Co-evolving Data-driven and NLU-driven Synthesizers for Generating Code in Domain Growth and Data Scarcity

Jiasheng Gu[†], **Zifan Nan**[§], **Zhiyuan Peng**[§], **Xipeng Shen**[§], **Dongkuan Xu**[§]
[†]University of Southern California; [§]North Carolina State University;
gujiashe@usc.edu; nanzf1993@gmail.com; jerrypeng1937@gmail.com
xshen5@ncsu.edu; dxu27@ncsu.edu

Abstract

Natural language programming automatically generates code based on a user's text query. Recent solutions are either data-driven or natural language understanding (NLU)-driven. However, the data-driven synthesizer requires a large number of query-code pairs for training, which hinders its application to low-resource programming languages with growing domains whose functionality and grammar can be actively updated. NLU-driven synthesizers solve this problem, but their code generation is slow and their performance rapidly saturates in the presence of ever-increasing data. In this paper, we propose a circular training framework, *Colead*, which co-evolves both the data-driven synthesizer and the NLU-driven synthesizer to achieve high-quality code generation in the presence of data scarcity and domain growth. The NLU-driven synthesizer generates query-code pairs to update the data-driven synthesizer, which shares a part of its updated model to improve the NLU-driven synthesizers, enabling the co-evolution of both. Experiments show that *Colead* gives better results than the baselines in the presence of domain growth and data scarcity, and *Colead* consistently improves the performance of both data-driven and NLU-driven synthesizers over the co-evolvement.

1 Introduction

Natural language (NL) programming aims to automatically generate programming code based on a user's text query (Xu et al., 2022b). It has gained increasing research interest in recent years and has a wide range of applications, not only providing an intuitive programming interface that democratizes artificial intelligence to common users (Chen et al., 2021) but also reducing the time and labor cost of turning ideas into code implementations (Yaghmazadeh et al., 2017; Desai et al., 2016).

Typical NL programming approaches can be categorized as rule-driven or data-driven. A rule-driven synthesizer generates code through prede-

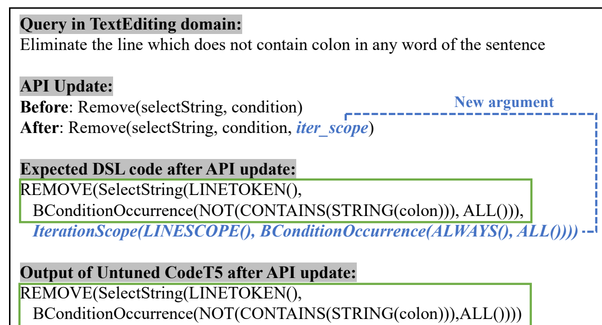

Figure 1: An example of a data-driven model generating code after one API updates in TextEditing (Desai et al., 2016) domain. The API *REMOVE* has a new argument. Without re-training, CodeT5 fails to accommodate this update (shown in blue) and will still generate code for the outdated API (shown in green boxes).

fined domain-specific rules, which requires expert knowledge. It made progress in early NL programming research (Le et al., 2013) but gradually lost its appeal due to technical difficulties in adapting to different programming languages. In parallel, data-driven synthesizers based on deep learning techniques have dominated recent studies (Bavishi et al., 2019; Gu et al., 2016; Li et al., 2022). They generate code through a neural network model. Training of the models requires large amounts of parallel data (Polosukhin and Skidanov, 2018), where each text prompt is paired with a corresponding piece of code. Data-driven synthesizers tend to outperform rule-driven synthesizers in many general-purpose languages (GPLs), like Python, C++, etc., which have massive parallel data available.

However, there are plenty of domain-specific languages (DSLs) that are low-resource. Some DSLs are dedicated to a specific application domain and has few usages and scarce parallel data. The problem of data scarcity hinders the application of data-driven synthesizers. Another challenging problem is the domain growth of programming languages, which are constantly updated in terms of grammar

Proceedings of the 2nd Workshop in Pattern-based Approaches to NLP in the Age of Deep Learning , pages 64–74
December 6, 2023. ©2023 Association for Computational Linguistics

and functionality. Figure 1 shows an example of a data-driven model (CodeT5) that cannot generate correct code after an API update. To adapt to these updates, data-driven synthesizers require frequent re-training, and it would be labor-intensive and time-consuming to collect new data for the update.

Natural language understanding (NLU)-driven synthesizers have recently been proposed as a compromise between rule-driven and data-driven synthesizers (Nan et al., 2020, 2021; Young et al., 2022; Nan et al., 2022). It circumvents the huge need for parallel data in training by utilizing input user queries and API documentation. Whenever the language is updated with a new API and programming grammar, only the API documentation needs to be modified accordingly to generate the new functional code. However, NLU-driven synthesizers have some limitations. To generate code, grammar and dependency graphs need to be built and traversed, which can be time-consuming and a bottleneck to the speed of code generation. As the training data increases, NLU-driven synthesizers quickly reach performance saturation and become inferior to data-driven synthesizers.

In this paper, we propose a co-evolving framework, *Colead* (for "Co-learning Rule and Data from Documention") that combines NLU-driven and data-driven synthesizers to exploit their complementary strengths and enable code generation in the presence of data scarcity and domain growth (Blum and Mitchell, 1998). *Colead* consists of three key components: a retriever, an NLU-driven synthesizer, and a data-driven synthesizer. The retriever fetches information most relevant to the user's text query from the API documentation. Given the information, the NLU-driven synthesizer constructs a grammar graph and a dependency graph to generate code. The generated code is paired with the user query to train the data-driven synthesizer, which in turn shares its updated encoder with the NLU-driven synthesizer, leading to the co-evolvement of both. We summarize contributions as follows:

- We propose a co-evolving framework, *Colead*, that combines data-driven and NLU-driven synthesizers to achieve their complementary strengths and enable code generation in the presence of data scarcity and domain growth.
- Experimental results on two datasets from distinct domains, Text Editing, and ATIS, demonstrate the effectiveness of *Colead* and show that it can work efficiently in domain growth.
- Our study shows that using both synthesizers together can lead to better performance in solving problems than using only one synthesizer. We show that the NLU-driven synthesizer proves effective in addressing the challenge of limited data availability, while the data-driven synthesizer capitalizes on parallel computing for efficient inference. They complement each other, compensating for their respective drawbacks.

2 Related Work

Large language models pre-trained on vast amounts of code have achieved significant progress in recent years (Li et al., 2022; Chen et al., 2021). These models can be classified as encoder-only, decoder-only, or encoder-decoder. An encoder-only model predicts masked code fragments based on their surroundings. It converts code into effective vector representations and facilitates a myriad of downstream tasks, such as code summarization (Ahmad et al., 2020), code classification (Gilda, 2017), and code clone detection (Ain et al., 2019; Fang et al., 2020). The representative models include CuBERT (Kanade et al., 2020), CodeBERT (Feng et al., 2020), and GraphCodeBERT (Guo et al., 2021). By contrast, a decoder-only model, such as CodeGPT (Lu et al., 2021), CODEGEN (Nijkamp et al., 2022), CERT (Zan et al., 2022), and Codex (Chen et al., 2021), predicts the next token given the previous tokens in an auto-regressive manner. However, these models are not a perfect fit for natural language programming tasks.

An encoder-decoder model first uses an encoder to encode the input sequence and then decodes it with a decoder into an output sequence conditioned on the input sequence. CodeT5 (Wang et al., 2021; Le et al., 2022), PLBART (Ahmad et al., 2021), PolyCoder (Xu et al., 2022a), and AlphaCode (Li et al., 2022) are examples of such models in code. Encoder-decoder models perform well on conditional code generation, such as code annotation, natural language programming, etc. Note that both decoder-only and encoder-decoder models can be employed directly for code generation. These models all share the drawback of requiring an abundance of data.

On the contrary, natural language understanding-driven approaches (Nan et al., 2020, 2021; Young

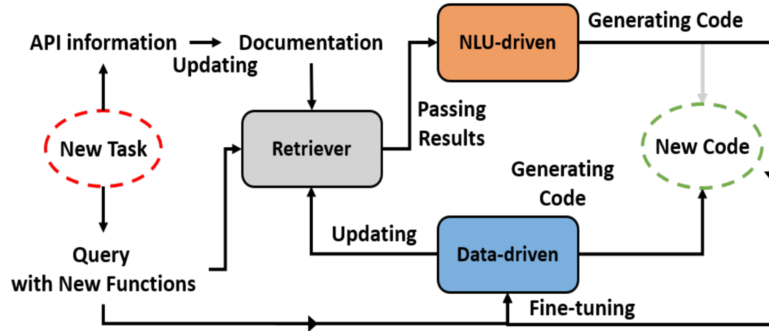

Figure 2: The overview of *Colead*. The main components are the retriever, the NLU-driven synthesizer, and the data-driven synthesizer. Retriever creates a circle by connecting NLU-driven and data-driven components, which serves as a basis for co-evolvement. This circle evolves iteratively to enhance each other's performance.

et al., 2022; Nan et al., 2022) require no training examples. They apply NLP techniques to both natural language (NL) queries and API documentations, extract the key components inside the NL queries and compose the mapped APIs into code expressions following the domain grammar. However, NLU-driven methods are not as powerful as data-driven models when there is a lot of data available for training.

3 Methodology

The major challenge of this work is how to efficiently connect the NLU-driven and data-driven synthesizers so that the joined framework possesses the ability to alleviate data scarcity and domain growth problems while ensuring performance limits and inference speed. The *Colead* framework is proposed to address this challenge, as shown in Fig 2. It has three key components: a retriever, an NLU-driven synthesizer, and a data-driven synthesizer.

3.1 Cycle of Co-evolvement

pThe cycle of co-evolvement is the core of our approach. It starts with the retriever fetching the relevant API documentation based on the user's text query. With the retrieved API documentation and the input query, the NLU-driven synthesizer generates the corresponding code without query-code pairs. Then, the generated code is paired with the query to train the data-driven synthesizer, which in turn shares its well-trained encoder to update the retriever. The updated retriever has a better matching ability and improves the data-driven synthesizer. In this way, the whole process forms a positive circle and co-evolution can be achieved.

The NLU-driven synthesizer, as the teacher of the data-driven synthesizer, reduces the huge demand on human labor to collect query-code pairs, which mitigates the data scarcity issue of DSLs. In addition, the introduced retriever can handle code generation for DSL in domain growth, where the DSL is under active development with frequent updates of new API functions. When new APIs are developed, they only need to be registered in the documentation. New queries from users along with the information retrieved from the updated documentation can be fed into the NLU-driven synthesizer to generate new functional code, which enables the supervised training of the data-driven synthesizer.

3.2 Retriever

The retriever is a shared front end of both the NLU-driven and the data-driven synthesizers. It maps a user's natural language query to the related API description in the documentation.

Specifically, both the natural language query and the API documentation are tokenized by Stanford-CoreNLP (Manning et al., 2014). Initially, exact match is performed based on the description to get the API associated with the query. In other words, the keyword of the query must be present in the API description. Such strict match condition is hard to satisfy, making the retriever fail to retrieve relevant keywords for many queries. To relax the condition and make fuzzy match possible, after the first round of co-evolution, the encoder of the data-driven synthesizer is leveraged to generate dense vector representations of tokens. Two tokens are considered to be matched when the cosine similarity of their vector representations exceeds a pre-defined threshold, which is tuned to have the best result based on experiments.

To realize fuzzy match, the encoder of CodeT5 and SimCSE (Gao et al., 2021) are explored in our

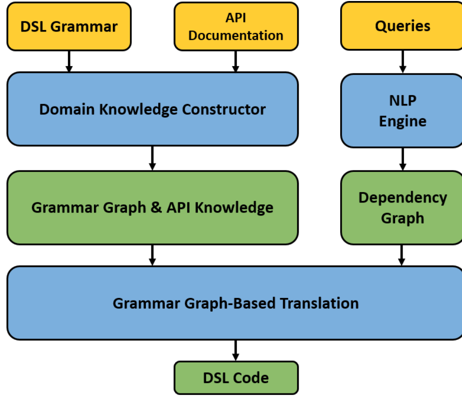

Figure 3: General structure diagram of NLU-driven code generator. It has three main components: a domain knowledge constructor, an NLP engine, and a grammar-graph–based translation module. The grammar graph enables the generated code to follow DSL grammar explicitly. The dependency graph guides the generation of code to follow the logic of the query.

experiment. SimCSE is a simple contrastive learning method that learns embeddings from unlabeled or labeled data. In our experiments, SimCSE is trained only on the queries and API documentation and employed for extracting token embedding.

3.3 NLU-driven Synthesizer

With the retrieved API and user queries, the NLU-driven synthesizer generates the corresponding code by following DSL grammar rules without learning from any specific parallel data.

A typical NLU-driven synthesizer has three components as shown in Fig 3 3: a domain knowledge constructor that processes the domain knowledge to aid code synthesis; an NLP engine that converts an NL-based query to a dependency graph and a grammar-graph–based translation module that generates code based on the dependency graph. The domain knowledge constructor takes two files as input: a document containing all the input and output parameters of the API and their descriptions, and a grammar file containing the context-free grammar written in Backus-Naur form (BNF) (Wikipedia contributors, 2022). The constructor parses the input file and generates two outputs: an API knowledge base for semantic mapping between NL-based queries and APIs, and a grammar graph that defines the search space for code generation. The NLP engine accepts NL-based queries and produces a dependency graph using various NLP techniques, including POS tagging, Lemmatization, NER, and dependency analysis. This dependency graph is sent to the grammar-graph–based translation module for code generation.

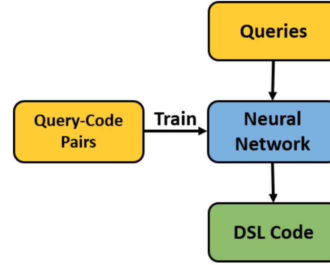

Figure 4: General architecture diagram of a data-driven synthesizer. A data-driven synthesizer can accept natural language queries directly and generate code after training on parallel data. As the training data and model size grow, the performance of the data-driven synthesizer will also progress.

As a "white box" approach, NLU-driven synthesizers are easy to interpret. Synthesis errors can be diagnosed and corrected by humans. However, designing such a synthesizer requires rich expert knowledge. The grammar rules require extensive modifications when adopted to a new program language.

3.4 Data-driven Synthesizer

NLU-driven synthesizers follow grammar rules designed by humans, which cannot cover all situations. The data-driven synthesizer can bridge this gap. It is based on neural networks and outputs the code directly given an input query as Fig 4. The large neural networks have demonstrated impressive success in many NLP tasks, including code generation. We experiment with two pre-trained language models, namely PyCodeGPT (Zan et al., 2022) and CodeT5 (Wang et al., 2021). Our preliminary results show that PyCodeGPT does not perform as well as CodeT5. A possible explanation is that the CodeT5 is an encoder-decoder model, which is more suitable for NL-based code generation. Therefore, we choose CodeT5 as the data-driven synthesizer in our experiments. We also explore CodeT5 in small, base, and large sizes and find that CodeT5-small performed the best. This may be due to the fact that our dataset is small and large models are prone to overfitting.

4 Experiments

Experiments are conducted to answer the following research questions:

- **RQ1:** Can the data-driven synthesizer benefit

DSL	Query	Code
Text Editing	Insert ":" after 1st word.	`INSERT(STRING(:), Position(AFTER(WORDTOKEN()), IntegerSet(INTEGER(1))), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), ALL())))`
ATIS	I would like to find the cheapest flight from Baltimore to Atlanta	`EXTRACT_ROW_MIN_F(COL_FARE()), AtomicRowPredSet( AtomicRowPred(EQ_DEPARTS(CITY(baltimore), ANY(), ANY(), ANY(), ANY()), EQ_ARRIVES(CITY(atlanta), ANY(), ANY(), ANY(), ANY())))`

Table 1: Examples of Text Editing and ATIS. The text editing language was created to allow consumers to use text editing features without knowing how to program. The Air Travel Information System (ATIS) is a DSL for accessing air travel information.

from the NLU-driven synthesizer of *Colead* in the presence of domain growth?

- **RQ2:** Can NLU-driven synthesizers generate training examples of sufficient quality to address data scarcity?
- **RQ3:** Can the *Colead* enable the NLU-driven synthesizer and the data-driven synthesizer to complete their co-evolvement?

4.1 Experimental Setup

The proposed *Colead* is evaluated on two popular datasets that are collected from different DSLs. Notably, due to the difficulty of collecting DSL data, current DSL datasets typically have only a few hundred entries.

- **Text Editing** [1] is a DSL with 52 APIs in total. It is designed for end-users of Office Suite applications to do text editing without the need of understanding the grammar and semantics of regular expressions, conditionals, loops, etc. The dataset for Text Editing consists of 467 query-code pairs.
- **Air Travel Information System (ATIS)** [2] is a DSL that provides support of predicates and expressions for querying air travel information, such as arrival/departure locations, times, dates, prices, etc. It is based on SQL-style operations, with 51 APIs in total. The dataset of ATIS consists of 535 query-code pairs.

Table 1 presents examples of query-code pairs in two DSL datasets. It can be observed that DSL's grammar format is different from GPL, and DSLs are usually single lines of code to accomplish operations. In addition, because DSLs require functions to be streamlined and compressed, the logic for writing is different from GPL. As a result, data-driven synthesizers trained on the GPLs cannot be deployed for these DSLs without fine-tuning.

We propose to use the Exact Match as the measure, i.e., the generated code is considered correct when it is exactly the same as the original code. Although we do not have test cases to test whether the generated code is correct due to the scarcity of DSL data, it is reasonable to use Exact Match because of the small range of variation in DSLs. We follow the dataset setup commonly used in machine learning, with a ratio of 8:2 for training and validation of the dataset.

Table 3 shows the results of the growing ATIS domain.

4.2 Scenario of Domain Growth

Programming languages are non-static. They are constantly growing and being updated with new functionality. Such domain growth is common in the real world, especially for DSLs.

To answer RQ1, we simulate this situation by incrementally adding and updating the APIs in the DSL documentation and adding new query code pairs associated with these new APIs to the training and validation datasets.

We divide the data into three groups, called Original Task (OT), Incompletely New Task (INT), and Completely New Task (CNT), as described in the caption of Table 2. In the original task, the original DSL implemented only basic API functionality. As the language evolves, new high-level APIs are

[1] shorturl.at/npFIS
[2] shorturl.at/sxyS5

	Stage 1			Stage 2			Stage 3		
	ot	int	cnt	ot	int	cnt	ot	int	cnt
Golden Data		OT			OT+INT			OT+CNT	
	82.54	00.00	00.00	86.13	80.00	00.00	88.88	00.00	80.00
Baseline		OT			OT+INT			OT+CNT	
		-		84.12	43.33	00.00	77.77	00.00	43.33
Ours ($Colead$)		OT			OT+INT			OT+CNT	
		-		85.71	50.00	00.00	85.71	00.00	50.00

Table 2: We train CodeT5 (Wang et al., 2021) on the growing TextEditing domain and use the Exact Match results. i) OT and ot, INT and int, CNT and cnt represent the training and validation set of Original Task(OT), Incompletely New Task(INT) and Complete New Task(CNT). ii) Markers without underlines mean that they belong to the golden dataset. Markers with underlines mean that the code for this dataset is generated from HISyn (Nan et al., 2020). Markers with double underlines mean that the data pairs in this dataset are 5 samples drawn from the correct dataset. iii) In Stage 1, only the Original Task is the subject of experiments. Stage 2 involves both Original Task and Incompletely New Task. In Stage 3, both Original Task and Complete New Task are the subjects of experiments.

	Stage 1			Stage 2			Stage 3		
	ot	int	cnt	ot	int	cnt	ot	int	cnt
Golden Data		OT			OT+INT			OT+CNT	
	79.76	00.00	00.00	96.42	70.83	00.00	95.00	00.00	00.00
Baseline		OT			OT+INT			OT+CNT	
		-		97.61	45.83	00.00	97.38	00.00	56.52
Ours ($Colead$)		OT			OT+INT			OT+CNT	
		-		97.62	25.00	00.00	97.61	00.00	17.39

Table 3: We also train CodeT5 (Wang et al., 2021) on the growing ATIS domain and use the same format as Table 2.

added. They may be incomplete and only partially functional. We refer to the data of these new APIs as Incompletely New Task. Finally, new APIs are developed and completed, bringing data for Completely New Task. For example, in the Original Task in the text editing domain, there is no REMOVE API. As shown in Table 4, the Incompletely New Task introduces the REMOVE API with limited functionality. In the Complete New Task, the IterationScope parameter is added to set the iteration scope to make REMOVE API complete.

Experiments are divided into three stages. In Stage 1, experiments are carried out on Original Task only. In Stage 2, Original Task and Incompletely New Task are included. And in Stage 3, Original Task and Completely New Task are included. We use three different data settings to train CodeT5 (Wang et al., 2021). The results are shown in Table 2. HISyn (Nan et al., 2020) tends to generate multiple code candidates because it is common for HISyn to construct graphs and traverse them to find several paths that satisfy its requirements. We choose to construct the dataset by selecting only the shortest of all candidates, as the results show that it is better than selecting all candidates. Al-

Incompletely New Task (INT)

```
REMOVE(SelectString(NUMBERTOKEN(),
BConditionOccurrence(
BETWEENCOND(STRING(colon),
STRING(colon),IMM()),ALL())))
```

Complete New Task (CNT)

```
REMOVE(SelectString(NUMBERTOKEN(),
BConditionOccurrence(
BETWEENCOND(STRING(colon),
STRING(colon), IMM()), ALL())),
IterationScope(LINESCOPE(),
BConditionOccurrence(ALWAYS(), ALL())))
```

Table 4: Examples of Incompletely New Task (INT) and Complete New Task (CNT). Notably, Original Task (OT) has no new task. Thus, we omit its table illustration for brevity. Compared with INT, the IterationScope is added to the REMOVE API to enable it to set the iteration scope in CNT.

though selecting all candidates has a larger quantity, a number of incorrect codes introduce noise and reduce data quality. The row "Golden" implies the accuracy that CodeT5 can achieve with exactly the right data. This is the best result we can expect from our strategy. We want the exact matching accuracy of the “Ours" row to be as close as possible to the "Golden" row. Similar results are obtained for ATIS, shown in Table 3.

Stage 1: This stage represents the early development of the DSL, where new tasks have not yet been introduced. At this stage, the training set has only examples of the Original Task. CodeT5 trained on data from the Original Task performs well on the corresponding validation set but has an accuracy of 0 on the validation set of the new task. Although these validation sets belong to the same domain, CodeT5 cannot handle the new API. This indicates that the model trained with the old data cannot handle the new API.

Stage 2: This stage represents a DSL in an intermediate stage, where a new task has been introduced, but it is still incomplete and simple. In order to generate the code for the new task, we need to supplement the corresponding API grammar and description. CodeT5 trained with the data generated by our method accomplishes the same adaptation to the Incompletely New Task as CodeT5 trained with the complete data. By adding the new task-related APIs to the documentation, our method is able to generate data that can be used for training in this stage.

Stage 3: This stage represents a DSL in its final stage, where a new task has been introduced and its development has been completed. Our method requires that both the corresponding grammar and the description in the API documentation be updated to be considered complete. Training data containing complete query-code pairs allows CodeT5 to learn the complete API, completing the migration from Incompletely New Task to Completely New Task. The data generated by our method accomplishes this as well, proving that it not only works on new tasks but can be applied to task updates as well.

From the above three stages, we can observe that our method can consistently provide data to CodeT5 for training in domain growth.

4.3 Scenario of Data Scarcity

Data scarcity occurs frequently in DSLs, as a number of DSLs are designed for specific applications. **To answer RQ2**, we analyze the performance of our method in solving the data scarcity problem. To establish a point of comparison, we set a baseline by including five relevant examples from the new task (specifically, the Incomplete New Task in Stage 2 and the Complete New Task in Stage 3) in the training set. This will be used as a baseline for evaluating the performance of our method.

The results are summarized in Table 2, from which it can be observed that our method outperforms the baseline in all stages. Compared to CodeT5 trained with golden data in Stage 1, the evaluation results of the baseline and $Colead$ on Original Task in Stage 2 increase by 1.58% and 3.17% respectively. This indicates that the commonality of the new task with the old task allows the introduction of new task data to enhance the model’s ability to handle the old task. The change in the evaluation results for Incomplete New Task is more pronounced, with the baseline and $Colead$ increasing from 0 to 43.33% and 50.00%, respectively. In Stage 3, the same changes continue to appear for the Completely New Task. To sum up, the above results show that our method can have better help than providing a few positive samples.

4.4 Study of Co-evolvement

To answer RQ3, we experimentally verify the effectiveness of $Colead$ co-evolving on both NLU-driven synthesizers and data-driven synthesizers.

We observe the performance changes of the two synthesizers throughout their co-evolvement. For CodeT5, we use a setup similar to the third stage in Section 4.2, i.e., using the code generated by HISyn for training. For HISyn, since it does not require training, we use the entire dataset for testing. To highlight the role of the retriever, we use documents whose descriptions have not been manually optimized, which places a higher demand on the matching ability of the retriever. We introduce SimCSE (Gao et al., 2021) as the baseline for comparison. The loop starts with an exact match retriever, which is then replaced by a trained CodeT5 encoder. From Table 5, we can observe that $Colead$ leads consistent improvements to CodeT5 during the loop. The "Original" row represents the results of CodeT5 trained on the data generated by HISyn using the exact match retriever.

		CodeT5		HISyn	
		TextEditing	ATIS	TextEditing	ATIS
Baselines	SimCSE	59.59	62.62	22.79	8.95
	Original	56.99	62.62	23.48	10.48
Ours(*Colead*)	Loop 1	58.06	63.55	23.44	10.37
	Loop 2	59.14	**65.42**	24.27	**10.63**
	Loop 3	**60.22**	61.68	**24.42**	10.37

Table 5: Exact Match accuracy of CodeT5 in the loops of *Colead* is on the left. Exact Match accuracy of HISyn with different retrievers in the loops of *Colead* is on the right. The first four lines belong to the Co-evolvement loop of *Colead*. SimCSE is independent of the circle and is not part of it. The bolded numbers are the best results.

Loop 1: From the original to Loop 1, the Exact Match of CodeT5 increases from 56.99% to 58.06% on the TextEditing dataset and from 62.62% to 63.55% on the ATIS dataset, which is contributed by replacing the exact match retriever with the encoder of the CodeT5. Meanwhile, the Exact Match of HISyn slightly decreases but it exceeds the Original later.

Loop 2: CodeT5 continues to improve its performance by 1.08% and 1.87% on the TextEditing and ATIS datasets. We can observe that while SimCSE can outperform Original and Loop 1, *Colead* outperforms SimCSE after Loop 2. HISyn continues to improve by 0.83% and 0.26% on the TextEditing and ATIS datasets, and it outperforms Original and SimCSE.

Loop 3: In Loop 3, CodeT5 and HISyn perform best among these cases on the TextEditing dataset. However, it is observed that the accuracy of ATIS decreased, which is unexpected. By diagnosing the grammar and dependency graphs, we are able to determine the root cause of the errors. Our analysis shows that the fuzzy match retriever increases the number of candidate APIs for mapping. The expansion of the range of candidate APIs exceeds the capability of HISyn, resulting in a degradation of the quality of the generated code.

We analyzed how *Colead* can improve NLU-driven synthesizers with reference to specific examples. Table 6 shows a specific case where the original HISyn generation failed but succeeded in *Colead*. The original HISyn generation fails because the semantic mapping in the retriever is not sufficient to handle the ambiguity of natural language. The updated retriever in *Colead* compensates for this shortcoming. We know from the grammar graph construction log that the original HISyn does not map the word "Remove" to the REMOVE API correctly but the wrong INSERT API. The reason is that the description in the API documentation does not explicitly include "Remove". The fuzzy match retriever is able to correctly match the corresponding API and therefore get the correct code.

Query	Remove colon before every line
Original	`INSERT(STRING(colon), Position(BEFORE(LINETOKEN()),ALL()), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(),ALL())))`
Colead	`REMOVE(SelectString(STRING(colon), BConditionOccurrence( BEFORECOND(LINETOKEN(),IMM()),ALL())), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(),ALL())))`
Query	Remove colon before every line
Original	`INSERT(STRING(colon), Position(BEFORE(LINETOKEN()),ALL()), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(),ALL())))`
Colead	`REMOVE( SelectString(STRING(colon), BConditionOccurrence( BEFORECOND(LINETOKEN(),IMM()),ALL())), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(),ALL())))`

Table 6: Comparison between the original HISyn and *Colead* generated code. In the example of the table, the original is wrong and *Colead* is correct. The original HISyn make a wrong decision for the query because it could not map near-synonyms.

From Table 5, we find that the performance trend of HISyn is similar to that of CodeT5. Although occasionally decreasing at the same time, they consistently improve with the cycle, which proves that they are co-evolving.

5 Discussion

We try ChatGPT (Ouyang et al., 2022a) to accomplish our task and compare it to our approach. To ensure that ChatGPT has the same data situation

Query	Insert colon before every 1st word
Ours	`INSERT(STRING(colon), Position(BEFORE(WORDTOKEN()), IntegerSet(INTEGER(1))), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), ALL())))`
0-shot ChatGPT	insert ":", start all
5-shot ChatGPT	`INSERT STRING(colon) Position(BEFORE(1st()), ALL()) IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), ALL()))`

Table 7: Comparison between ChatGPT and *Colead* generated code. In the example of the table, 0-shot ChatGPT is completely wrong and *Colead* is correct. But 5-shot ChatGPT performs well.

as our approach, we feed the entire grammar and documentation into ChatGPT. We use the prompt "Based on the above grammar and documentation, write DSL code to accomplish my query" to instruct ChatGPT and prepend 5 query-code pairs in the 5-shot setting.

In Table 7, we observed a difference in performance between ChatGPT and *Colead*. While *Colead* can output exactly the right answer, the performance of 0-shot ChatGPT and 5-shot ChatGPT is very different.

ChatGPT does not require any additional training. It utilizes contextual information to provide inferences, which demonstrates its versatility and adaptability to various tasks. However, if no sample is provided, ChatGPT would not perform well. 0-shot ChatGPT partially understands the request and outputs plausible code. 5-shot ChatGPT generates an almost correct answer, but the grammar is not standardized enough. While correcting it once could easily solve the problem, this step alone incurs significant labor costs, whereas *Colead* is generated strictly according to grammatical rules, and thus has a much smaller probability of grammatical errors.

6 Conclusion and Future Work

Domain growth and data scarcity are two challenges that hinder the application of code generation to DSLs. We have shown that our proposed *Colead* framework can effectively mitigate these problems. Our framework combines NLU-driven and data-driven synthesizers, where the NLU-driven synthesizer alleviates the data-hungry issue of the data-driven one and the data-driven synthesizer provides better semantic mapping for the NLU-driven synthesizer to improve code quality.

Future work should consider code generation by leveraging grammar rules to regularize language models. Components in the NLU-driven synthesizer can be further improved, e.g., semantic mapping, NLP engines, etc. More powerful language models, such as GPT3 (Brown et al., 2020; Ouyang et al., 2022b), can be leveraged to improve the data-driven synthesizer.

Limitations

Number of datasets. Due to the limited number of publicly available DSL datasets, the authors evaluate their method on two DSL datasets. In order to fully validate the capability of the proposed method, it would be desirable to collect more real-world datasets. One potential approach is to manually collect DSL data for a domain, however, this would be costly. Another approach is to apply active learning methods (Ren et al., 2021) to automatically identify relevant datasets as alternative DSL datasets.

Ethics Statement

The authors declare that they adhere to general ethical principles, professional responsibility, principles of professional leadership, and ethical guidelines. In studies involving human participants, all procedures were in accordance with ACL's ethics policy.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In NAACL-HLT, pages 2655–2668. Association for Computational Linguistics.

Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. IEEE Access, 7:86121–86144.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. Autopandas: neural-backed generators for program synthesis. Proc. ACM Program. Lang., 3(OOPSLA):168:1–168:27.

Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In

Proceedings of the eleventh annual conference on Computational learning theory, pages 92–100.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In NeurIPS.

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.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In ICSE, pages 345–356. ACM.

Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 516–527.

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

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In Empirical Methods in Natural Language Processing (EMNLP).

Shlok Gilda. 2017. Source code classification using neural networks. In 2017 14th international joint conference on computer science and software engineering (JCSSE), pages 1–6. IEEE.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. CoRR, abs/1605.08535.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In ICLR. OpenReview.net.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In ICML, volume 119 of Proceedings of Machine Learning Research, pages 5110–5121. PMLR.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In Advances in Neural Information Processing Systems.

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: synthesizing smartphone automation scripts from natural language. In MobiSys, pages 193–206. ACM.

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.

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 NeurIPS Datasets and Benchmarks.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In ACL (System Demonstrations), pages 55–60. The Association for Computer Linguistics.

Zifan Nan, Hui Guan, and Xipeng Shen. 2020. Hisyn: human learning-inspired natural language programming. In ESEC/SIGSOFT FSE, pages 75–86. ACM.

Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. 2021. Deep nlp-based co-evolvement for synthesizing code analysis from natural language. In CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021, pages 141–152. ACM.

Zifan Nan, Xipeng Shen, and Hui Guan. 2022. Enabling near real-time nlu-driven natural language programming through dynamic grammar graph-based translation. In IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022, pages 278–289. IEEE.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. CoRR, abs/2203.13474.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022a. Training language models to follow instructions with human feedback.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022b. Training language models to follow instructions with human feedback. CoRR, abs/2203.02155.

Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. CoRR, abs/1802.04335.

Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. ACM computing surveys (CSUR), 54(9):1–40.

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 EMNLP (1), pages 8696–8708. Association for Computational Linguistics.

Wikipedia contributors. 2022. Backus–naur form — Wikipedia, the free encyclopedia. [Online; accessed 18-January-2023].

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022a. A systematic evaluation of large language models of code. In MAPS@PLDI, pages 1–10. ACM.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022b. In-ide code generation from natural language: Promise and challenges. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(2):1–47.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. Proceedings of the ACM on Programming Languages, 1(OOPSLA):1–26.

Mitchell Young, Zifan Nan, and Xipeng Shen. 2022. IDE augmented with human-learning inspired natural language programming. In 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022, pages 110–114. ACM/IEEE.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: continual pre-training on sketches for library-oriented code generation. In IJCAI, pages 2369–2375. ijcai.org.