RETROcode: Leveraging a Code Database for Improved Natural Language to Code Generation

Anonymous ACL submission

Abstract

As the availability of text and code has increased, large-scale pre-trained models have demonstrated considerable potential in tackling code generation problems. These models usually apply a supervised fine-tuning approach, training on pairs of natural language problem statements and corresponding ground-truth programs. However, the strategy of increasing the model size and training data quantity, despite potential performance improvements, also inflates computational costs and can lead to overfitting (Lai et al., 2022). Considering these issues, we introduce RETROcode, a novel adaptation of the RETRO architecture (Borgeaud et al., 2022) for sequence-to-sequence models, that strategically employs a sizable code database as an auxiliary method for model scaling. Unlike approaches that solely increase model and data size, RETROcode enables the model to directly access a large code database for making predictions. This provides an efficient mechanism to augment language models with substantial-scale memory. Our work includes an empirical analysis of methods for integrating information from natural language and code from database in the generation process. Leveraging a large database, we outperform classic architectures with similar number of parameters on our test sets and we achieve results that are getting closer to Codex despite it having a significantly larger parameter and training data size.

1 Introduction

Code generation is the task of automatically creating computer programs from natural language, generating potentially previously unseen code. It has a wide range of applications, from creating code snippets for developers to generating complete software applications. In recent years, the increasing availability of large amounts of code and natural language data has facilitated the development of powerful neural network models that can perform code generation with high accuracy.

One challenge in working with large amounts of natural language and code data is the lack of aligned examples, which require human expertise to annotate. To address this issue, one approach is to use large pre-trained models that have been trained on a large volume of code and/or natural language data, and then fine-tune them on the available annotated data (Xu et al., 2020; Wang et al., 2021; Chen et al., 2021; Li et al., 2022). The use of large models with a high number of parameters can provide computational benefits during training and inference, as well as improved memorization of the training data. However, training these models can be computationally expensive, and the large number of parameters may lead to overfitting on the training data (Bender et al., 2021; Karmakar et al., 2022).

An alternative approach for translating natural language to code is code retrieval, which involves searching for and retrieving an appropriate code snippet from a code database (Wan et al., 2019; Ling et al., 2021; Gu et al., 2021). However, these methods are becoming less commonly used as it is now possible to use pre-trained models that are trained on the entire code database and generate personalized code responses to a given query.

Methods for natural language generation often involve the use of generative models that are trained to associate text with data in a database. These solutions have two main advantages: they allow for the separation of world knowledge from language learning, and they enable the use of smaller model sizes. For instance, the Knn-Based Composite Memory system (Fan et al., 2021) assists a conversational agent by providing access to information from similar discussions and by supplying relevant knowledge from various sources based on the input user prompt. Another example is RETRO architecture (Borgeaud et al., 2022), which pro-

vides information to a language model as decoding goes using sentences similar to what was generated. In both cases, queries are made to a database by comparing the embeddings of the input or output with those in the database to obtain the nearest neighbours, and the resulting information is provided to the encoder or decoder, respectively.

In this paper, we introduce RETROcode, a transformer-based architecture that integrates a sequence-to-sequence architecture into Borgeaud et al. (2022). This facilitates the simultaneous processing of dual inputs: natural language utterances and analogous code snippets retrieved from a database. Our strategy strives to harness the extensive available code data while minimizing the model parameters.

We present two methods for integrating this information within the decoder and conduct an in-depth analysis of the impact of various critical components on system performance. Our results outperform architectures with an equivalent number of parameters and are the close to Codex's performance in our code generation task, albeit with significantly fewer parameters. This article delivers the following contributions:

- It establishes a novel transformer sequence-to-sequence architecture that combines information from natural language input and similar code from a database.
- It investigates the impact of key architecture components on system performance, including database preprocessing, database code size, and two distinct methods for integrating database information into the decoder.
- It proposes an effective hybrid database to not only take advantage of the large amounts of code available but also of natural language to code alignments.

This article is organized as follows: In Section 2, we provide a formal description of the query system used to retrieve neighbours from the database during decoding. In Section 3, we detail our model architecture, including two methods for merging natural language intent with information retrieved from the database. In Section 4, we highlight the critical elements of our architecture. Finally, in Section 5, we conduct experiments to examine the various key elements of our model, comparing 3 different approaches to generate code from natural language.

2 Query Architecture

In this Section, we describe the database query system which is designed to retrieve similar codes from the database in response to a query as illustrated in Figure 1. The function $Query^k(C_q)$ is defined to take a code chunk of size m and return its k-nearest code neighbours from the database. The embedding of the current code C_q is calculated and compared to the embeddings from the database with an L_2 distance.

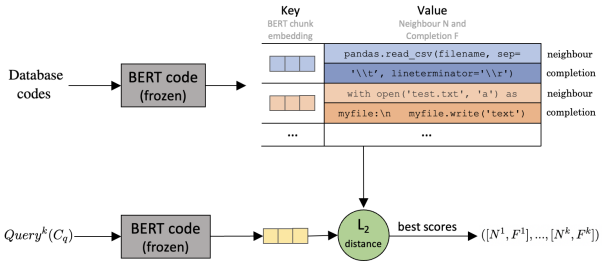

Figure 1: Process of $Query_k(C_q)$ to obtain k-nearest neighbours and their continuation. Here, the chunk length m to construct the database is equal to 8.

We first introduce the database organization and then explain how the query system is designed.

2.1 Database structure

We structure our database $\mathcal{D}$ as a key-value memory. Each value consists of two continuous chunks of code tokens of size m, referred to as $[N, F]$. N is the neighbouring chunk that is used to compute the key, while F is the continuation of the code from N, adding information. The key embedding is then computed with a frozen CODEBERT on N.

We choose to use a frozen model for the embedding calculation to optimize the efficiency of the database query system as it avoids to re-compute embeddings over the entire database during training. It further enables the addition of new code chunks to the database after training.

Note that the concatenation $[N, F]$ is not necessarily a complete snippet of code, it depends of the size of m which is one of the crucial parameters of our model.

2.2 Neighbours Retrieval

Given such a database, the query embedding of C_q is also built with a frozen CODEBERT. To retrieve the k-nearest neighbours and their continuations

from $\mathcal{D}$, we use the L_2 distance:

$$Query_k(C_q) = (\mathcal{N}_1, ..., \mathcal{N}_k) \text{ where } \mathcal{N}_i = [N_i, F_i]$$

Note that for a database of T elements, we query the approximate nearest neighbours in $O\log(T)$, relying on the Faiss library (Johnson et al., 2021) [1].

3 Model

3.1 Objective

We consider a family of models that generate a code Y from a natural language description X. The models have a generic form:

$$p(Y \mid X) = \prod_t p(Y_t \mid Y_{<t}, X) \tag{1}$$

where $Y = \{Y_t : t \in [\![1, L]\!]\}$ and $X = \{X_i : i \in [\![1, n]\!]\}$. The decoding objective aims to find the most-probable hypothesis among all candidate hypotheses by solving the following optimization problem:

$$\hat{Y} = \underset{Y}{\text{argmax}}\, p(Y \mid X) \tag{2}$$

3.2 Baseline

To address this problem, we consider as a baseline the classic transformer architecture from Vaswani et al. (2017) with some minimal changes: we replace Layer Normalisation with Root Mean Square (RMS) normalisation (Zhang and Sennrich, 2019) and use rotary embedding (Su et al., 2021). As we employ residual connections (He et al., 2015) between each sub-layer followed by a RMS normalization. We define the notations:

$$\text{Sublayer}^{\text{FFW}}(X) = \text{RMSNorm}(X + \text{FFW}(X))$$
$$\text{Sublayer}^{\text{SA}}(X) = \text{RMSNorm}(X + \text{SA}(X))$$
$$\text{Sublayer}^{\text{CA}}(X, Y) = \text{RMSNorm}(Y + \text{CA}(X, Y))$$
$$\text{Sublayer}^{\text{CCA}}(X, Y) = \text{RMSNorm}(Y + \text{CCA}(X, Y))$$

where RMSNorm is a Root Mean Square normalization, FFW is a fully-connected feed-forward network. The self-attention SA and the cross-attention CA are classically defined as in Vaswani et al. (2017) with MultiHead(Q, K, V) where Q, K, and V are the query, key, and value matrices respectively. The chunked-cross attention CCA is defined as in Borgeaud et al. (2022) to handle the interaction between the model and the retrieved data in chunks without breaking autoregressivity (see Appendix A for details). We can then define the encoder's layer for natural language as:

$$\begin{aligned} \text{ENC\_NL}(X) &= \text{Sublayer}^{\text{FFW}}(H) \\ H &= \text{Sublayer}^{\text{SA}}(X) \end{aligned} \tag{3}$$

and the code decoder's layer as follows:

$$\begin{aligned} \text{DEC}(X, Y) &= \text{Sublayer}^{\text{FFW}}(C_{nl}) \\ C_{nl} &= \text{Sublayer}^{\text{CA}}(E, C) \\ E &= \text{ENC}(X) \\ C &= \text{Sublayer}^{\text{SA}}(Y) \end{aligned} \tag{4}$$

The methodology for computing hidden representations using a transformer has been detailed. To predict code tokens, our approach is to leverage a standard application of the softmax function across the model's vocabulary. However, to address the challenge of rare word terms, especially relevant in the context of very specific variable names, we enhance this with the inclusion of a pointer network (Vinyals et al., 2015). In accordance with methodologies outlined in Yin and Neubig (2018); Beau and Crabbé (2022), the final output layer of our model is a fusion of a softmax distribution over the vocabulary and the results derived from the pointer network. This design ensures an effective balance between handling general language structures and accommodating specific OOV terms.

3.3 Gathering neighbours' information

Crucially, our proposed architecture incorporates a transformer guided by neighbours retrieved from an external code database. Here, we explain how we integrate information from the code database into the code generation process, that is the value returned by the $Query_k(C_q)$ function.

The information from the retrieved neighbours must be encoded to be integrated into the decoder. Each encoding of $\mathcal{N}_i$ is conditioned with the code already generated by the decoder (Y) as in Borgeaud et al. (2022):

$$\begin{aligned} \text{ENC\_NB}(\mathcal{N}_i, Y) &= \text{Sublayer}^{\text{FFW}}(H) \\ H &= \text{Sublayer}^{\text{CA}}(C, E) \\ C &= \text{Sublayer}^{\text{SA}}(Y) \\ E &= \text{Sublayer}^{\text{SA}}(\mathcal{N}_i) \end{aligned} \tag{5}$$

As a result, the encoding E_{nb} of the neighbours is the concatenation of the encoding of each retrieved

[1] https://github.com/facebookresearch/faiss

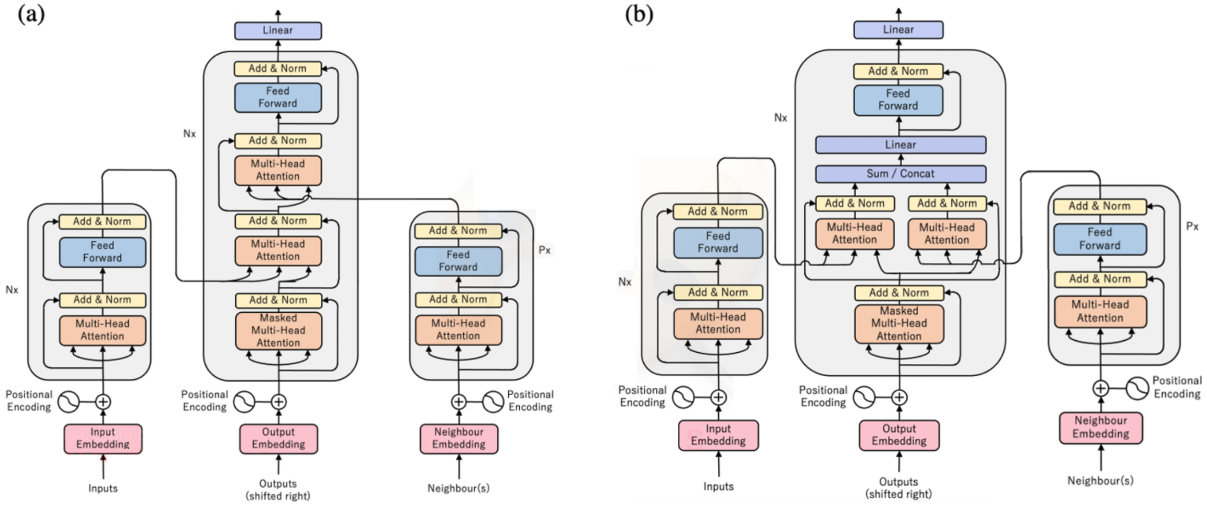

Figure 2: Illustration of the RETROcode architecture, which includes two variations for integrating neighbour encoding into the baseline model [2]. (a) Sequential aggregation: we incorporate the information from the neighbours into the code generation process using a two-step process. First, we use the classic cross-attention mechanism to combine the information from the natural language. Then, we perform a second cross-attention between the output of the first cross-attention and the neighbours. This process is described in equation 6. (b) Parallel aggregation: we separately compute the information from the neighbours and the natural language with the decoder using cross-attention, and then merge the results with a linear layer as described in equation 7.

neighbour:

$$
\begin{aligned}
E_{nb} &= \text{ENC\_NB}(\mathcal{N}, Y) \\
&= [\text{ENC\_NB}(\mathcal{N}_1, Y) : \ldots : \text{ENC\_NB}(\mathcal{N}_k, Y)]
\end{aligned}
$$

Note that we cannot use directly the database key to feed our decoder since we wish to integrate not only similar codes but also their continuation, which are not included in the key computation.

3.4 Decoding with natural language and neighbours

As we aim to feed the decoder with information from natural language and the retrieved neighbours to guide upcoming predictions, we describe here an update of equation 4 taking advantage of the embedding E_{nb} gathered from the neighbours. To do this, we use two different methods (Figure 2).

First, we introduce the sequential aggregation where the neighbour information is mixed with the natural language information thanks to the cross-attention (as represented on the left of Figure 2):

$$
\begin{aligned}
\text{DEC}(X, Y, \mathcal{N}) &= \text{Sublayer}^{\text{FFW}}(C_{nb}) \\
C_{nb} &= \text{Sublayer}^{\text{CCA}}(E_{nb}, C_{nl}) \\
E_{nb} &= \text{ENC\_NB}(\mathcal{N}, Y) \\
C_{nl} &= \text{Sublayer}^{\text{CA}}(E_{nl}, C) \\
E_{nl} &= \text{ENC\_NL}(X) \\
C &= \text{Sublayer}^{\text{SA}}(Y)
\end{aligned}
\tag{6}
$$

The second solution computes the cross-attention between the neighbours and the natural language in parallel and then aggregate the information through a linear layer (as shown on the right of Figure 2):

$$
\begin{aligned}
\text{DEC}(X, Y, \mathcal{N}) &= \text{Sublayer}^{\text{FFW}}(C_{merge}) \\
C_{merge} &= \text{Linear}(C_{nb} + C_{nl}) \\
C_{nb} &= \text{Sublayer}^{\text{CCA}}(E_{nb}, C) \\
E_{nb} &= \text{ENC\_NB}(\mathcal{N}, Y) \\
C_{nl} &= \text{Sublayer}^{\text{CA}}(E_{nl}, C) \\
E_{nl} &= \text{ENC\_NL}(X) \\
C &= \text{Sublayer}^{\text{SA}}(Y)
\end{aligned}
\tag{7}
$$

[2] Here, the neighbour's encoder is not constrained by the code being generated as in the original RETRO architecture and it is a classic transformer encoder. We drop it because it does not impact the results. See Appendix B for further details.

The neighbour encoding provides a strong signal to the decoder, so we use equations 6 and 7 every p layers and otherwise we use the baseline equation 4.

4 Dataset and preprocessing

In this Section, we describe the characteristics of the CoNaLa dataset on which we have tested our different architectures, the available code data to construct our database and the creation of the database.

4.1 Dataset

In this study we use one specific dataset, CoNaLa, to perform our code generation task.

CoNaLa is a comprehensive corpus, comprising approximately 600,000 pairs of natural language expressions and their corresponding Python code fragments, sourced from StackOverflow. Among these examples, a subset of 2,879 pairs has undergone meticulous manual cleaning by professional developers, which significantly enhances their quality. This subset is further divided into a training set comprising 2,379 pairs and a fixed test set containing 500 pairs. All results reported in the article are based on these manually curated examples, unless stated otherwise. We created a fixed development set by extracting 200 examples from the total 2,379 examples within our training data.

Substitution We preprocess the CoNaLa dataset by normalising the names of variables and constants which are denoted by quotes in the natural language of the 2379 manually curated examples as done in (Yin and Neubig, 2018; Beau and Crabbé, 2022; Zhou et al., 2023). This is done by substituting the actual names of the variables with a predefined set of normalized names that the statistical model can recognize. For example, all variables are renamed to `var_0`, `var_1`, etc. and all lists are renamed to `lst_0`, `lst_1`, etc. in both the natural language and code.

Evaluation To compare with previous work, we report the standard evaluation metric for CoNaLa. Hence, we report corpus-level BLEU and compare with other works on the fixed test set.

4.2 Database creation

The database's construction is the backbone of our model's framework. Here we delve into the specifics of the dataset used for building our **classic database** as described in 2.1, and further explore the varied permutations surrounding this dataset.

Our intent is to harness the potential of the 600,000 code snippets extracted from CoNaLa. Given their intrinsic noise and sporadic alignment with natural language, these snippets raise a significant challenge when incorporated into model training. Nevertheless, the high volume of these snippets - corresponding to Python idiomatic tasks on StackOverflow - potentially holds a value for our model during its generation phase. We consistently draw from the totality of the 600,000 available codes with the 2,379 clean examples to construct our database. However, initial variations are introduced by modulating the length m of the segments, consequently leading to databases of different sizes (since each code is divided into m chunks and each chunk corresponds to a single entry in the database). We vary the chunk length from 2, 4 and 8 (not counting the continuation which is of the same size of m) because in average code snippet from CoNaLa are of length 14.08. Additionally, we introduce variations in the code snippets by integrating them 'as is' into the database or employing the substitution mechanism to standardize the codes by replacing the variable names (the heuristic for replace variable names and examples of substitution mechanism for mined examples are given in Appendix D).

One limitation of the classical approach is the absence of constraints for the initially generated tokens. To leverage the statement and code snippet pairs, we opt to create a **hybrid database** by integrating natural language embeddings as keys, along with the corresponding initial code segments as values. This integration guides the initial stages of our decoding process (Figure 3 in Appendix A). This variation aims to assist the model in generating the correct beginning of the code, which can be critical. Incorrect initial code sequences could lead to error propagation that becomes challenging to rectify in later generation steps, resulting in incorrect retrieval of neighbors as well.

We use codeBERT [3] to construct database embeddings and utilize Faiss [4] to form the index.

5 Experiments

The experiments compare three strategies for code generation. We start by describing our experimen-

[3] https://github.com/microsoft/CodeBERT
[4] https://github.com/facebookresearch/faiss

tal protocol, highlighting the critical parameters utilized in our experiments[5]. Then, we provide an analysis of the baseline transformer approach, thoroughly detailed in Section 3.2.

To test the contribution of the database, we first evaluate an enhanced version of the baseline transformer with the **classic database** (Section 3.4). This experiment is designed to investigate the effectiveness of augmenting the model with a broader context of code structure and familiar patterns.

Third, we investigate the **hybrid database** approach, enhancing the database with natural language to constraint the decoding process at the beginning.

5.1 Methodology

Given the amount of code at our disposal, we leverage CODEBERT for natural language encoding, thereby ensuring its compatibility with our pre-existing seq2seq architecture. This approach is apt, considering CODEBERT's training not only involves code but also incorporates document strings corresponding to that code, thereby imbuing CODEBERT with capabilities for understanding natural language.

For encoding and decoding tasks associated with the neighbours, we use 6-layer transformers equipped with 8 heads, maintaining hidden dimensions at a constant 256. We adhere to a fixed dropout of 0.4 across all cross-attention layers.

In all our experiments, we use two neighbours and use cross-attention every three layers as recommended by Borgeaud et al. (2022). To optimize our model training, we precompute the neighbours during the database creation phase. Thus, our experimental strategy encompasses evaluating different chunk size configurations within the database, and also assessing the impact of variable name replacement. For the decoding process, a beam width of 15 was employed.

5.2 Baseline

Table 1 summarizes the evaluation results of our two baseline configurations on our development set.

The first setup uses a system size of 168M parameters and yields a BLEU score of 35.19 ± 0.63 trained on the 2379 cleaned examples from CoNaLa.

[5]The code of our experiments is publicly accessible and can be found at `anonymized address`.

System	Size	BLEU
Baseline	168M	35.19 ± 0.63
Baseline + 100k mined	168M	38.05 ± 1.08

Table 1: Baseline results on the development set. The scores reported are the mean and standard deviation resulting from training with 5 different seeds.

The second configuration incorporates an additional 100,000 mined examples into the system. The integration of these mined examples significantly enhances the model's performance, leading to a higher BLEU score of 38.05 ± 1.08.

The improvement observed in the 'Baseline + 100k mined' configuration highlights the effectiveness of augmenting the training set with mined examples. This observation supports the hypothesis that using mined examples can indeed serve as a significant strategy to improve the performance of code retrieval tasks.

5.3 RETROcode with classic database

We now evaluate our models using the classic database to guide code generation. More specifically, we study the impact of different key variables, such as the substitution mechanism in the database, the chunk size 'm', and the method for aggregating the neighbours (either sequentially or in parallel), on the performance of our model.

Architecture	Substitution	chunk size m	BLEU
Parallel	False	2	32.98 ± 0.93
		4	28.53 ± 1.05
		8	31.56 ± 0.72
	True	2	34.54 ± 0.58
		4	30.27 ± 0.74
		8	34.14 ± 0.29
Sequential	False	2	34.35 ± 0.36
		4	29.09 ± 0.24
		8	31.71 ± 0.49
	True	2	$\mathbf{35.23 \pm 0.53}$
		4	31.59 ± 0.67
		8	34.60 ± 0.65

Table 2: Comprehensive comparison of BLEU scores, each obtained from five different training sessions, on the development set by varying key parameters: system architecture (sequential or parallel), implementation of the substitution mechanism in the database, and the chunk size utilized in constructing the database. Each score represents a mean value along with the associated standard deviation.

From Table 2, we observe that in all cases, the model performance is worse than that of our baseline, despite being trained on the same number of examples. A qualitative manual observation re-

vealed that this disappointing behavior comes from generation errors at beginning of the sequence that are further propagated. The initial tokens of code are indeed generated without information from the neighbours (see Appendix E for detailed output examples). The erroneous prefixes cause the query mechanism to retrieve similar beginning erroneous chunks, diverting our model from the correct path and consequently reducing the BLEU score significantly. From manual inspection again, we observe that the initial tokens of code generated are not fundamentally incorrect, but still different from the ground truth.

Before providing a solution to overcome this problem, let us first highlight the main trends for our different variables.

System Architecture The results illustrate a significant variation between the parallel and sequential architectures. The sequential architecture appears to yield higher BLEU scores compared to the parallel one, particularly when the substitution mechanism is employed. It seems that cross-attention is a better way to merge information from natural language and neighbours rather than use a separate cross-attention treatment with a final linear layer.

Substitution The implementation of a substitution mechanism consistently enhances the model's performance across both architectures and all chunk sizes. This increase in BLEU scores signifies that normalization of variable names through substitution can greatly help in retrieving appropriate neighbours and accurately predicting code. This is expected, given that one of the main difficulties in predicting code lies in predicting the variable name as described by Beau and Crabbé (2022). Furthermore, the retrieval of neighbouring codes is enhanced by substitution, which standardizes the code that has the same objective but uses different variable names.

Chunk Size The impact of chunk size on model performance appears intricate, with no explicit pattern discernible from the Table. BLEU scores vary, not strictly correlating with the size of the chunks. For example, sometimes the smaller chunk size of 2 improves the results, likely by enabling the model to process more localized information from its neighbours. Conversely, larger chunk sizes, such as 8, also deliver good results as they let the model operate more independently during generation, with neighbouring data having less impact on the code's tail end. In the case of an intermediate chunk size of 4, however, the model seems to retrieve less relevant information, thus leading to confusion and potentially lower-quality code generation.

5.4 RETROcode with hybrid database

To avoid mismatches between the generated code and the reference code, we propose an initial stage of inference driven by a **hybrid database** build from CoNaLa's clean and noisy pairs. By associating natural language embeddings (as key) with the beginnings of related codes (as value), we can query the database using the natural language input statement and retrieve corresponding code beginnings. This thereby guides the model from the generation's outset. The results for this method are detailed in Table 3.

Architecture	Substitution	chunk size m	BLEU
Parallel	False	2	35.87 ± 0.71
		4	32.22 ± 0.38
		8	36.81 ± 1.07
	True	2	36.09 ± 0.90
		4	33.75 ± 0.23
		8	37.76 ± 1.06
Sequential	False	2	39.10 ± 0.79
		4	35.28 ± 0.50
		8	43.03 ± 1.18
	True	2	39.45 ± 1.08
		4	36.20 ± 1.17
		8	$\mathbf{43.56 \pm 0.81}$

Table 3: Exhaustive comparison of BLEU scores attained from five different training instances on the development set. Parameters echo those in Table 2, but with the initial database now augmented by the hybrid database; natural language embeddings (keys) are matched with the beginnings of corresponding codes (values). All scores represent means and corresponding standard deviations.

The implementation of the hybrid database significantly enhances performance across all configurations. It notably achieves a BLEU score of 43.56 with a chunk size of 8, exceeding the baseline + 100k by 5.5 BLEU points. We observe empirically, that this method constraining generation from the very beginning often leads to codes closely resembling the ground truth, especially when m=8, allowing the model to frequently clone first neighbours that closely mirror the ground truth (see Appendix F for detailed output examples).

The observations for the different factors, as discussed in 5.3, remain consistent.

5.5 Test set

Finally we compare in table 4 our best models against other state of the art systems on CoNaLa from 5.4. Additionally, to assess the robustness and general applicability of our model, we employ an alternative dataset, CodeXGlue (Lu et al., 2021), with different properties.

System	Size	BLEU	CodeBLEU
ChatGPT-3.5-turbo [6]	?B	**53.15**	**60.50**
Codex (Chen et al., 2021)	12B	43.16	-
CodeT5 + DocPrompting (Zhou et al., 2023)	220M	36.22	-
CodeT5 (Wang et al., 2021)	220M	34.57	-
kNN-BERTranX (Zhou and Chen)	240M	37.29	39.04
BERTranX (Beau and Crabbé, 2022)	130M	34.20	-
RETROcode (parallel) + hybrid db	180M	38.23	38.50
RETROcode (sequential) + hybrid db	176M	**43.09**	**44.18**

Table 4: Comparative analysis of system evaluated on the CoNaLa.

CoNaLa Test We present result on Table 4. All systems use pre-training on external sources; for instance, we use CODEBERT as the natural language encoder, whereas BERTranX utilizes BERT, and CodeT5, a seq2seq architecture, is pre-trained on the CodeSearchNet dataset (Husain et al., 2019). ChatGPT and Codex, on the other hand, are pre-trained on a vast, undisclosed dataset. A unique strategy is seen in Zhou et al. (2023)'s approach, which enhances CodeT5's performance by incorporating additional information retrieved from a documentation database. BERTranX focuses on generating syntactically correct Python code through the construction of abstract syntax trees with a grammar-based decoder, while kNN-BERTranX enhances this with a grammar database. Our method, RETROcode, distinguishes itself in this competitive field by surpassing systems with similar scale and data sources by almost 5 BLEU points. It closely approaches the performance level of Codex, despite being significantly smaller in size — 66 times less than that of Codex — when tested on the CoNaLa dataset. However, it is important to note that our system still trails behind ChatGPT, which benefits from a considerably larger scale with possible data contamination and is more finely tuned for developer assistance.

CodeXGlue Test The CodeXGlue dataset comprises 250,000 training examples and 15,000 test examples, each pairing a docstring with its corresponding Python function. This dataset poses a distinct challenge compared to development aid tasks, as it requires the generation of complete functions rather than mere one-liners. To adapt to this different coding requirement, we custom-build our database using the CodeXGlue training set, supplemented with examples from the Stack dataset (Kocetkov et al., 2022). In this context, we maintain the use of two neighboring data points but increase the chunk size to $m = 32$ to accommodate the complexity and length of the required code generation. Redcoder-Ext also utilizes a code database, but its approach involves appending the retrieved code tokens directly to the input for processing through a pre-trained seq2seq model. Meanwhile, GAP-Gen advances Python code generation by emphasizing fine-tuning over pre-training and utilizes Syntax-Flow and Variable-Flow to guide its generation process. In our assessments on the CodeXGlue dataset, our sequential RETROcode model demonstrates superior performance, surpassing Redcoder-Ext by nearly 3 BLEU points and 4 CodeBLEU points. This improvement is likely attributable to a more refined process of integrating neighboring data and managing the information flow.

For both datasets, we compute the $r(C)$ metric as utilized in RETRO which quantifies the overlap between test and database examples for both dataset. For CoNaLa, with $m = 8$, we obtain a value of $r(C) = 7.3\%$ while for CodeXGlue with $m = 32$, we get $r(C) = 10.2\%$.

System	Size	BLEU	CodeBLEU
ChatGPT-3.5-turbo[7]	?B	**40.36**	**54.47**
Redcoder-Ext (Parvez et al., 2021)	140M	24.43	30.21
GAP-Gen (Zhao et al., 2023)	220M	22.3	24.1
RETROcode (parallel) + hybrid db	180M	23.54	25.87
RETROcode (sequential) + hybrid db	176M	**27.41**	**33.92**

Table 5: Comparative analysis of system evaluated on the CodeXGlue.

6 Conclusion

In this paper, we introduced two novel seq2seq architectures to leverage natural language and a sizable code database for improved code generation. Our results reveal that the best way to integrate information from natural language and database neighbors is through direct cross-attention. We also identified the necessity to guide the initial stage of our generation process, achievable through a hybrid database that maximizes the benefits of the rich code resources and aligned pairs embedded in the dataset.

[6] Evaluated on December 10, 2023. The date is specified to account for ongoing advancements in the ChatGPT model.

[7] Evaluation made on December 15th 2023.

Limitations

One limitation is that the model size is limited when scaling up with the database, but this also results in an increase in computation time due to the need for periodic database queries. Specifically, the baseline model processes each test example in approximately 0.11 seconds on average, while our enhanced model with a chunk size of 8 exhibits an average processing time of 0,38 seconds.

Following the evaluation protocol used by CoNaLa and CodeXGlue, we use the BLEU and codeBLEU scores, but they do have inherent limitations. Firstly, the BLEU score does not vouch for the executability of the code - a single erroneous token can lead to a compilation error, despite high BLEU scores. Secondly, both scores do not accommodate for multiple viable codes capable of accomplishing the same task. In subsequent work, we plan to enrich our datasets and evaluation protocol with unit tests specifically testing the syntax and semantics of the generated code. The inclusion of such tests is expected to facilitate the formulation of more relevant metrics tailored for code generation evaluation.

Another critical limitation lies in the construction of the database. Caution is required to prevent the inclusion of hazardous or confidential code, which could pose security risks if utilized by our model. Ensuring the safety and integrity of the database content is paramount to avoid these potential dangers.

References

Nathanaël Beau and Benoît Crabbé. 2022. The impact of lexical and grammatical processing on generating code from natural language. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 2204–2214. Association for Computational Linguistics.

Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Margaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big? In *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, pages 610–623. ACM.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 2206–2240. PMLR.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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. *CoRR*, abs/2107.03374.

Angela Fan, Claire Gardent, Chloé Braud, and Antoine Bordes. 2021. Augmenting transformers with knn-based composite memory for dialog. *Trans. Assoc. Comput. Linguistics*, 9:82–99.

Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. Cradle: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547.

Anjan Karmakar, Julian Aron Prenner, Marco D'Ambros, and Romain Robbes. 2022. Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation. *CoRR*, abs/2212.02684.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf,

Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 TB of permissively licensed source code. *CoRR*, abs/2211.15533.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2022. DS-1000: A natural and reliable benchmark for data science code generation. *CoRR*, abs/2211.11501.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5):88:1–88:21.

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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. 2021. Roformer: Enhanced transformer with rotary position embedding. *CoRR*, abs/2104.09864.

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*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Neural Information Processing Systems*.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 13–25. IEEE.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 6045–6052. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pages 7–12. Association for Computational Linguistics.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. 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*, pages 12360–12371.

Junchen Zhao, Yurun Song, Junlin Wang, and Ian Harris. 2023. GAP-gen: Guided automatic python code generation. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 37–51, Dubrovnik, Croatia. Association for Computational Linguistics.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Xiangyu Zhang1 Yu Zhou and Guang Yang1 Taolue Chen. Syntax-aware retrieval augmented code generation.

A Chunked cross-attention details

Our retrieval transformer model divides the output sequence into smaller segments, and uses information from previously processed segments to

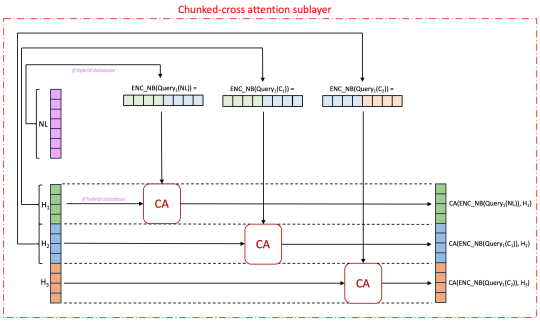

Figure 3: Illustration of chunk-cross attention mechanism with chunk length $m = 4$. This illustration introduces a variation of the database, discussed in 4.2, featuring a hybrid database.

improve the accuracy of its predictions for the current segment. Specifically, the model retrieves text that is similar to the previous segment and uses this information to inform its predictions for the current segment. During training, it is important to not break autoregressivity of the model giving neighbours information too early to the decoder. To ensure that the model maintains autoregressivity during training, we use chunked-cross attention for neighours as in Borgeaud et al. (2022) where the input sequences are divided into smaller chunks, and the model performs cross-attention within each chunk.

Here's a breakdown of the process:

- The input sequence is divided into smaller chunks. The sequence denoted as Y is split into l chunks, each of size m. This means that the hidden state C is represented as a set of smaller chunks, denoted as $(C_u = (C_{um+i})_{i\in[\![1,m]\!]})$, where u is an index that ranges from $[\![1, l]\!]$.
- After the sequence is chunked, chunked cross-attention is computed between each chunk C_u and its corresponding neighbour encodings $E_{nb,u}$. For each chunk C_u, for each token $i \in [\![1, m]\!]$, we define:

$$\textsc{Cca}(C, E_{nb}) = \textsc{Ca}(C_{um+i}, E_{nb,u})$$

- There's a special case for the first m tokens, which can't attend to any neighbour of a previous chunk. For these positions, cross-attention is defined as the identity. This means that for all tokens j in the range from 1 to m, the output of the chunked cross-attention operation is just the input itself, i.e., $\textsc{Cca}(C, E_{nb})_j = C_j$.
- It's also important to note that this process is autoregressive, which means that the output of the chunked cross-attention operation at position i depends on all the tokens from position 0 to i that have been input into the operation $\textsc{Cca}$.

This chunked cross-attention mechanism allows the model to handle sequences of data efficiently, by focusing on smaller chunks of the sequence at a time, while maintaining the ability to learn dependencies between different parts of the sequence through the cross-attention operation.

B Neighbours constrained encoder

The architecture proposed by Borgeaud et al. (2022) suggests that the generated code should constrain each layer of the neighbour encoder.

A straightforward strategy would be to use an encoder architecture akin to the natural language encoder, featuring two sub-layers; one for self-attention and one for feed-forward operations, as follows:

$$\begin{aligned} \text{ENC\_NB}(\mathcal{N}) &= \text{Sublayer}^{\text{FFW}}(H) \\ H &= \text{Sublayer}^{\text{SA}}(\mathcal{N}) \end{aligned} \tag{8}$$

We evaluate these two methods using our optimal model architecture - a sequential model with a hybrid database, as detailed in section 5.4. The database is preprocessed to normalize variable names, and it employs a chunk size of $m = 8$. The results of this comparison are presented in Table 6.

System	Constrained Encoder	CoNaLa BLEU
RETROcode (sequential)	False	43.56 ± 0.81
RETROcode (sequential)	True	43.03 ± 0.31

Table 6: Analysis of results from the development set, gathered from five distinct seeds, for our optimal model, both with and without the constrained neighbour encoder. Each BLEU score is expressed as an average value, accompanied by its corresponding standard deviation.

Interestingly, the BLEU scores show a marginal decrease when employing the constrained encoder approach. However, the standard deviation associated with the constrained method is notably lower, implying more consistent performance across different seeds. Hence, we decided to use a classical encoder for all experiments.

C Datasets mined examples

We present in Table D different examples from the CoNaLa mined examples used to construct our database.

Intent	Snippet
Convert binary string to list of integers using Python	`[s[i:i + 3] for i in range(0, len(s), 3)]`
How can I generate a list of consecutive numbers?	`list(range(9))`
Converting byte string in unicode string	`c.decode( 'unicode_escape')`
Python: Get relative path from comparing two absolute paths	`from os.path import relpath`
A python function that accepts as an argument either a scalar or a numpy array	`if isinstance(x, np.ndarray): return y`
Delimit a specific column and add them as columns in CSV	`df.join(c3)`
How can I find start and end occurrence of character in Python	`df1 = df[df['test'] !=df['test'].shift(+1)]`
Making multiple calls with asyncio and adding result to a dictionary	`loop.run_until_complete (asyncio.wait(tasks))`
Python regex matching in conditionals	`match = patt.match(line)`
How do I use matplotlib autopct?	`plt.show()`

Table 7: 10 examples pick randomly from CoNaLa mined examples.

D Database normalized examples

As mentioned in the section 4.2, we can detect and normalise the variable names of the codes to build the database. To detect variable names, we use the astor library [8] to transform each code snippet into an abstract syntax tree. Once completed, we browse the tree's leaves and retrieve the variable names, excluding those corresponding to library calls such as pandas or numpy. Examples of variable normalization are shown in Table 8:

E Inference process for classic database

Code generated at each time step m We showcase outputs at each time step where the model queries the database to provide a deeper understanding of the model's performance for each chunk size m. We exclusively display our top-performing models for each chunk size, corresponding to the sequential architecture coupled with a normalized database.

[8] https://pypi.org/project/astor/

Code	Normalized Code
`results = [r for k in keywords for r in re.findall( k, message.lower())]`	`var0 = [r for k in var1 for r in re.findall( k, var2.lower())]`
`getattr(a, 'print_test')()`	`getattr(var0, 'var1')()`
`json.dumps(geodata)`	`json.dumps(var0)`
`df.groupby([ df.index.date, 'action']) .count()`	`var0.groupby([ str0.count()`
`format(5e-10, 'f')`	`format(5e-10, 'var0')`

Table 8: 5 examples pick randomly from CoNaLa mined examples before and after substitution mechanism

First example

Intent: *count the occurrences of item str0 in list var0*

Ground truth: `var0.count('str0')`

For $m = 2$:

t	Code Generated	Retrieved Neighbours
2	`<s>var0`	`var0)[-1`
		`2)[:, None`
4	`<s>var0.count`	`.count('/`
		`.count('str0`
6	`<s>var0.count('str0`	`'str0')</s>`
		`'str0')</s>`
8	`<s>var0.count('str0')</s>`	-
		-

For $m = 4$:

t	Code Generated	Retrieved Neighbours
4	`<s>len(var0`	`=len(var0 - 7)`
		`<s>len(var0)</s><pad><pad>`
7	`<s>len(var0)</s>`	-
		-

For $m = 8$:

Error Analysis of code generated For $m = 2$, the model correctly generates the necessary code structure and accurately translates the given intent. The small chunk size allows the model to gradually build up the code, getting the initial variable at step 2, then adding the `.count` function at step 4, and finally adding the argument `'str0'` at step 6. It then correctly ends the sequence at step 8.

For $m = 4$, the model fails to accurately translate the given intent. Model generates `len` keyword and retrieves out of context neighbours.

t	Code Generated	Retrieved Neighbours
8	<s>count = sum(1 for item in	<s>count = len([item for item in lst0 if item == str0]))
		<s>count = lst0.count(str0)</s><pad>
12	<s>count = sum(1 for item in var0 if item == str0)</s>	-
		-
		-
		-

For $m = 8$, the model generates a more complex piece of code than the ground truth but it correctly implements the logic of the given intent. This shows that larger chunk sizes can lead to more complex but still accurate translations of the given intent. However, after step 8 the model fails to retrieve any useful neighboring chunks, which could again be due to the larger chunk size.

In summary, a smaller chunk size can be beneficial for the accuracy of simpler code sequences, while a larger chunk size can be better suited to generating more complex code. However, the latter can also lead to issues with retrieving useful neighboring chunks and correcting errors in the generated code.

Second example

Intent: *Joining data from dataframe var0 with data from dataframe var1 based on matching values of column str0 in both dataframes*

Ground truth: `var0.merge(var1, on='str0')`

For $m = 2$:

t	Code Generated	Retrieved Neighbours
2	<s>pd	<s>pd.con
		<s>pd.con
4	<s>pd.merge(	ge(var0,
		ge(var0,
6	<s>pd.merge(var0, var1	var1)</s>
		'str0')</s>
8	<s>pd.merge(var0, var1, on	, on='k
		, on=['
10	<s>pd.merge(var0, var1, on=['	=['lst0']
		=['var0',
12	<s>pd.merge(var0, var1, on=['str0',	str0', 'var1
		=[var4', 'var2
14	<s>pd.merge(var0, var1, on=['str0', on='	on='str0')
		on='str1')
16	<s>pd.merge(var0, var1, on=['str0', on='str0']	str0']</s>)
		var4')]
18	<s>pd.merge(var0, var1, on=['str0', on='str0']</s>	-
		-

For $m = 4$:

For $m = 8$:

Error Analysis of code generated This second example is longer and more complex than the first one.

For $m = 2$, we notice that the code begins to take shape from the second step with the initiation of the "pd" command, a familiar Pandas syntax. As the chunk size is quite small, the code is updated with high frequency, allowing the model to regularly revise its sequence based on new neighboring chunks. However, this approach has a drawback. The model has trouble creating longer, more complex code structures, possibly due to the small chunk size causing it to focus on smaller fragments of code rather than the overall structure.

t	Code Generated	Retrieved Neighbours
4	<s>s1 =	<s>s1=pd.mer)
		cols=str2)</s><pad><pad>
8	<s>s1=pd.mer	pd.merge(var0,
		<s>pd.merge(var0,
12	<s>s1=pd.merge(var0,	q(var0, var1, args
		array(str0, dtype=np
16	<s>s1=pd.merge(var0, var1, '	var2, 'var3', var0
		var1, 'var2':var2
20	<s>s1=pd.merge(var0, var1, 'var1)</s>	-
		-

t	Code Generated	Retrieved Neighbours
8	<s>s1 = pd.mer	<s>s1=pd.merge(var0,var1,how
		<s>df1 = pd.read_hdf('str0',
16	<s>s1=pd.merge(var0,var1,how	ge(var0,var1,how= 'inner',on['str0']
		seq in zip(var0, var0[1:])]</s><pad>
22	<s>s1=pd.merge(var0,var1,how= 'inner' ,on= 'str0')</s>	-
		-
		-
		-

For $m = 4$, the model is making efficient use of the neighboring code chunks to generate a relatively more complex structure. However, it hallucinates at the end copying neighbours. The size of m looks too small to handle complex structure and too big to locally update code.

For $m = 8$, the model correctly generates a more complex structure with a larger chunk size. It begins the "pd.merge" command at the 8th step and finishes the function with appropriate syntax at the 22nd step. However, the model copies the how argument, which was not requested, and as in the other cases, it does not start like the reference, which reduces the BLEU score.

In summary, there are trade-offs associated with the choice of chunk size. A smaller chunk size facilitates more frequent adjustments but may struggle with larger structures, while a larger chunk size may generate more complex code but lacks the granularity to refine it.

F Inference process for hybrid database

Code generated at each time step Here we give an example of the inference process with our best

model only for $m = 8$.

First example

Intent: *Convert a list of lists var0 to list of integers*

Ground truth: `var0=[int(".join([str(y) for y in x])) for x in var0]`

t	Code Generated	Retrieved Neighbours
0	-	<s>[int(".join(str(d) for d in x))
		<s>[".join(str(d) for d in x) for x
8	<s>var0=[int(".join(str(	<s>[int(x) for x in str(var0)]</s><pad><pad>]
		<s>[".join(str(d) for d in x) for x
14	<s>var0=[int(".join(str(d for d in var0</s>	- -
		- -

Error Analysis of code generated The code here is well predicted with our hybrid database thanks to useful neighbours retrieved at $t = 0$. It is interesting to note that even if the code is valid, the BLEU score is not equal to 100 given the dummy variable d predicted by the model.