenerates the words that the model is least confident about using in the masking scheme. Sun et al. [41] incorporated a structured inference module into the non-autoregressive models. Specifically, they applied the linear-chain conditional random field on top of NAT predictions to capture token dependency.

8 CONCLUSION & FUTURE WORK

In this article, we propose a non-autoregressive model for code completion, aiming at improving the efficiency and accuracy of line-level code completion. We first perform an in-depth analysis to explore the dependency among the target tokens. Based on the analysis results, we propose SANAR, a syntax-aware non-autoregressive model for line-level code completion. We boost our approach with an adaptive syntax-aware sampling strategy, which can dynamically glance some code snippets from the target sequence depending on its difficulty and the token types. Experimental results show that our approach outperforms state-of-the-art code completion baselines of similar model size as well as significantly reduces the inference time. We are the first to apply the NAR model for generating code.

We believe this work represents a significant advance in code generation, which will be beneficial as a building block for many other code generation tasks. In the future, we shall seek to exploit our decoding strategy for other software engineering applications, for example, program translation, code summarization, and so on. Besides, incorporating domain-specific knowledge into the context, such as the local context and detailed user intent, also holds promise for enhancing the performance of code generation tasks, which will be the focus of our future research.

REFERENCES

[1] aiXcoder. 2018. aiXcoder. https://www.aixcoder.com/

[2] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR'13, San Francisco, CA, USA, May 18–19, 2013*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE Computer Society, 207–216. https://doi.org/10.1109/MSR.2013.6624029

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.

[5] Matteo Ciniselli, Luca Pascarella, Emad Aghajani, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. Source code recommender systems: The practitioners' perspective. *arXiv preprint arXiv:2302.04098* (2023).

[6] Copilot. 2021. Copilot. https://copilot.github.com/

[7] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *International Conference on Machine Learning*. PMLR, 1475–1485.

[8] Damai Dai, Li Dong, Yaru Hao, Zhifang Sui, Baobao Chang, and Furu Wei. 2021. Knowledge neurons in pretrained transformers. *arXiv preprint arXiv:2104.08696* (2021).

[9] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007* (2022).

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP (Findings) (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.

[11] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A generative model for code infilling and synthesis. *ArXiv* abs/2204.05999 (2022).

[12] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. Mask-predict: Parallel decoding of conditional masked language models. 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 2019, Hong Kong, China, November 3–7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 6111–6120. https://doi.org/10.18653/v1/D19-1633

[13] Zi Gong, Yinpeng Guo, Pingyi Zhou, Cuiyun Gao, Yasheng Wang, and Zenglin Xu. 2022. MultiCoder: Multi-programming-lingual pre-training for low-resource code completion. *arXiv preprint arXiv:2212.09666* (2022).

[14] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*. 369–376.

[15] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O. K. Li, and Richard Socher. 2018. Non-autoregressive neural machine translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30–May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[16] Jiatao Gu, Changhan Wang, and Junbo Zhao. 2019. Levenshtein transformer. In *NeurIPS*. 11179–11189.

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. (2022), 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[18] Tianyu He, Xu Tan, Yingce Xia, Di He, Tao Qin, Zhibo Chen, and Tie-Yan Liu. 2018. Layer-wise coordination between encoder and decoder for neural machine translation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3–8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7955–7965.

[19] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[20] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. (2012), 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[21] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. *arXiv preprint arXiv:2202.06689* (2022).

[22] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: Open-vocabulary models for source code. In *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1073–1085. https://doi.org/10.1145/3377811.3380342

[23] Jungo Kasai, James Cross, Marjan Ghazvininejad, and Jiatao Gu. 2020. Non-autoregressive machine translation with disentangled context transformer. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13–18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5144–5155.

[24] Jungo Kasai, Nikolaos Pappas, Hao Peng, James Cross, and Noah A. Smith. 2021. Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021*. OpenReview.net.

[25] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22–30 May 2021*. IEEE, 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[26] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13–19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 4159–4165. https://doi.org/10.24963/ijcai.2018/578

[27] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *ICPC'20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13–15, 2020*. ACM, 37–47. https://doi.org/10.1145/3387904.3389261

[28] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21–25, 2020*. IEEE, 473–485. https://doi.org/10.1145/3324884.3416591

[29] Yapeng Liu, Zhiqiu Huang, Yaoshen Yu, Yasir Hussain, and Lile Lin. 2022. Improving code completion by sequence features and structural features. In *Proceedings of the 4th World Symposium on Software Engineering*. 51–58.

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[31] Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard H. Hovy. 2019. FlowSeq: Non-autoregressive conditional sequence generation with generative flow. 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 2019, Hong Kong, China, November 3–7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 4281–4291. https://doi.org/10.18653/v1/D19-1437

[32] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2023. Fitting missing API puzzles with machine translation techniques. *Expert Systems with Applications* 216 (2023), 119477.

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[34] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. 311–318.

[36] Lihua Qian, Hao Zhou, Yu Bao, Mingxuan Wang, Lin Qiu, Weinan Zhang, Yong Yu, and Lei Li. 2021. Glancing transformer for non-autoregressive neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1–6, 2021*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 1993–2003. https://doi.org/10.18653/v1/2021.acl-long.155

[37] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732* (2015).

[38] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30–November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 731–747. https://doi.org/10.1145/2983990.2984041

[39] Yi Ren, Jinglin Liu, Xu Tan, Zhou Zhao, Sheng Zhao, and Tie-Yan Liu. 2020. A study of non-autoregressive model for sequence generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5–10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 149–159. https://doi.org/10.18653/v1/2020.acl-main.15

[40] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17 (2010), 181–212.

[41] Zhiqing Sun, Zhuohan Li, Haoqing Wang, Di He, Zi Lin, and Zhi-Hong Deng. 2019. Fast structured decoding for sequence models. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8–14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 3011–3020.

[42] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems* 27 (2014).

[43] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: Code generation using transformer. In *ESEC/FSE'20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8–13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[44] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR'21)*. IEEE, 329–340.

[45] Tabnine. 2018. Tabnine. https://www.tabnine.com/

[46] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16–22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 269–280. https://doi.org/10.1145/2635868.2635875

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference*

on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[48] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. Practitioners' expectations on code completion. *arXiv preprint arXiv:2301.03846* (2023).

[49] Wenhan Wang, Sijie Shen, Ge Li, and Zhi Jin. 2020. Towards full-line code completion with neural language models. *arXiv preprint arXiv:2009.08603* (2020).

[50] Yau-Shian Wang, Hung-Yi Lee, and Yun-Nung Chen. 2019. Tree transformer: Integrating tree structures into self-attention. *arXiv preprint arXiv:1909.06639* (2019).

[51] Zejun Wang, Fang Liu, Yiyang Hao, and Zhi Jin. 2023. AdaComplete: Improve DL-based code completion method's domain adaptability. *Automated Software Engineering* 30, 1 (2023), 11.

[52] Bingzhen Wei, Mingxuan Wang, Hao Zhou, Junyang Lin, and Xu Sun. 2019. Imitation learning for non-autoregressive neural machine translation. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28– August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 1304–1312. https://doi.org/10.18653/v1/p19-1125

[53] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[54] Kechi Zhang, Ge Li, and Zhi Jin. 2022. What does transformer learn about source code? *arXiv preprint arXiv:2207.08466* (2022).

Received 27 March 2023; revised 30 January 2024; accepted 8 February 2024