RETROcode: Scaling Natural Language to Code Translation with a Syntactic and Database-Driven Approach

Anonymous ACL submission

Abstract

As the availability of text and code data increases, large-scale pre-trained models have demonstrated strong performance in code generation through extensive training on vast datasets. However, scaling model size and dataset volume to enhance performance can lead to substantial computational costs and a heightened risk of overfitting. To address these challenges, we introduce RETROcode, an adaptation of the RETRO architecture (Borgeaud et al., 2022) for sequence-to-sequence models that leverages a large external code database as an auxiliary resource. RETROcode enhances efficiency by integrating external memory for code prediction rather than solely increasing model parameters or dataset size. We explore both token-based code generation and grammar-constrained generation to ensure syntactic validity, specifically in low-resource settings. Our results demonstrate that RETROcode not only outperforms similarly sized architectures—particularly when maintaining syntactic correctness—but also approaches the performance of significantly larger models like Codex 12B and Mistral 7B on two development aid datasets, despite being trained from scratch on smaller datasets.

1 Introduction

Code generation entails the automatic creation of computer programs from natural language, facilitating the generation of previously unseen code. This capability is particularly valuable for developers, enhancing productivity through automated support in software development. Recent advancements in large-scale neural models, particularly pre-trained models like Codex (Chen et al., 2021a)—which powers GitHub Copilot—and ChatGPT (OpenAI, 2023), have demonstrated high accuracy in code generation to help developers (Peng et al., 2023). However, these proprietary models are not openly available for fine-tuning or customization, constraining their utility in scenarios where developers require domain-specific adaptations or operate with limited computational resources. Moreover, even open models such as CodeLlama (Rozière et al., 2023), Mistral (Jiang et al., 2023), or StarCoder (Li et al., 2023) present significant computational costs for fine-tuning and inference, highlighting the need for more accessible and efficient methods tailored to specific tasks.

Retrieval-Augmented Generation (RAG) reduces computational costs through a hybrid approach that integrates code retrieval and generative modeling. Instead of depending solely on large, resource-intensive models, RAG retrieves pertinent code or text snippets from external databases to inform the generation process. This methodology allows for smaller model sizes and reduces dependence on extensive training data while maintaining high-quality outputs. By utilizing external data, RAG empowers developers to tailor models to their specific datasets, achieving robust performance with minimal computational overhead. Techniques such as the RETRO architecture (Borgeaud et al., 2022) effectively merge retrieval and generation, embedding sequences to identify the closest database matches, thereby facilitating efficient alternative scaling methods without substantial computational demands.

Moreover, most code generation methods operate at the token level, mimicking natural language generation, but without ensuring the syntactic validity of the output. kNN-BERTranX (Zhang et al., 2023a) addresses this issue by integrating syntax-aware generation through retrieval-based methods. Using a k-nearest neighbors (kNN) algorithm (Fan et al., 2021), kNN-BERTranX retrieves similar code snippets and generates Abstract Syntax Trees (ASTs), ensuring that the generated code adheres to programming language rules. This combination of retrieval efficiency and AST-based generation improves both accuracy and reliability while allowing the model to work with more limited resources.

In this paper, we introduce RETROcode, a transformer-based architecture that incorporates a sequence-to-sequence model into the framework of Borgeaud et al. (2022). Our approach leverages a large code database to retrieve analogous code snippets during generation, allowing developers to work with their own data and use models that minimize computational cost. RETROcode predicts code tokens directly or generates ASTs, facilitating a comparison between token-level and structure-aware generation methods. Our contributions include:

- A novel architecture combining retrieval and generation for resource-efficient code generation tasks.
- A comprehensive analysis of key components, such as variations in database retrieval and generation methods.
- Competitive performance on code aid development benchmarks with minimal training data, enabling developers with limited compute power to achieve results comparable to large-scale models.

This paper is organized as follows: Section 2 describes the query system used to retrieve neighbors from the database during decoding. Section 3 outlines the architecture of our model. Section 4 highlights the key elements of our experimental setup. Finally, in Section 5, we evaluate the performance of RETROcode, comparing token and AST-based approaches for generating 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.

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.

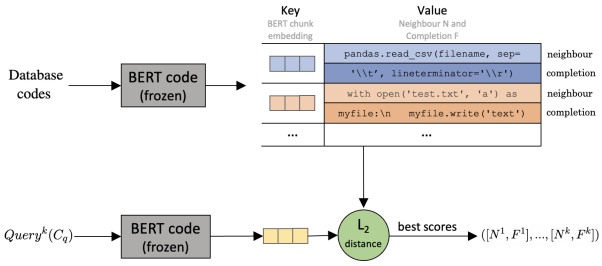

Figure 1: Process of $Query_k(C_q)$ to obtain k-nearest neighbours and their continuation. Here, the chunk length m to construct the database is equal to 8.

We 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).

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:

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

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:

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

Depending on the generation mode, Y can be represented as either a sequence of code tokens or a derivation sequence. The latter ensures syntactic correctness through grammar-based generation. For grammar-based methods, we employ an established transition system based on the Earley parser to generate Abstract Syntax Trees (ASTs) (Yin and Neubig, 2018; Beau and Crabbé, 2022; Zhang et al., 2023a). This system guarantees syntactic validity by flattening the AST into a derivation sequence, which can then be deterministically transformed into valid code.

3.2 Baseline

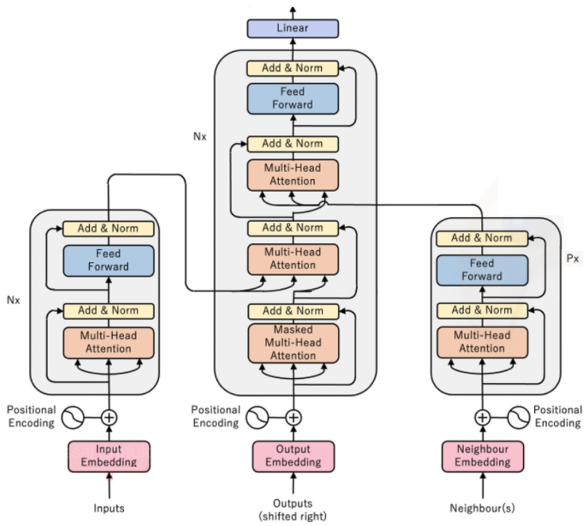

Figure 2: Illustration of the RETROseq2seq architecture. We incorporate the information from the neighbours into the code generation process using a two-step process [1]. First, we use the classic cross-attention mechanism to combine the information from the natural language. Then, we perform a second cross-attention between the output of the first cross-attention and the neighbours. This process is described in equation 6.

We consider the classic transformer architecture from Vaswani et al. (2017) as our baseline with some minor 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 sublayer followed by a RMS normalization. We define the notations:

$$\text{Sublayer}^{\text{FFW}}(X) = \text{RMSNorm}(X + \textsc{ffw}(X))$$
$$\text{Sublayer}^{\text{SA}}(X) = \text{RMSNorm}(X + \textsc{sa}(X))$$
$$\text{Sublayer}^{\text{CA}}(X, Y) = \text{RMSNorm}(Y + \textsc{ca}(X, Y))$$
$$\text{Sublayer}^{\text{CCA}}(X, Y) = \text{RMSNorm}(Y + \textsc{cca}(X, Y))$$

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 $\text{MultiHead}(Q, K, V)$ 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 manage rare variable name terms.

[1] 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.

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 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 introduce the sequential aggregation where the neighbour information is mixed with the natural language information thanks to the cross-attention (as represented in 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 neighbour encoding provides a strong signal to the decoder, so we use equation 6 every p layers and otherwise we use the baseline equation 4.

Additionally, we evaluate an alternative merging method that processes natural language and neighbor information in parallel, as detailed in Appendix C. However, this approach is not used as it performs less effectively than the primary method.

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

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.

CoNaLa was chosen due to its inclusion of a training set and its development aid specialization, distinguishing it from most recent datasets, like DS-1000 (Lai et al., 2022) or Odex (Wang et al., 2023), which provide only evaluation sets. Our objective is to train models from scratch in a low-resource setting for comparative analysis with larger models, a goal not feasible with datasets lacking a training component.

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 2,379 manually curated examples as done in 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 database.

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.To build our database, we combine these snippets with 2,379 clean examples, continuously drawing from the entire pool of 600,000 codes. We introduce variation by adjusting the chunk length m—the number of tokens or derivation steps per database entry—resulting in databases of different sizes. For code snippets, we experiment with chunk lengths of 2, 4, and 8 (not including the continuation of the same size), while for derivation sequences, we use lengths of 4, 8, and 16. These values align with the average snippet lengths from CoNaLa: 14.08 for code and 24.0 for derivation sequences. 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 E).

One limitation 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 [2] to construct database embeddings for code tokens and we trained a grammarBERT model to embed derivation sequences. To do this we finetune the codeBERT model with the original masked language modeling task on derivation sequences to capitalize on its comprehensive training in NLP—vital for constructing the hybrid database—and its proficiency in code token prediction, which could enhance terminal symbol prediction (e.g. variable names). We give training details for grammarBERT in Appendix F. We utilize Faiss [3] to form the index.

5 Experiments

The experiments compare three strategies for code generation. We start by describing our experimental protocol, highlighting the critical parameters utilized in our experiments[4]. Then, we provide an analysis of the baseline transformer approach, thoroughly detailed in Section 3.2. Finally, we test the contribution of the database, 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

[2] https://github.com/microsoft/CodeBERT

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

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

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.

System	Size	BLEU
Baseline (tokens)	168M	35.19 ± 0.63
Baseline (tokens) + 100k mined	168M	38.05 ± 1.08
Baseline (grammar)	168M	35.42 ± 1.75
Baseline (grammar) + 100k mined	168M	39.01 ± 1.11

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

In the second configuration, we enrich the training set by adding 100,000 mined examples, as recommended by Zhou et al. (2023). Despite the availability of 600,000 mined examples in our database, we limit the inclusion to 100k during training to avoid performance degradation due to data noise and misalignment between natural language and code, a challenge well-documented in previous research (Yin and Neubig, 2018; Beau and Crabbé, 2022). Incorporating these 100k mined examples results in a significant performance boost, with the BLEU score improving to 38.05 ± 1.08 for the token-based model.

For grammar-based models, the results exhibit a similar trend. The baseline grammar-based configuration achieves 35.42 ± 1.75, while adding 100,000 mined examples raises the score to 39.01 ± 1.11. This shows that grammar-based generation, which ensures syntactic correctness, performs competitively, benefiting from the additional mined data even more than the token-based approach.

These findings highlight the value of augmenting training with a select number of mined examples, confirming its effectiveness for improving code generation performance.

5.3 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 2.

Mode	Substitution	chunk size m	BLEU
Token	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	43.56 ± 0.81
Grammar	False	4	40.39 ± 0.51
		8	39.90 ± 0.66
		16	44.34 ± 1.01
	True	4	41.95 ± 1.29
		8	40.58 ± 0.75
		16	$\mathbf{44.92 \pm 0.99}$

Table 2: Exhaustive comparison of BLEU scores attained from five different training instances on the development set. Here, natural language embeddings (keys) are matched with the beginnings of corresponding codes (values). All scores represent means and corresponding standard deviations.

The hybrid database consistently improves performance across all configurations compared to the baseline. In the token-based setting, it achieves the best result with a BLEU score of 43.56 at chunk size $m = 8$, surpassing the baseline by 5.5 BLEU points. In the grammar-based setting, the hybrid approach reaches a BLEU score of 44.92 at chunk size $m = 16$, marking the highest improvement over the baseline. These results highlight the effectiveness of the hybrid database in boosting performance for both token-based and grammar-based models, with structured guidance significantly enhancing the quality of code generation.

Let us first examine the key trends for the different variables in our experiments.

Generation Mode The evaluation of generation modes shows a clear advantage for grammar-based methods over token-based approaches. Grammar-based models consistently yield higher BLEU scores, with the best result at chunk size $m = 16$, achieving 44.92 ± 0.99 when using variable substitution. This underscores the benefit of incorporating grammatical rules, which enhance syntactic accuracy and reduce code generation errors, particularly in low-resource settings. In comparison, token-based models exhibit greater performance

variability. At chunk size $m = 2$, they achieve a BLEU score of 39.10 ± 0.79, which decreases to 35.28 ± 0.50 for $m = 4$. However, performance improves again at $m = 8$, reaching a BLEU score of 43.03 ± 1.18. Despite this recovery, grammar-based models generally outperform token-based models across all chunk sizes, demonstrating their robustness and effectiveness in improving code generation quality across various conditions.

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 influence of chunk size on model performance is complex and does not follow a straightforward pattern. For token-based models, smaller chunk sizes like $m = 2$ perform well, but accuracy drops at $m = 4$, while larger chunks like $m = 8$ significantly boost performance. This suggests that small chunks capture more localized information, while larger chunks allow the model to operate more independently, improving generation. For grammar-based models, larger chunks ($m = 8$ and $m = 16$) consistently yield better results, particularly when substitution is used. However, small chunk sizes like $m = 4$ may lead to incomplete syntactic structures, reducing accuracy. This highlights that chunk size influences how effectively the model captures code structure, with larger chunks benefiting grammar-based approaches by providing more context.

5.4 Test set

Finally, we compare our best models against other state-of-the-art systems on the CoNaLa dataset, as described in Section 5.3. To further assess the models' performance beyond BLEU scores, we also evaluate them on the CodeInsight dataset (Beau and Crabbé, 2024), which includes unit tests to more concretely measure functional accuracy.

System	Size	BLEU	CodeBLEU
ChatGPT-3.5-turbo [5]	?B	**53.15**	**60.50**
Codex (Chen et al., 2021a)	12B	43.16	-
CodeT5 + DocPrompting (Zhou et al., 2023)	220M	36.22	-
CodeT5 (Wang et al., 2021)	220M	34.57	-
kNN-BERTranX (Zhang et al., 2023b)	240M	37.29	39.04
BERTranX (Beau and Crabbé, 2022)	130M	34.20	-
RETROcode (token) + hybrid db	180M	43.09	44.18
RETROcode (grammar) + hybrid db	180M	**45.37**	**48.39**

Table 3: Comparative analysis of system evaluated on the CoNaLa.

CoNaLa Test We present the results in Table 3, reporting standard BLEU score and including CodeBLEU (Ren et al., 2020) to provide additional insights into code-specific performance. All models leverage pre-training on external data sources. For example, our method employs CODEBERT as the natural language encoder, while BERTranX relies on BERT, and CodeT5, a seq2seq model, is pre-trained on the CodeSearchNet dataset (Husain et al., 2019). ChatGPT and Codex, in contrast, are pre-trained on large, undisclosed datasets. An interesting approach is seen in Zhou et al. (2023), where performance is boosted by augmenting CodeT5 with documentation-based information. BERTranX, focused on generating syntactically correct Python code, builds abstract syntax trees using a grammar-based decoder, while kNN-BERTranX further improves this by integrating a grammar-based database.

In this competitive landscape, our method, RETROcode, stands out by surpassing several state-of-the-art models of comparable size and resources. The grammar-based RETROcode achieves the best performance, outperforming its token-based counterpart by more than 2 BLEU points, with a BLEU score of 45.37 and a CodeBLEU score of 48.39. Despite being 66 times smaller, RETROcode outperforms Codex.

However, ChatGPT remains the top performer with a BLEU score of 53.15 and a CodeBLEU score of 60.50, likely due to its significantly larger model size and extensive pre-training. While this advantage is clear, concerns about potential data contamination in such large-scale systems remain relevant.

CodeInsight Test CodeInsight dataset expands on CoNaLa by incorporating more complex programming scenarios, particularly through specialized Stack Overflow questions focused on code

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

System	Size	BLEU	CodeBLEU	pass@1
Mistral (Jiang et al., 2023)	7B	54.41	47.83	38.48
CodeLLaMa (Rozière et al., 2023)	13B	58.16	48.85	53.14
Starcoder (Li et al., 2023)	15B	57.21	48.80	52.53
Transformer-tokens	168M	37.52	42.70	17.41
RETROcode (token)	180M	50.71	44.79	34.50
Transformer-grammar	168M	37.57	43.89	18.20
RETROcode (grammar)	180M	51.25	46.29	36.86

Table 4: Comparative analysis of system evaluated on CodeInsight.

libraries. It consists of 3,409 examples, split into 1,549 for training and 1,860 for evaluation, with each evaluation example accompanied by 3 unit tests. The dataset is designed to enhance the evaluation of models in natural language-to-code generation for real-world programming tasks.

In this analysis, we compare a baseline transformer model with the RETROcode model, which utilizes a hybrid database. To accommodate the increased complexity of CodeInsight's examples, which have an average code length of 30.8 ± 15.4 tokens, we adjusted the chunk size to $m = 14$ for token-based generation and $m = 24$ for grammar-based generation. All other hyperparameters and settings are consistent with those used in the CoNaLa phase.

While CoNaLa primarily focuses on one-liner code snippets that handle isolated, simple tasks, CodeInsight features full-function code examples that aim to reflect the complexity of real-world programming challenges. This shift is critical, as CoNaLa's short, self-contained examples lack the structural depth and context necessary, which are central in CodeInsight. To address this gap, we expanded our database by incorporating examples from datasets such as Odex (Wang et al., 2023) and DS-1000 (Lai et al., 2022), alongside the 1,500 examples from the CodeInsight training set, which are closely aligned with real-world programming tasks. This integration involved standardizing function names, argument structures, and task descriptions to match CodeInsight's patterns (e.g., all functions are named `test` and argument names are normalized). Additionally, we increased the number of relevant neighbors by incorporating 10,000 synthetic examples generated by GPT-4, transforming CoNaLa-mined examples to better fit our task. The methodology for constructing our final database is detailed in Appendix H.

Table 4 presents the performance of various models on the CodeInsight dataset. The models have been fine-tuned, and we report results from Beau and Crabbé (2024). Larger models, such as CodeLLaMa (13B) and Starcoder (15B), deliver the highest BLEU and CodeBLEU scores, with CodeLLaMa achieving the best performance overall at 58.16 BLEU and 53.14 pass@1. Their superior results are largely attributed to the advantages of model scale.

In contrast, smaller models like our RETROcode variants (180M) perform competitively despite their size. RETROcode (token) shows a substantial improvement over the baseline transformer (BLEU 50.71 vs. 37.52), highlighting the effectiveness of retrieval-based enhancements. RETROcode (grammar) achieves the best results among our models, with 51.25 BLEU, 46.29 CodeBLEU, and 36.86 pass@1, narrowing the gap with larger models like Mistral (7B) while being much smaller. These comparisons are against LLMs fine-tuned on CodeInsight, underscoring the advantages of grammar-based retrieval and the hybrid database in boosting code generation.

For both datasets, we calculate the $r(C)$ metric, as used in RETRO, to quantify the overlap between test and database examples. In the case of CoNaLa, for the token-based model with $m = 8$, we obtain $r(C) = 7.3\%$, while for the grammar-based model, this value increases to $r(C) = 15.3\%$ at $m = 16$. This higher overlap in grammar-based models is expected, as derivation sequences contain many similar tokens that represent structural grammar rules, whereas terminal symbols, which carry the semantic content, are less frequent. For CodeInsight, our final database configuration gives $r(C) = 11.9\%$ for $m = 14$, rising to 18.3% for $m = 24$.

6 Conclusion

We present RETROcode, a retrieval-based code generation model that approaches the performance of LLMs while being significantly smaller and trained on substantially less data. RETROcode offers practical advantages for industrial applications, particularly due to its lower computational and training demands compared to LLMs. Additionally, its reliance on a customizable database allows for flexible updates by incorporating new examples without retraining the model. The grammar-based RETROcode variant further ensures syntactic correctness, providing an added layer of reliability in code generation.

Limitations

One limitation of RETROcode is its scalability in relation to the size of the database. As the database grows, computation times increase due to the higher frequency of database queries required during inference. For example, while the baseline model processes test examples in around 0.11 seconds each, RETROcode with a chunk size of 8 requires approximately 0.38 seconds per example. To mitigate this, advanced indexing technologies like SCANN and Faiss are employed for k-nearest neighbor searches, enabling rapid querying of large databases, up to 2 trillion tokens, in just 10 milliseconds during evaluations. These technologies significantly improve the model's efficiency in real-time applications.

Another challenge arises from the use of a static CodeBERT model. As Python versions and software libraries evolve, CodeBERT requires updates to stay current. While CodeBERT remains versatile and effective without task-specific modifications, its static nature may limit its adaptability to new developments. Similarly, the grammar-based RETROcode model faces limitations related to its dependency on a specific Python version. As the Python language evolves, even minor changes in syntax or structure can impact the model's performance. Regular updates are essential to keep the model aligned with new Python versions and to adapt to changes in both functionality and language grammar.

The RETROcode model's efficiency in code generation is also heavily influenced by the structure and diversity of the examples stored in the database. This reliance can be a potential weakness, as the model's performance may suffer if the database lacks sufficient variety or fails to cover a broad range of programming paradigms and styles. A limited database can result in suboptimal code generation across different applications. However, this limitation can be addressed by incorporating synthetic examples into the database, designed to fill gaps and extend the model's coverage of diverse coding styles. Importantly, since RETROcode relies on a static BERT model, additional examples can be added to the database without the need to retrain the model. This flexibility allows the model to continuously improve by expanding the database with more diverse and representative examples, enhancing its ability to generate more effective code for a wider range of scenarios.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

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.

Nathanaël Beau and Benoît Crabbé. 2024. Codeinsight: A curated dataset of practical coding solutions from stack overflow. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 5935–5947. Association for Computational Linguistics.

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. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

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. 2021b. 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.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. *CoRR*, abs/2105.09938.

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.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

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

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.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

OpenAI. 2023. Gpt-4: Generative pre-trained transformer 4. https://openai.com. Accessed: 2024-02-06.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of AI on developer productivity: Evidence from github copilot. *CoRR*, abs/2302.06590.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

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*.

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.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1271–1290. 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.

Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023a. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1291–1302. Association for Computational Linguistics.

Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023b. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1291–1302. 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*.

A Chunked cross-attention details

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

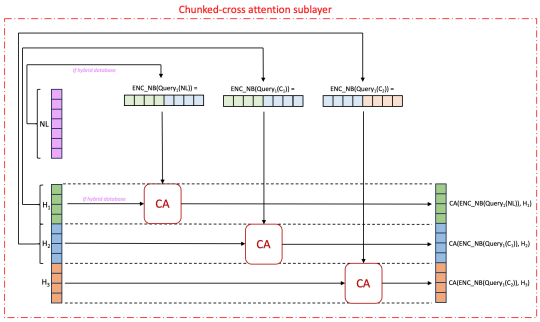

Figure 3: Illustration of chunk-cross attention mechanism with chunk length $m = 4$. This illustration introduces a variation of the database, discussed in 4.2, featuring a hybrid database.

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:

$$\begin{aligned} \text{ENC_NB}(\mathcal{N}) &= \text{Sublayer}^{\text{FFW}}(H) \\ H &= \text{Sublayer}^{\text{SA}}(\mathcal{N}) \end{aligned} \tag{7}$$

We evaluate these two methods using our optimal model architecture - a sequential model with a hybrid database, as detailed in section 5.3. 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 5.

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

Table 5: 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 Parallel Merging Strategy

The second solution we tried computes the cross-attention between the neighbours and the natural language in parallel and then aggregate the information through a linear layer:

$$\begin{aligned} \text{DEC}(X, Y, \mathcal{N}) &= \text{Sublayer}^{\text{FFW}}(C_{merge}) \\ C_{merge} &= \text{Linear}(C_{nb} + C_{nl}) \\ C_{nb} &= \text{Sublayer}^{\text{CCA}}(E_{nb}, C) \\ E_{nb} &= \text{ENC_NB}(\mathcal{N}, Y) \\ C_{nl} &= \text{Sublayer}^{\text{CA}}(E_{nl}, C) \\ E_{nl} &= \text{ENC_NL}(X) \\ C &= \text{Sublayer}^{\text{SA}}(Y) \end{aligned} \tag{8}$$

We evaluate this method using our optimal model architecture - a sequential model with a hybrid database, as detailed in section 5.3 - and report it in Table 6.

Architecture	chunk size m	BLEU
Parallel	2	36.09 ± 0.90
	4	33.75 ± 0.23
	8	37.76 ± 1.06
Sequential	2	39.45 ± 1.08
	4	36.20 ± 1.17
	8	$\mathbf{43.56 \pm 0.81}$

Table 6: Exhaustive comparison of BLEU scores attained from five different training instances on CoNaLa development set for both merging architecture. We vary the size of chunk size m, a critical parameters. All scores represent means and corresponding standard deviations.

Despite the parallel approach achieving reasonable performance, the sequential aggregation consistently outperforms it. For instance, even the lowest-performing sequential model with a chunk size of $m = 4$ scores close to the best parallel model (36.20 ± 1.17 vs. 37.76 ± 1.06 for $m = 8$). This highlights the advantage of sequential aggregation in handling the integration of neighbor information.

D Datasets mined examples

We present in Table 7 different examples from the CoNaLa mined examples used to construct our database.

E 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 [6] to transform each code snippet into an abstract syntax tree. Once completed, we browse the tree's leaves and retrieve the variable

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

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.

names, excluding those corresponding to library calls such as pandas or numpy. Examples of variable normalization are shown in Table 8.

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

F grammarBERT training details

We fine-tune the codeBERT model using the original masked language modeling (MLM) task on derivation sequences to leverage its strong foundation in natural language processing (NLP) and its code token prediction abilities. This approach aims to combine codeBERT's training in natural language and code token tasks for a more specialized application in derivation sequence representation and retrieval, which is essential for constructing our hybrid database.

Handling Vocabulary Differences The training process employs two vocabularies: one for ASDL Python grammar rules and another for terminal symbol prediction. Originally, the terminal vocabulary was derived from the dataset, which risked Out of Vocabulary issues. To address this, we now use the pre-trained codeBERT tokenizer for terminal symbols to ensure compatibility with the database entries.

During preprocessing, code is transformed into a derivation sequence, where grammar rules are mapped to the grammar-specific vocabulary, and terminals are tokenized using the codeBERT tokenizer. This means that terminal tokens, which represent leaf nodes, may consist of multiple subtokens (e.g., "enumerate" tokenizes as `[e, num, rate]`). To manage this, we introduce an additional symbol into the codeBERT vocabulary that indicates the completion of terminal symbol prediction. This symbol serves a distinct role from the REDUCE action, which signals the end of a grammar rule. While REDUCE marks the conclusion of a rule, this new token ensures the proper termination of terminal sequences before the model proceeds with predicting subsequent terminals.

Dataset For this task, we used the Stack dataset, which contains over 6TB of source code across 358 programming languages under permissive licenses. From the Python subset—approximately 100GB from over 20 million files—we randomly sampled around 4 million code snippets for the MLM task. This sample was split into 3,800,000 training examples, 100,000 development examples, and 100,000 test examples.

The dataset also contains a significant amount of natural language in the form of comments and docstrings. To preserve codeBERT's ability to process natural language, one example with a docstring is included for every five examples. This ensures that the model continues to perform well on both code and natural language.

Training Procedure We incorporated grammar rules into codeBERT's pretrained vocabulary to enhance its representation of both grammar rules and terminal symbols. This adapted model, referred to

as grammarBERT, was fine-tuned over 10 epochs with a learning rate of 510^{-5}, using a warm-up period of one epoch.

It is worth noting that Python's ASDL grammar can vary across different versions of the language. For this training process, Python 3.9 was used to ensure compatibility with the latest grammar specifications.

G 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.

H Creation of CodeInsight Database

To address the limitations stemming from the structural differences between the one-liner snippets in the CoNaLa dataset and the full-function code found in the CodeInsight dataset, we expanded the initial database by incorporating examples from additional datasets that are more structurally aligned with our task.

Dataset Augmentation We incorporated datasets containing function-level code with return statements to match real-world programming scenarios. Specifically, we added:

- **Odex (Wang et al., 2023):** 945 natural language and code pairs examples
- **DS-1000 (Lai et al., 2022):** 1,000 natural language and code pairs examples
- **MBPP (Austin et al., 2021):** 974 natural language and code pairs examples
- **HumanEval (Chen et al., 2021b):** 164 natural language and code pairs examples
- **APPS(Hendrycks et al., 2021):** 110,000 Python functions associated with 4,639 intents

Normalization of Added Examples To ensure consistency between the newly integrated examples and CodeInsight dataset, we applied four key transformations to align the structure of the examples:

1. **Function Name Standardization:** All function names were changed to "test" to match CodeInsight conventions.
2. **Argument Normalization:** Function arguments were normalized for consistency with CodeInsight.
3. **Intent Argument Transformation:** Detected argument names in intents were normalized to match the function arguments.
4. **Trimming Intent Descriptions:** Overly long intent descriptions were shortened for brevity.

These steps ensured the added examples align structurally and contextually with our task.

Synthetic Example Generation To further improve database alignment with CodeInsight, we generated synthetic examples to fill structural gaps. To create these synthetic examples, we used GPT-4 to transform the CoNaLa examples to match the structure of CodeInsight's full-function tasks. The synthetic example generation process involved the following pre-prompt:

> "You're an expert-level Python developer. You are tasked with annotating code snippets from Stack Overflow for a code generation task. Transform the code to create a function named 'test' with normalized function argument names (e.g., 'var0', 'lst0', 'df0', etc.), and ensure the function returns the result of the initial snippet."

For example, the code snippet:

```
sorted_list = sorted(text_list, key=
                     lambda x: int(x))
```

was transformed into:

```
def test(lst0):
    return sorted(lst0, key=lambda x:
                  int(x))
```

Another example:

```
data = np.array([[1, 2, 3], [4, 5, 6]])
df = pd.DataFrame(data=data,
          index=["row1","row2"],
          columns=["col1","col2","col3"])
```

was transformed into:

```
def test(arr0, lst0, lst1):
   return pd.DataFrame(data=arr0, index=
                     lst0, columns=lst1)
```

We modified 10,000 CoNaLa examples using this method, balancing cost-efficiency since large-scale GPT-4 data synthesis is expensive. While the generated examples are not always executable or exact replicas of our target dataset, they align well with the structural needs of CodeInsight. GPT-4's advanced capabilities allowed us to produce relevant, compact examples at lower cost, improving the database quality overall.