To See is Not to Master: Teaching LLMs to Use Private Libraries for Code Generation

Yitong Zhang*
zhangyt42@buaa.edu.cn
College of AI, Tsinghua University
Beijing, China
Proxseer Inc.
California, USA

Chengze Li*
231220004@smail.nju.edu.cn
School of Computer Science, Nanjing University
Nanjing, China

Ruize Chen*
231250003@smail.nju.edu.cn
Software Institute, Nanjing University
Nanjing, China

Guowei Yang
gavin@eniacode.com
Proxseer Inc.
California, USA

Xiaoran Jia
jiaxiaoran576@gmail.com
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China

Yijie Ren
23373276@buaa.edu.cn
School of Computer Science and Engineering, Beihang University
Beijing, China

Jia Li†
jia_li@mail.tsinghua.edu.cn
College of AI, Tsinghua University
Beijing, China

Abstract

Large Language Models (LLMs) have shown strong potential for code generation, yet they remain limited in private-library-oriented code generation, where the goal is to generate code using APIs from private libraries. Existing approaches mainly rely on retrieving private-library API documentation and injecting relevant knowledge into the context at inference time. However, our study shows that this is insufficient: *even given accurate required knowledge, LLMs still struggle to invoke private-library APIs effectively.*

To address this limitation, we propose *PriCoder*, an approach that teaches LLMs to *invoke* private-library APIs through automatically synthesized data. Specifically, *PriCoder* models private-library data synthesis as the construction of a graph, and alternates between two graph operators: ❶ *Progressive Graph Evolution*, which improves data diversity by progressively synthesizing more diverse training samples from basic ones, and ❷ *Multidimensional Graph Pruning*, which improves data quality through a rigorous filtering pipeline. To support rigorous evaluation, we construct two new benchmarks based on recently released libraries that are unfamiliar to the tested models. Experiments on three mainstream LLMs show that *PriCoder* substantially improves private-library-oriented code generation, yielding gains of over 20% in *pass@1* in many settings, while causing negligible impact on general code generation capability. Our code and benchmarks are publicly available at https://github.com/eniacode/PriCoder.

CCS Concepts

• **Software and its engineering** → **Software libraries and repositories**; **Automatic programming**; • **Computing methodologies** → **Neural networks**; **Natural language processing**.

Keywords

Private-Library-Oriented Code Generation, Large Langugae Models

ACM Reference Format:
Yitong Zhang, Chengze Li, Ruize Chen, Guowei Yang, Xiaoran Jia, Yijie Ren, and Jia Li. 2018. To See is Not to Master: Teaching LLMs to Use Private Libraries for Code Generation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Recently, Large Language Models (LLMs) have shown strong potential for code generation [3, 15, 19, 20]. However, many real-world development scenarios rely heavily on internal private libraries, which are rarely included in public training corpora [41, 48]. This gives rise to an important yet challenging task, namely *Private-Library-Oriented Code Generation*, which aims to automatically generate code that invokes private-library APIs to satisfy specific coding requirements. Since LLMs typically lack prior knowledge of such libraries, they often struggle to use private libraries effectively, which largely restricts the practical impact of LLMs in real-world software development.

*Equal contribution. ❶ Yitong Zhang and Chengze Li jointly conceived the idea. ❷ Yitong Zhang took the lead in writing the paper. ❸ Chengze Li implemented the proposed method. ❹ Ruize Chen contributed to baseline reproduction.
†Corresponding author.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

To mitigate this deficiency, a growing body of work has explored private-library-oriented code generation through Retrieval-Augmented Generation (RAG) [23, 41, 48, 52]. These approaches typically retrieve relevant API specifications from documentation and inject them into the context. They implicitly assume that, once the model is presented with the required knowledge, it can reliably invoke the required APIs to satisfy the coding requirement. However, we notice that a fundamental question remains underexplored: ***given the required API knowledge, can LLMs actually utilize it effectively?*** To answer this question, we conduct an empirical study in Section 3. Our results show that even when provided with detailed specifications of all required APIs, LLMs still struggle to invoke these APIs effectively to satisfy the coding requirements. For example, even when equipped with the complete set of required private APIs, the *pass@1* of LLaMA3.1-8B-Instruct [8] improves only from 8.13% to 13.10%, which remains far from practical usability. Moreover, simply scaling up model size does not fundamentally resolve this issue: when given the same complete API knowledge, LLaMA3.1-70B-Instruct [8] improves by only 8.32% in *pass@1*. These findings suggest that the key bottleneck is not merely whether the model can *see* the right API knowledge before generation, but whether it can *invoke* private-library APIs correctly during generation. Therefore, in this paper, different from most prior work [23, 30, 52], we focus on how to improve LLMs' ability to effectively *invoke* private-library APIs.

One fundamental reason why LLMs perform poorly on private libraries is that such libraries are typically absent from the training corpora [48]. This naturally suggests training LLMs on data related to the target private library. A natural next question, then, is how to obtain training data about private libraries at scale. Due to the closed-source nature of private libraries, real-world code that invokes them is often scarce, while manually curating such data is labor-intensive and impractical [23, 31]. We therefore turn to LLM-driven data synthesis, where LLMs automatically generate training samples that are subsequently used to fine-tune themself [27, 29, 42, 43]. However, direct synthesis is far from trivial, because the models initially lack sufficient prior knowledge of the target private library. As a result, directly synthesized data often suffers from two major challenges. ❶ ***Low Data Diversity.*** LLMs tend to generate overly basic requirements that can be solved with only a small number of private API. In contrast, real-world development scenarios often require the coordinated invocation of multiple private APIs to satisfy diverse and complex coding requirements. This creates a substantial gap between synthesized data and practical development needs. ❷ ***Poor Data Quality.*** LLMs are prone to producing flawed training samples, such as those containing syntax errors, non-existent APIs, or semantically incorrect API invocations. Directly training on such noisy data can be counterproductive and may even degrade the model's overall capabilities.

To address these challenges, we propose *PriCoder*, an approach that teaches LLMs to *invoke* private-library APIs. The main idea of *PriCoder* is to let LLMs automatically synthesize training samples tailored to the target private library and then learn private-library API invocation through training. Specifically, we model this data synthesis process as the construction of a graph. Starting from private API specifications, *PriCoder* relies on two key graph operators to grow and refine this graph: ❶ First, ***Progressive Graph Evolution*** improves data diversity by progressively synthesizing new, diverse sample nodes based on existing basic ones. This enables the graph to explore a much larger API composition space, thereby better covering real-world development scenarios. ❷ Second, ***Multidimensional Graph Pruning*** ensures data quality by removing low-quality sample nodes through a multidimensional verification pipeline: we (i) eliminate samples with syntactic errors, (ii) validate executability via automatically generated tests, and (iii) further assess overall functionality automatically. By alternately evolving and pruning this graph, *PriCoder* constructs a high-diversity and high-quality synthetic dataset, enabling LLMs to effectively master private-library API invocation without human intervention.

Evaluating private-library-oriented code generation requires libraries that are unseen to the model. However, many existing benchmarks are built on libraries that have been publicly released for a long time, making them unreliable for assessing the private-library-oriented generation capability of today's popular LLMs [41, 48]. To obtain a more faithful evaluation, we select two recently released libraries, `ndonnx` [37] and `numba-cuda` [35], both of which were introduced in 2024 and continued to evolve throughout 2025. Based on them, we carefully construct two new benchmarks, namely `NdonnxEval` and `NumbaEval`, which contain 169 and 187 instances, respectively. We apply *PriCoder* to three mainstream LLMs. Our results show that *PriCoder* substantially improves LLMs' private-library-oriented code generation capability, yielding gains of more than 20% in *pass@1* in many settings. Additionally, these gains come with negligible impact on the models' general capabilities. Ablation studies further confirm that both Progressive Graph Evolution and Multidimensional Graph Pruning are essential to the overall effectiveness of our approach.

In summary, our contributions are threefold:

- We show that LLMs struggle to invoke private-library APIs for code generation, even when the required API knowledge is provided in the context.
- We propose *PriCoder*, which can enable LLMs to learn private-library knowledge automatically and improve their ability to invoke private-library APIs.
- We carefully construct and open-source two novel benchmarks to support rigorous evaluation. Extensive experiments demonstrate the effectiveness of *PriCoder*.

2 Background and Related Work

2.1 Large Language Models for Code Generation

Large language models (LLMs) have recently emerged as a powerful paradigm for code generation, substantially improving the automation of software development [21, 22, 36, 45, 50]. Recent model families, such as Qwen [13, 44], DeepSeek [10, 24], LLaMA [8, 38], and GPT [14, 39], have demonstrated strong performance on a wide range of code generation benchmarks, suggesting that modern LLMs can effectively capture common programming patterns, language syntax, and widely used library knowledge. However, these capabilities are still fundamentally bounded by the knowledge acquired during training [40]. Once training is completed, the model's internal knowledge becomes fixed, making it difficult to reliably solve tasks that depend on non-public knowledge [16, 34, 46].

Table 1: *pass@k* (%) on `NumbaEval` and `NdonnxEval`.

Method	NumbaEval			NdonnxEval		
	pass@1	*pass@3*	*pass@5*	*pass@1*	*pass@3*	*pass@5*
`DeepSeek-Coder-6.7B-Instruct`						
Vanilla	10.59	21.26	27.82	20.36	29.16	33.26
Oracle	17.97	34.23	42.45	38.05	50.82	55.86
Gain (↑)	**+7.38**	**+12.97**	**+14.63**	**+17.69**	**+21.66**	**+22.60**
`LLaMa3.1-8B-Instruct`						
Vanilla	8.13	16.97	21.73	15.86	23.50	26.55
Oracle	13.10	26.73	33.66	30.36	42.79	47.45
Gain (↑)	**+4.97**	**+9.76**	**+11.93**	**+14.50**	**+19.29**	**+20.90**
`LLaMa3.1-70B-Instruct`						
Vanilla	35.88	52.41	59.28	27.99	36.07	39.34
Oracle	44.20	57.16	62.08	48.60	58.83	60.29
Gain (↑)	**+8.32**	**+4.75**	**+2.80**	**+20.61**	**+22.76**	**+20.94**

2.2 Private-Library-Oriented Code Generation

Private-Library-Oriented Code Generation refers to generating code that leverages private-library APIs to satisfy specific coding requirements [48]. Unlike conventional library-oriented code generation [9, 25, 26, 49], models typically have little to no prior knowledge of the target library, where such private libraries are widely used within companies but are rarely included in public training corpora. To enable models to utilize private libraries effectively, a growing body of research has been proposed. For example, APIFinder [48] and DocPrompting [52] adopt straightforward Retrieval-Augmented Generation (RAG) [18] to retrieve relevant APIs and include their information in the prompt. EpiGEN [23] and CAPIR [30] improve retrieval accuracy by decomposing coding requirements into some subtasks before retrieving API information. ExploraCoder [41] further incorporates execution feedback to mitigate issues caused by ambiguity in API documentation. Overall, most existing approaches rely on RAG to inject more accurate API knowledge into the context, so that the model can ***see*** sufficient information at inference time, while paying less attention to teaching the model to ***invoke*** private-library APIs effectively.

3 Motivation

Most prior work on private-library-oriented code generation is based on RAG. They implicitly assume that once the model is presented with the required API knowledge, it can reliably invoke required APIs to satisfy the coding requirement. In this section, we challenge this assumption and show that *seeing* the right API knowledge does not necessarily translate into *invoking* it correctly.

To disentangle the impact of retrieval quality, we design an oracle setting that eliminates potential retrieval errors. Specifically, for each problem, we construct an *oracle* prompt that includes the full specifications of all APIs required to solve the task, including their signatures and clear functional descriptions. This oracle knowledge represents an idealized upper bound for existing RAG-based approaches. We compare it with a *vanilla* setting, which solves the same requirement but does not provide the model with any API documentation. Experiments are conducted on two benchmarks, `NumbaEval` and `NdonnxEval` (described in Section 5), using DeepSeek-Coder-6.7B-Instruct [10], LLaMa3.1-8B-Instruct [8], and LLaMa3.1-70B-Instruct [8].

Case 1:

Input:

Goal: *Apply a scalar gain to every element of an array, returning a scaled output with the same shape.*

Required API Knowledge (Oracle) :

- ***ndonnx.asarray****(obj, *, dtype=None, device=None, copy=None) -> Array:*
  Converts an array-like object (e.g., NumPy array, scalar, nested sequence) into an ***ndonnx.Array****.*
- *ndonnx.multiply(x1, x2) -> Array:*
  Performs elementwise multiplication between two inputs and returns an ndonnx.Array. ***At least one input must be an ndonnx.Array****.*

Output:

```
1:  def apply_gain(x: Array, gain: float)
2:      return ndonnx.multiply(x, gain)
```

Error:

At least one argument to 'multiply()' is of type 'ndonnx.Array'.

Case 2:

Input:

Goal: *Compute the elementwise ratio between two arrays, with an option to replace divisions by zero with a specified fallback value.*

Required API Knowledge (Oracle) :

- *ndonnx.where(****cond, a, b****) -> Array:*
  Elementwise selection: returns a where cond is true, otherwise b ...
- *...*

Output:

```
1:  def safe_ratio(x: Array, y: Array, fallback=0.0):
2:      x = ndonnx.asarray(x)
3:      y = ndonnx.asarray(y)
4:      quotient = ndonnx.divide(x, y)
5:      zero_denominator = ndonnx.where(y == 0)
6:      quotient = ndonnx.where(zero_denominator, fallback, quotient)
7:      return quotient
```

Error:

'where()' missing 2 required positional arguments: 'a' and 'b'.

Figure 1: Two cases on `NdonnxEval` with Deepseek-Coder-6.7B-Instruct. Despite having access to oracle required API knowledge, the model fails to invoke these APIs effectively.

Table 1 reports *pass@k* on `NumbaEval` and `NdonnxEval`. Although providing perfect API specifications improves performance, the absolute gains remain far from sufficient for practical deployment. For instance, on `NumbaEval`, *oracle* prompting only increases *pass@1* from 10.59% to 17.97% for DeepSeek-Coder-6.7B-Instruct, and from 8.13% to 13.10% for LLaMa3.1-8B-Instruct, leaving most instances unsolved. Furthermore, even when scaling up to the substantially larger LLaMa3.1-70B-Instruct, the *pass@1* score also experiences a modest increase from 35.88% to 44.20%, indicating that simply increasing model parameters cannot overcome this fundamental bottleneck. Overall, these results suggest that even when the required APIs are explicitly provided in the context, LLMs still struggle to invoke them effectively.

We further analyze the failed cases and find that errors are largely attributable to *ineffective invocation* of the provided oracle APIs, rather than missing any required knowledge in the context. In many instances, the model either does not invoke a necessary API even when it is explicitly available, or invokes it in an incorrect manner. We highlight two representative cases in Figure 1. ① In the first case, the task requires converting an array into an `ndonnx.Array` via `ndonnx.asarray` before applying `ndonnx.multiply`. Despite being provided with both API specifications, the model directly calls `ndonnx.multiply` on the raw input and omits the required

conversion, which triggers a type error at runtime. ② In the second case, the task requires safely handling division-by-zero by using `ndonnx.where` to select between the computed quotient and a fallback value. Although the model attempts to use `ndonnx.where`, it misinterprets the API signature and calls `ndonnx.where` with only one argument, leading to an argument error.

Motivation 1: Even with oracle API knowledge, LLMs frequently fail to invoke APIs actively and correctly. The key bottleneck is enabling LLMs to *invoke* private-library APIs effectively, rather than merely *seeing* required API knowledge in context.

Motivated by the above findings, we argue that LLMs should be trained to learn how to invoke private-library APIs. However, training data for private libraries is highly scarce. We therefore consider a natural direction: synthesizing training data with LLMs [1, 31]. However, naively prompting LLMs to directly generate such training data about private libraries faces some critical challenges. To illustrate this, we conduct an additional study on ndonnx, a library that is unfamiliar to all evaluated models in this paper. Specifically, we inject the complete API documentation of the library into the prompt and ask DeepSeek-Coder-6.7B-Instruct to directly synthesize 200 training instances, each consisting of a coding requirement and its corresponding solution.

Our analysis reveals two major challenges that prevent such naively synthesized data from being directly used for training: ❶ ***Low Data Diversity.*** We find that 84% of the synthesized samples invoke fewer than four APIs. This suggests that when LLMs are unfamiliar with the target private library, they tend to generate overly basic requirements, with many samples relying on only a small set of APIs. In contrast, real-world private-library-oriented development involves substantially more diverse and multiple API compositions. As a result, naively synthesized data exhibits limited diversity and fails to provide broad coverage of realistic development needs. ❷ ***Poor Data Quality.*** Through manual inspection, we find that 57% of the synthesized samples contain obvious runtime errors, often caused by invoking non-existent APIs or using existing APIs incorrectly. More importantly, this only reflects failures that are immediately observable at execution time. Many additional samples may still suffer from functional errors even if they can run successfully. Such low-quality data is unreliable for training and may even be harmful if used directly for fine-tuning.

Motivation 2: Directly prompting LLMs to synthesize training data for private libraries faces two major challenges, namely *Low Data Diversity* and *Poor Data Quality*.

Building on the above motivation, we propose *Progressive Graph Evolution* and *Multidimensional Graph Pruning* in the next section, to automatically synthesize high-diversity and high-quality training data for private-library-oriented code generation.

4 Methodology

In this section, we present PriCoder, an approach designed to teach LLMs to *invoke* private libraries effectively.

Algorithm 1 Overall Procedure of PriCoder

Input: Private-library API specifications $\mathcal{S}$, LLM p_θ, Target dataset size N
Output: Fine-tuned LLM $p_{\hat{\theta}}$

procedure PriCoder($\mathcal{S}, p_\theta, N$)
  Initialize synthesis graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} \leftarrow \mathcal{S}, \mathcal{E} \leftarrow \emptyset$
  while $|\mathcal{V} \setminus \mathcal{S}| < N$ **do**
    $\mathcal{G}, v_{\text{new}} \leftarrow$ Evolution($\mathcal{G}, \mathcal{S}$) ▷ *Progressive Graph Evolution*
    $\mathcal{G} \leftarrow$ Pruning($\mathcal{G}, v_{\text{new}}$) ▷ *Multidimensional Graph Pruning*
  end while
  $\mathcal{D}_{\text{syn}} \leftarrow \{(r, y) \mid (r, y) \in \mathcal{V} \setminus \mathcal{S}\}$
  Fine-tune p_θ on $\mathcal{D}_{\text{syn}}$ optimizing Eq. 1 to obtain $p_{\hat{\theta}}$ ▷ *Training*
  return $p_{\hat{\theta}}$
end procedure

4.1 Overview

At a high level, PriCoder first synthesizes training data tailored to the target private library and then fine-tunes the model on the synthesized data. The overall procedure of this approach is detailed in Algorithm 1. PriCoder models private-library data synthesis as the construction of a ***synthesis graph***. Formally, let $\mathcal{L}$ denote a private library with API set $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ and corresponding specifications $\mathcal{S} = \{s(a) \mid a \in \mathcal{A}\}$, where $s(a)$ includes the signature and functional description of API a. Based on $\mathcal{L}$, we construct a synthesis graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, whose nodes consist of ***API nodes*** and ***sample nodes***. Each API node corresponds to one private-library API specification $s(a)$, while each sample node corresponds to one synthesized training sample (r, y), where r is a coding requirement and y is its reference solution. An edge $(u, v) \in \mathcal{E}$ indicates that node v is synthesized based on node u. Since each new sample node is generated only from pre-existing nodes, $\mathcal{G}$ is naturally a Directed Acyclic Graph (DAG).

Starting from API nodes, PriCoder builds $\mathcal{G}$ into a larger graph step by step. As discussed in Section 3, naively synthesized data often suffers from low diversity and poor quality. To address this, PriCoder introduces two graph operators. First, ***Progressive Graph Evolution*** (Section 4.2) expands the graph by synthesizing new sample nodes based on existing nodes, enabling the graph to grow from API-level knowledge to increasingly diverse training samples. Second, ***Multidimensional Graph Pruning*** (Section 4.3) removes low-quality sample nodes through automatic verification, ensuring that only reliable nodes remain in the graph. By alternately evolving and pruning the graph, PriCoder automatically constructs a high-diversity and high-quality synthetic dataset $\mathcal{D}_{\text{syn}} = \{(r_i, y_i)\}_{i=1}^{N}$ from the retained sample nodes. Figure 2 provides an overview of PriCoder.

We then use $\mathcal{D}_{\text{syn}}$ to fine-tune a model p_θ for private-library-oriented code generation (Section 4.4). Given a coding requirement r, the model is trained to generate a functionally correct solution y by optimizing the standard maximum-likelihood objective:

$$\min_{\theta} \mathcal{L}(\theta) = -\sum_{(r,y)\in\mathcal{D}_{\text{syn}}} \log p_\theta(y \mid r). \tag{1}$$

4.2 Progressive Graph Evolution

To address the low-diversity challenge identified in Section 3, we introduce *Progressive Graph Evolution* (Algorithm 2), a graph operator that progressively grows the synthesis graph from basic nodes to increasingly diverse ones. Specifically, we first evolve the graph from API nodes into initial sample nodes, and then iteratively expand it by synthesizing new sample nodes based on existing sample nodes.

Algorithm 2 Progressive Graph Evolution

Input: Current graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, API specifications $\mathcal{S}$
Output: Updated graph $\mathcal{G}$ and the newly added sample node v_{new}

procedure EVOLUTION($\mathcal{G}, \mathcal{S}$)
  if $\mathcal{G}$ lacks sufficient initial sample nodes **then** ▷ *Graph Initial Evolution*
    Sample $\mathcal{S}_{\text{api}} \subset \mathcal{S}$
    $(r, y) \leftarrow \text{Evolve}(\mathcal{S}_{\text{api}}, \mathcal{I}_{\text{init}})$
    $\mathcal{V}_{\text{parents}} \leftarrow \mathcal{S}_{\text{api}}$
  else ▷ *Graph Iterative Evolution*
    Sample $S_{\text{sample}} \subset \mathcal{V} \setminus \mathcal{S}$
    $(r, y) \leftarrow \text{Evolve}(S_{\text{sample}}, \mathcal{I}_{\text{iter}})$
    $\mathcal{V}_{\text{parents}} \leftarrow S_{\text{sample}}$
  end if
  $v_{\text{new}} \leftarrow (r, y)$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{v_{\text{new}}\}$
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u, v_{\text{new}}) \mid u \in \mathcal{V}_{\text{parents}}\}$
  return $\mathcal{G}, v_{\text{new}}$
end procedure

In this way, the graph can explore a much larger API composition space, thereby producing increasingly diverse training samples.

(1) Graph Initial Evolution. At the beginning, the graph contains only API nodes. We repeatedly sample a subset of API nodes and use their associated specifications $\mathcal{S}_{\text{api}} \subset \mathcal{S}$ as seeds. We then prompt the LLM to synthesize a sample node based on these APIs:

$$(r, y) = \text{Evolve}(\mathcal{S}_{\text{api}}, \mathcal{I}_{\text{init}}), \tag{2}$$

where $\mathcal{I}_{\text{init}}$ denotes the instruction prompt for graph initial evolution, r is the generated coding requirement, and y is the corresponding reference solution. This process produces the initial sample nodes in the graph. Since randomly sampled APIs may not always be tightly related, we do not strictly require the LLM to invoke every seed API in the generated solution.

(2) Graph Iterative Evolution. Once a pool of sample nodes has been obtained, we further grow the graph by iteratively synthesizing new sample nodes based on existing ones. Specifically, we randomly sample a small set of existing sample nodes $S_{\text{sample}} = \{(r_i, y_i)\}_{i=1}^{m}$ and prompt the LLM to evolve them into a new sample:

$$(r', y') = \text{Evolve}(S_{\text{sample}}, \mathcal{I}_{\text{iter}}), \tag{3}$$

where $\mathcal{I}_{\text{iter}}$ is the instruction prompt for graph iterative evolution. The operator Evolve$(\cdot)$ instructs the LLM to integrate multiple existing requirements into a single coherent and more diverse requirement r', and to produce a reference solution y' that satisfies this new requirement through coordinated invocation of multiple private APIs. By progressively applying graph iterative evolution, the synthesis graph moves beyond basic training samples and gradually covers more diverse development scenarios.

4.3 Multidimensional Graph Pruning

To address the poor-quality challenge identified in Section 3, we introduce *Multidimensional Graph Pruning* (Algorithm 3), a graph operator that removes low-quality sample nodes through multidimensional quality verification. For any new sample node (r, y), we retain it in the Graph only if it passes three complementary checks which target different failure modes: syntactic correctness, execution validity, and overall functionality. Formally, we define three binary validators $V_{\text{syn}}, V_{\text{exe}}, V_{\text{sem}} \in \{0, 1\}$ and retain a sample node in the Graph only if

$$V(r, y) = V_{\text{syn}}(r, y) \cdot V_{\text{exe}}(r, y) \cdot V_{\text{sem}}(r, y) = 1. \tag{4}$$

Algorithm 3 Multidimensional Graph Pruning

Input: Current graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Newly added node $v_{\text{new}} = (r, y)$
Output: Pruned graph $\mathcal{G}$

procedure PRUNING($\mathcal{G}, v_{\text{new}}$)
  $V_{\text{syn}} \leftarrow$ Parse y using standard AST parser ▷ *Syntactic Verification*
  $V_{\text{exe}} \leftarrow$ Execute y against generated unit tests ▷ *Execution Verification*
  $V_{\text{sem}} \leftarrow$ LLM-as-a-Judge assesses (r, y) ▷ *Overall Functionality Verification*
  if $V_{\text{syn}} \cdot V_{\text{exe}} \cdot V_{\text{sem}} == 0$ **then**
    $\mathcal{V} \leftarrow \mathcal{V} \setminus \{v_{\text{new}}\}$
    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{(u, v_{\text{new}}) \in \mathcal{E}\}$
  end if
  return $\mathcal{G}$
end procedure

(1) Syntactic Verification. We first perform lightweight static verification to ensure the basic well-formedness of the generated code. Specifically, we parse the reference solution y using the standard Abstract Syntax Tree (AST) parser of the target programming language. Any sample that fails to parse is discarded. This step removes obviously invalid code before invoking more expensive verification procedures.

(2) Execution Verification. Syntactic correctness does not imply correct private-API invocation. To better ensure execution validity and eliminate mistakes, we verify executability via test-based execution. For each sample (r, y), we prompt the LLM to generate a set of unit tests $T = \{t_j\}$, including representative inputs and corresponding assertions. We then execute y against T in a sandboxed environment. Samples are discarded if they raise runtime errors or fail any assertion.

(3) Overall Functionality Verification. Even if a sample is executable, the synthesized requirement and solution may still be unreasonable. To further improve data quality, we follow the widely adopted LLM-as-a-Judge paradigm [6, 7, 11] to assess the overall functionality of the synthesized samples. Concretely, we use the LLM to assess its own synthesized sample, including (i) whether the requirement r is realistic and well-defined, and (ii) whether the solution y truly satisfies r in intent. Any sample that fails this evaluation is also discarded.

4.4 Training and Deployment

After repeatedly applying Progressive Graph Evolution and Multidimensional Graph Pruning, we collect all retained sample nodes in the final graph and construct the synthetic training set $\mathcal{D}_{\text{syn}}$. We then fine-tune the model on $\mathcal{D}_{\text{syn}}$ using the objective in Eq. 1.

After training, the model can be directly applied to private-library-oriented code generation. Given a coding requirement r, the deployed model generates code as:

$$\hat{y} = \arg\max_{y} p_{\hat{\theta}}(y \mid r), \tag{5}$$

where the learned parameters $\hat{\theta}$ encode how to effectively invoke private-library APIs.

Moreover, we view PRICODER as orthogonal to prior RAG-based approaches. Existing approaches primarily improve how well the model can *see* the right API information at inference time, whereas PRICODER focuses on enabling the model to learn how to *invoke* private-library APIs effectively. In practice, the two directions can be combined. Let Retrieve(r) return relevant API knowledge. We

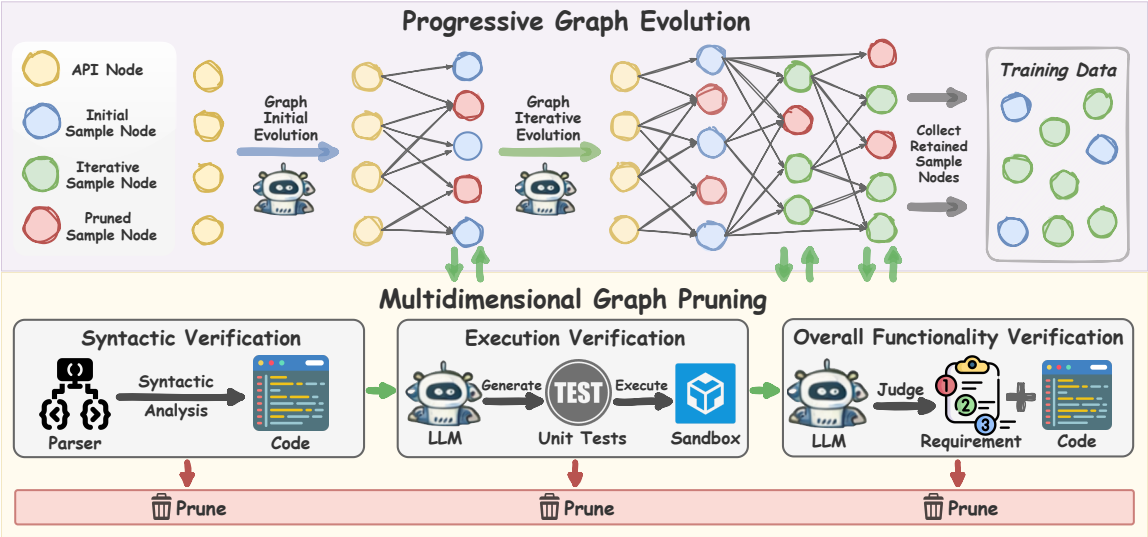


Figure 2: Overview of *PriCoder*. Starting from API nodes, *PriCoder* progressively evolves the synthesis graph to construct increasingly diverse sample nodes, while multidimensional graph pruning removes low-quality nodes through syntactic verification, execution verification, and overall functionality verification.

can condition the fine-tuned model on the augmented prompt:

$$\hat{y} = \arg\max_{y} p_{\hat{\theta}}\big(y \mid [r; \text{Retrieve}(r)]\big), \tag{6}$$

where $[\cdot;\cdot]$ denotes concatenation.

5 Benchmark Construction

Evaluating private-library-oriented code generation requires benchmarks built on target libraries that are unseen by the tested models. However, many existing benchmarks rely on libraries that were released long ago and may have already been fully learned by recent LLMs. For example, `TorchDataEval` [48], a benchmark widely used in prior work [47–49], evaluates models on the TorchData library [32], which was released in December 2021, well before the knowledge cutoffs of most modern LLMs. As a result, improvements on such benchmarks do not necessarily provide convincing evidence that a method truly enhances private-library-oriented code generation.

To enable a more rigorous evaluation, we construct two new benchmarks, `NdonnxEval` and `NumbaEval`. These benchmarks are designed to better reflect realistic private-library-oriented code generation scenarios, in which the target libraries are largely absent from the training corpora of modern LLMs. We describe the construction process below.

Library Selection. The primary requirement is to choose libraries that are unfamiliar to the evaluated models. To this end, we select two libraries, `ndonnx` [37] and `numba-cuda` [35], both of which were released in 2024 and underwent substantial development throughout 2025. In addition, both libraries are well-maintained and provide detailed API documentation, which facilitates reliable benchmark construction and evaluation.

LLM-Driven Draft Generation. To reduce manual effort, we leverage GPT-5.2 [39] to generate initial benchmark drafts, as it has already acquired sufficient knowledge of both libraries. For each library, we prompt the model to generate more than 500 draft instances, each consisting of a coding requirement and its corresponding unit tests.

Manual Curation and Refinement. To ensure benchmark quality, five student authors with extensive programming experience (more than four years on average) manually filtered and refined the generated drafts. To further ensure that each instance is indeed solvable, we also provided a reference solution for every instance and verified that it passes all corresponding test cases. After more than 150 person-hours of collective effort, we finalized 169 instances for `NdonnxEval` and 187 instances for `NumbaEval`. On average, each instance contains more than 9 test cases and requires the coordinated use of more than 4 distinct APIs. Detailed statistics of the two benchmarks are summarized in Table 2.

6 Experimental Setup

To assess *PriCoder*, we conduct comprehensive experiments to answer four Research Questions (RQs). In this section, we present the details of our experimental setup, including benchmarks, evaluation metrics, baselines, models, and other implementation details.

6.1 Research Questions

Our study aims to answer the following RQs.

RQ1: How does PriCoder perform in private-library-oriented code generation? This RQ aims to verify that *PriCoder* can effectively teach LLMs to invoke private-library APIs, thereby achieving superior results in private-library-oriented tasks. To answer this RQ, we compare *PriCoder* with multiple baselines across three LLMs.

RQ2: Does PriCoder negatively affect general code generation capability? Updating model parameters may introduce unintended degradation on general-purpose coding tasks. To assess this risk, we evaluate *PriCoder* on widely used public code generation benchmarks that do not involve the private libraries.

Table 2: Overview of the constructed benchmarks. #Instances denotes the number of instances in each benchmark; Avg. APIs denotes the average number of required APIs per instance; Avg. Tests denotes the average number of test cases per instance; Library denotes the target private library; First Release and Last Update denote the initial release time and the latest update time of the target library, respectively; and #APIs denotes the total number of APIs in the target library.

Benchmark	#Instances	Avg. APIs	Avg. Tests	Library	First Release	Last Update	#APIs
NdonnxEval	169	4.56	9.00	ndonnx	2024.06	2025.12	179
NumbaEval	187	6.73	9.10	numba-cuda	2024.06	2026.02	725

RQ3: What is the contribution of each component in PriCoder? Since PriCoder consists of two key components, we conduct comprehensive ablation studies to isolate their individual effects and quantify their contributions to overall performance.

RQ4: How key factors affects the effectiveness of PriCoder? This RQ studies how key factors influence the effectiveness of PriCoder. In particular, we focus on two key factors: the scale of the synthesized training data and the LLM used for data synthesis, and analyze how they affect the performance of PriCoder.

6.2 Benchmarks

We evaluate PriCoder on three benchmarks, including two novel private-library benchmarks and one widely used public benchmark.

- NumbaEval and NdonnxEval (Section 5). We release two novel benchmarks for rigorous evaluation of private-library-oriented code generation. NumbaEval evaluates code generation with numba-cuda library, and NdonnxEval evaluates code generation with ndonnx library.
- HumanEval [5]. We include this existing benchmark to assess general code generation capability. It consists of 164 hand-written programming tasks and mainly involves built-in functions, without requiring any third-party libraries.

6.3 Metrics

We adopt unbiased *pass@k* and *exec@k* as our main metrics.

- ***pass@k.*** For each instance, we sample $n \geq k$ candidate solutions (we use $n = 10$ and $k \in \{1, 3, 5\}$), execute the provided test cases, and count the number of passing solutions c. Following prior work [2, 4], we compute *pass@k* using the unbiased estimator:

$$pass@k = \mathbb{E}_{\text{instances}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \tag{7}$$

- ***exec@k.*** As shown in Section 3, models often misuse private APIs and trigger runtime failures. We therefore report *exec@k* to measure basic executability. It is defined analogously to *pass@k*, except that a solution is counted as successful if it runs to completion on the test inputs without raising any exception.

6.4 Baselines

Our core contribution lies in enabling models to automatically learn how to *use* private libraries. Therefore, our fundamental baseline is the original model that has not been applied with PriCoder, which we denote as ***Vanilla***.

Most existing approaches for private-library-oriented code generation are based on RAG. Although they are orthogonal to PriCoder, we include three representative approaches for comparison.

- ***Naive RAG* [48, 52].** It adopts a straightforward retrieval-augmented generation pipeline to search for relevant APIs and include their specifications directly in the prompt.
- ***EpiGen* [23].** It utilizes an LLM to decompose complex coding requirements into subtasks and retrieves API for each subtask to improve the relevance of injected knowledge.
- ***CAPIR* [30].** Building on task decomposition, *CAPIR* further reranks retrieved APIs using an LLM to improve the accuracy of injected API knowledge.

Furthermore, we include a baseline that represents the theoretical upper bound for existing RAG-based approaches. In this setting, we explicitly provide the LLM with the specifications of all APIs required to solve the specific coding requirements. Consistent with Section 3, we term this setting ***Oracle***.

6.5 Models

We apply PriCoder to three widely used LLMs that lack prior training exposure to the target libraries in NumbaEval and NdonnxEval: DeepSeek-Coder-6.7B-Instruct [10], Qwen2.5-Coder-7B-Instruct [13], and LLaMa3.1-8B-Instruct [8]. For brevity, we hereafter refer to them as DeepSeek-6.7B, Qwen-7B, and LLaMa-8B.

6.6 Implementation Details

For PriCoder, unless otherwise specified, we use each target model to synthesize its own training data to avoid confounding the results with potential distillation from a stronger model. For numba-cuda, we synthesize 6K training instances per model, and for ndonnx, we synthesize 20K training instances per model. The two synthesized datasets are used to train separate models for their corresponding benchmarks. During data synthesis, the number of samples produced by Graph Initial Evolution and Graph Iterative Evolution is controlled at a ratio of 1 : 2. We fine-tune the models using LoRA [12] with AdamW [28] for one epoch and a batch size of 16.

For baselines that require task decomposition or reranking, we use the same LLM as the backbone for these steps. We adopt the widely used BGE-M3 [33] as the embedding model for retrieval. For baseline-specific hyperparameters, we perform grid search on NdonnxEval and use the best-performing configuration.

To ensure fair comparison, all methods, including PriCoder, generate 10 samples per requirement with identical decoding settings, using temperature 0.5 and top-p 0.95. We use vLLM [17] as the inference framework and LLaMA-Factory [51] for model fine-tuning. All experiments are conducted on a server equipped with eight NVIDIA A100 GPUs, each with 80 GB of memory. Additional details are provided in the *Supplementary Material*.

Table 3: *pass@k* and *exec@k* (%) on `NdonnxEval` and `NumbaEval`.

Model	Method	NdonnxEval						NumbaEval					
		pass@1	*pass@3*	*pass@5*	*exec@1*	*exec@3*	*exec@5*	*pass@1*	*pass@3*	*pass@5*	*exec@1*	*exec@3*	*exec@5*
DeepSeek-6.7B	*Vanilla*	20.35	29.16	33.26	25.74	37.09	42.20	10.59	21.26	27.82	22.25	42.34	52.23
	+PriCoder	48.17	60.36	64.92	61.72	74.19	78.58	24.12	38.47	45.04	52.73	74.26	82.21
	Naive RAG	14.97	25.16	29.87	25.15	38.75	43.85	8.07	18.05	24.94	17.38	38.04	50.71
	+PriCoder	46.80	60.60	65.60	62.49	76.03	80.58	19.30	32.66	39.01	43.48	66.74	75.76
	EpiGen	16.86	25.78	30.27	27.04	40.32	46.31	11.18	23.61	30.97	22.83	45.91	57.40
	+PriCoder	44.79	59.04	63.39	61.89	74.95	78.95	24.49	38.70	45.28	51.39	72.05	79.73
	CAPIR	18.76	29.22	34.15	28.88	41.27	45.72	10.32	21.52	27.82	20.00	40.64	51.32
	+PriCoder	47.10	60.01	65.04	62.37	75.90	80.57	24.28	38.30	44.81	53.85	74.81	81.54
	Oracle	38.05	50.82	55.86	51.01	67.34	72.96	17.97	34.23	42.45	30.32	55.73	66.82
	+PriCoder	55.44	67.56	71.79	71.01	82.86	86.94	28.66	43.24	48.88	55.19	77.48	84.32
Qwen-7B	*Vanilla*	23.37	31.69	35.42	28.52	38.91	43.40	16.04	31.82	39.51	26.74	51.56	63.08
	+PriCoder	47.99	59.25	62.76	55.86	67.58	71.68	35.61	51.83	58.29	56.47	77.54	83.60
	Naive RAG	28.40	35.89	38.52	37.04	44.69	47.53	16.68	33.18	41.55	26.63	50.44	61.57
	+PriCoder	45.38	55.74	59.43	56.57	67.31	71.09	33.37	47.34	53.28	55.78	73.64	79.56
	EpiGen	28.82	35.69	38.00	33.73	41.48	44.46	16.47	32.75	40.70	27.43	52.33	63.15
	+PriCoder	47.63	56.49	59.58	57.93	68.09	71.50	35.08	49.74	56.69	56.84	75.43	81.06
	CAPIR	29.41	36.82	39.68	36.21	43.68	46.79	14.81	29.16	36.76	25.45	48.45	59.92
	+PriCoder	44.44	54.66	58.58	56.92	67.57	70.97	32.46	48.09	54.49	52.89	73.09	79.57
	Oracle	52.60	62.64	66.16	60.24	70.49	73.87	24.97	43.01	50.54	35.78	61.33	71.36
	+PriCoder	56.39	66.97	69.87	63.37	74.32	77.92	38.18	53.64	59.73	58.93	77.22	82.37
LLaMa-8B	*Vanilla*	15.86	23.50	26.55	23.14	34.38	38.65	8.13	16.97	21.73	22.78	44.36	55.25
	+PriCoder	32.31	45.28	50.57	50.18	65.69	70.92	23.58	35.34	40.84	65.94	82.92	87.90
	Naive RAG	15.74	23.71	26.74	30.77	42.88	46.70	6.79	15.89	21.88	19.73	41.44	53.16
	+PriCoder	22.78	35.51	40.91	46.39	62.77	68.62	25.13	37.72	43.92	65.56	82.12	87.14
	EpiGen	11.72	19.86	22.93	23.55	38.04	43.94	9.57	19.55	25.32	24.97	48.97	60.37
	+PriCoder	27.51	41.07	46.99	45.80	61.75	68.08	23.96	36.33	42.28	64.49	82.02	87.04
	CAPIR	13.20	19.97	22.69	22.25	34.77	40.18	8.40	18.19	23.86	25.67	49.08	60.62
	+PriCoder	30.24	42.61	47.17	48.17	63.47	68.59	23.64	35.16	40.36	66.20	82.72	87.42
	Oracle	30.36	42.79	47.45	46.57	64.42	70.38	13.10	26.73	33.66	29.36	52.52	62.55
	+PriCoder	36.75	51.93	57.64	55.98	75.35	81.29	27.75	42.45	49.09	67.54	84.60	89.39

7 Experimental Results

7.1 RQ1: Effectiveness on Code Generation with Private Library

The primary objective of PriCoder is to improve models' ability to invoke private-library APIs. To assess this capability, we evaluate the effectiveness of PriCoder on private-library-oriented code generation benchmarks in this RQ.

Setting. We apply the baselines and PriCoder to the three models described in Section 6.5 and evaluate them on the two benchmarks introduced in Section 5. We report *pass@k* and *exec@k* as evaluation metrics, where $k \in \{1, 3, 5\}$. As discussed in Section 4.4, PriCoder is orthogonal to existing RAG-based approaches. Therefore, in addition to evaluating each baseline independently, we also combine PriCoder with these baselines following Eq. 6.

Results. The results of different approaches are reported in Table 3.

❶ *PriCoder substantially improves private-library-oriented code generation performance.* Across different models and benchmarks, PriCoder yields substantial and consistent gains. For example, on `NdonnxEval` with `Qwen-7B`, PriCoder increases *pass@1* from 23.37% to 47.99% and improves *exec@1* from 28.52% to 55.86%. Similarly, on `NumbaEval` with `LLaMa-8B`, PriCoder raises *pass@5* from 21.73% to 40.84% and boosts *exec@5* from 55.25% to 87.90%. In contrast, existing RAG-based approaches usually bring only marginal improvements. For instance, on `NdonnxEval` with `Qwen-7B`, *EpiGen* improves *pass@1* and *exec@1* by only 5.45% and 5.21%, respectively.

❷ *PriCoder can be effectively combined with existing RAG-based approaches.* In many cases, combining PriCoder with RAG-based approaches yields better results than using either of them alone. For example, when combined with *CAPIR*, PriCoder achieves higher *pass@5* and *exec@5* on `NdonnxEval` with `DeepSeek-6.7B` than either individual method. Moreover, when combined with *Oracle*, PriCoder achieves the best results in nearly all settings. At the same time, we also observe that combining PriCoder with some RAG-based approaches sometimes yields performance similar to using PriCoder alone. We attribute this to the retrieval of irrelevant API knowledge, which may offset the benefit of retrieved relevant knowledge. This suggests that improving retrieval quality in private-library scenarios remains an important direction for future work.

Answer to RQ1: PriCoder can effectively teach LLMs to invoke private-library APIs, yielding substantial performance gains and also combining well with existing RAG-based approaches.

Table 4: *pass@k* and *exec@k* (%) on `HumanEval`.

Method	*pass@1*	*pass@5*	*exec@1*	*exec@5*
`DeepSeek-6.7B`				
Vanilla	71.89 (-0.00)	86.96 (-0.00)	93.60 (-0.00)	98.27 (-0.00)
Naive RAG	47.13 (-24.76)	70.88 (-16.08)	64.45 (-29.15)	88.65 (-9.62)
EpiGen	48.41 (-23.48)	72.88 (-14.08)	64.82 (-28.78)	89.54 (-8.73)
CAPIR	46.52 (-25.37)	71.29 (-15.67)	67.20 (-26.40)	86.90 (-11.37)
PriCoder	69.82 (-2.07)	85.92 (-1.04)	95.30 (+1.70)	98.37 (+0.10)
`Qwen-7B`				
Vanilla	85.43 (-0.00)	92.85 (-0.00)	98.29 (-0.00)	99.95 (-0.00)
Naive RAG	58.96 (-26.47)	77.91 (-14.94)	70.79 (-27.50)	88.91 (-11.04)
EpiGen	58.23 (-27.20)	77.23 (-15.62)	69.51 (-28.78)	87.14 (-12.81)
CAPIR	53.17 (-32.26)	72.17 (-20.68)	65.91 (-32.38)	84.33 (-15.62)
PriCoder	84.02 (-1.41)	90.99 (-1.86)	98.22 (-0.07)	99.78 (-0.17)
`LLaMa-8B`				
Vanilla	64.82 (-0.00)	78.17 (-0.00)	95.37 (-0.00)	99.29 (-0.00)
Naive RAG	46.59 (-18.23)	72.13 (-6.04)	75.67 (-19.70)	94.50 (-4.79)
EpiGen	46.40 (-18.42)	70.78 (-7.39)	78.84 (-16.53)	95.01 (-4.28)
CAPIR	39.88 (-24.94)	63.93 (-14.24)	66.34 (-29.03)	87.56 (-11.73)
PriCoder	65.12 (+0.30)	78.94 (+0.77)	96.46 (+1.09)	99.56 (+0.27)

7.2 RQ2: Impact on General Code Generation

Although *PriCoder* substantially improves models' ability to use private libraries, real-world development involves many requirements that do not rely on private-library APIs. In RQ2, we evaluate whether training with *PriCoder* compromises models' general code generation capability.

Setting. We apply the baselines and *PriCoder* to the three models described in Section 6.5 and evaluate them on `HumanEval`. We report *pass@k* and *exec@k* as evaluation metrics, where $k \in \{1, 5\}$.

Results. Table 4 reports the *pass@k* and *exec@k* on `HumanEval`. Due to space constraints, the results presented here focus on the `numba-cuda` setting. Specifically, for the RAG-based baselines, the retrieved corpus consists of the `numba-cuda` API documentation, and *PriCoder* is fine-tuned on the synthetic data generated for `numba-cuda` library. The results for `ndonnx` exhibit similar trends and are provided in the *Supplementary Material*.

❶ *PriCoder introduces negligible degradation to general code generation capabilities compared to the baselines.* For instance, for `Qwen-7B`, *PriCoder* reduces *pass@5* by only 1.86%, while *CAPIR* decreases *pass@5* by 20.68%. We attribute this degradation to the fact that RAG baselines inject private-library knowledge into the context without teaching the model when such knowledge should be applied, which can encourage unnecessary private-library API usage even on tasks that do not require it. Figure 3 illustrates such a failure case, where *Naive RAG* incorrectly invokes `numba-cuda` APIs for a requirement that does not need them, leading to an incorrect solution.

❷ *PriCoder can even enhance a model's general code generation capabilities.* Interestingly, in several settings, *PriCoder* actually improves both *pass@k* and *exec@k*. For instance, *PriCoder* increases the *pass@5* of `LLaMa` from 78.17% to 78.94% and its *exec@1* from 95.37% to 96.46%. We attribute this unexpected benefit to the rigorous Multidimensional Graph Pruning. By exclusively training on data that has passed Multidimensional Graph Pruning, the model reinforces its adherence to correct syntax and programming paradigms. As shown in Figure 3, the *Vanilla* model fails to use `isinstance`, whereas the model trained with *PriCoder* applies it correctly and successfully passes the test cases.

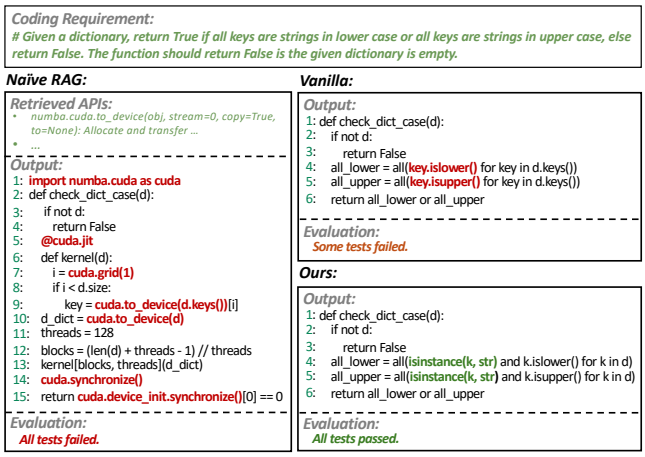


Figure 3: Case Study on `HumanEval` with `LLaMa-8B`.

Answer to RQ2: *PriCoder* introduces nearly zero degradation to the general code generation capabilities of the models and can even yield improvements in certain scenarios.

7.3 RQ3: Ablation Study

PriCoder consists of two key components, namely Progressive Graph Evolution and Multidimensional Graph Pruning. In this RQ, we conduct an ablation study to quantify the contribution of each component to the overall effectiveness of *PriCoder*.

Setting. To perform this ablation analysis, we design four experimental settings: ❶ ***Full PriCoder.*** We retain both Progressive Graph Evolution and Multidimensional Graph Pruning, which corresponds to the complete *PriCoder*. ❷ ***w/o Graph Pruning.*** We retain Progressive Graph Evolution but remove Multidimensional Graph Pruning, such that all evolved sample nodes are directly used for training without any filtering. ❸ ***w/o Graph Evolution.*** We retain Multidimensional Graph Pruning but remove Progressive Graph Evolution, so that the model directly synthesizes basic training samples based on API specifications. ❹ ***Direct Synthesis.*** We remove both Progressive Graph Evolution and Multidimensional Graph Pruning, and instead directly prompt the LLM with complete API documents to synthesize training samples for fine-tuning.

We conduct experiments on `Qwen-7B` using the three benchmarks described in Section 6.2. For `NdonnxEval` and `NumbaEval`, we evaluate models trained on synthesized data constructed for `ndonnx` and `numba-cuda`, respectively. For `HumanEval`, we evaluate the model trained on synthesized data constructed for `ndonnx`. We report *pass@k* and *exec@k* as evaluation metrics, where $k \in \{1, 5\}$. To ensure a fair comparison, we keep the amount of training data the same across all settings. We also include the *Vanilla* model as a reference point to more clearly show the improvements brought by different settings.

Table 5: Ablation study of *PriCoder* on `NdonnxEval` and `NumbaEval`.

Setting	NdonnxEval				NumbaEval			
	pass@1	*pass@5*	*exec@1*	*exec@5*	*pass@1*	*pass@5*	*exec@1*	*exec@5*
Vanilla	23.37 (+0.00)	35.42 (+0.00)	28.52 (+0.00)	43.40 (+0.00)	16.04 (+0.00)	39.51 (+0.00)	26.74 (+0.00)	63.08 (+0.00)
Direct Synthesis	32.96 (+9.59)	40.99 (+5.57)	39.11 (+10.59)	46.83 (+3.43)	17.91 (+1.87)	40.69 (+1.18)	31.55 (+4.81)	66.67 (+3.59)
w/o Graph Pruning	43.85 (+20.48)	57.61 (+22.19)	52.07 (+23.55)	67.36 (+23.96)	28.45 (+12.41)	46.92 (+7.41)	49.25 (+22.51)	76.17 (+13.09)
w/o Graph Evolution	44.85 (+21.48)	57.66 (+22.24)	52.60 (+24.08)	67.02 (+23.62)	32.57 (+16.53)	52.74 (+13.23)	53.96 (+27.22)	79.16 (+16.08)
PriCoder	47.99 (+24.62)	62.76 (+27.34)	55.86 (+27.34)	71.68 (+28.28)	35.61 (+19.57)	58.29 (+18.78)	56.47 (+29.73)	83.60 (+20.52)

Table 6: Ablation study of *PriCoder* on `HumanEval`.

CDS	*pass@1*	*pass@5*	*exec@1*	*exec@5*
Vanilla	85.43 (-0.00)	92.85 (-0.00)	98.29 (-0.00)	99.95 (-0.00)
Direct Synthesis	82.50 (-2.93)	88.25 (-4.60)	97.86 (-0.43)	98.79 (-1.16)
w/o Graph Pruning	82.13 (-3.30)	89.29 (-3.56)	97.74 (-0.55)	98.64 (-1.31)
w/o Graph Evolution	84.61 (-0.82)	90.45 (-2.40)	98.50 (+0.21)	99.91 (-0.04)
PriCoder	84.02 (-1.41)	90.99 (-1.86)	98.22 (-0.07)	99.78 (-0.17)

Results. Table 5 reports the ablation results on `NdonnxEval` and `NumbaEval`, and Table 6 reports the results on `HumanEval`.

❶ *Both Progressive Graph Evolution and Multidimensional Graph Pruning contribute to private-library-oriented code generation.* Removing either component leads to a clear performance drop on both `NdonnxEval` and `NumbaEval`. For example, without Progressive Graph Evolution, *pass@5* on `NdonnxEval` decreases from 62.70% to 57.66%. Without Multidimensional Graph Pruning, *pass@5* on `NumbaEval` drops from 58.29% to 46.92%. The performance degradation becomes even more severe under direct synthesis, where *exec@1* on `NumbaEval` further drops sharply from 56.47% to 31.55%.

❷ *Multidimensional Graph Pruning helps preserve general code generation capability.* When Multidimensional Graph Pruning is removed, both *pass@k* and *exec@k* on `HumanEval` decline. This proved that training on unfiltered synthesized data can introduce harmful noise, which may reinforce incorrect coding patterns and negatively affect general code generation.

Answer to RQ3: Both Progressive Graph Evolution and Multidimensional Graph Pruning are crucial to the performance of *PriCoder*.

7.4 RQ4: Analyzing Data Synthesis in *PriCoder*

Since *PriCoder* relies heavily on automated data synthesis, this RQ explores how this process impacts overall performance. We particularly analyze two key factors: *the scale of the synthesized data* and *the model used for synthesizing data*, and analyze how they affect the effectiveness of *PriCoder*.

Setting. To assess the impact of training data scale, we select `DeepSeek-6.7B` and train it on synthesized datasets of size {500, 1000, 5000, 10000, 20000, 30000, 40000}. To assess the impact of the model used for synthesizing training data, we select `Qwen-7B` and `LLaMa-8B` to synthesize 20K training samples, and then use the resulting data to train `Qwen-7B` and `LLaMa-8B`, respectively. All evaluations are conducted on the `NdonnxEval`, and we report both *pass@5* and *exec@5*.

Results. The results are shown in Figure 4 and Figure 5.

Impact of the scale of the synthesized data. ❶ *Performance improves steadily as the scale of synthesized data increases.* As the training set grows, the model achieves progressively better results.

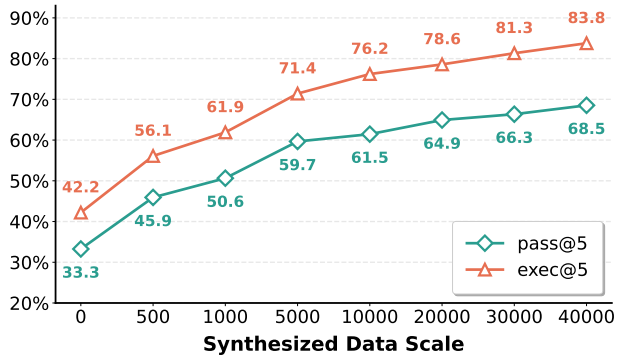


Figure 4: *pass@5* and *exec@5* on `NdonnxEval` with different synthesized training data scale.

For example, when the data scale increases from 10,000 to 40,000, *pass@5* improves from 61.5% to 68.5%. This trend suggests that organizations with sufficient resources can further strengthen private-library-oriented code generation by scaling up synthesized training data. ❷ *Substantial gains can be achieved even with a small amount of synthesized data.* Even with only 1,000 synthesized training samples, *pass@5* and *exec@5* increase from 33.3% and 42.2% to 50.6% and 61.9%, respectively. This indicates that *PriCoder* can already deliver meaningful improvements with a relatively small training set, making it practical for frequent and low-cost adaptation to newly evolved private-library knowledge.

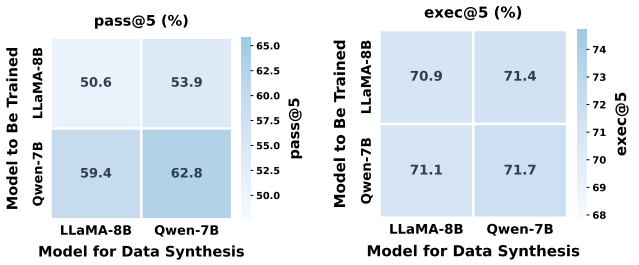
Impact of the model used for data synthesis. As shown in Figure 5, when training `LLaMa-8B`, using data synthesized by the stronger `Qwen-7B` yields performance similar to using data synthesized by `LLaMa-8B` itself. Specifically, *pass@5* reaches 53.9% and 50.6%, while *exec@5* reaches 71.4% and 70.9%, respectively. This result suggests that the effectiveness of *PriCoder* is relatively robust to the choice of the synthesis model. In practice, this implies that one may synthesize data once and then use it to adapt newly released models with relatively low additional cost.

Answer to RQ4: Increasing the scale of synthesized data consistently improves performance, while substantial gains can already be achieved with a small training set. Meanwhile, *PriCoder* remains robust to the model used for data synthesis.

8 Discussion

8.1 Computational Overhead Discussion

PriCoder requires both data synthesis and model fine-tuning, which inevitably introduces additional computational cost. However, this overhead is practically acceptable for most enterprises. In RQ1 and RQ2, for each target model, the total time for synthesizing training



(a) ***pass@5*** **on** `NdonnxEval`. **(b)** ***exec@5*** **on** `NdonnxEval`.

Figure 5: Impact of the model used for synthesizing training data on the effectiveness of *PriCoder*.

data and fine-tuning does not exceed 50 hours. Given that private libraries in real-world enterprise settings are typically maintained and used over long periods of time, and often support a substantial volume of downstream development, such a one-time adaptation cost is economically reasonable for long-term deployment. More detailed cost analysis is provided in the *Supplementary Material*.

8.2 Threats to Validity

We identify three primary threats to the validity of our study:

❶ *Generalizability of our results.* A potential threat is whether our findings generalize across different tasks and models. To mitigate this threat, we evaluate *PriCoder* on multiple benchmarks and models, including two benchmarks for private-library-oriented code generation and one widely used benchmark for general code generation. Following prior work [2, 4, 20], we report unbiased *pass@k* and *exec@k*, and repeat all experiments 10 times to reduce randomness. We also compare *PriCoder* against three representative prior approaches, together with an *Oracle* baseline that directly provides required API specifications.

❷ *Reliability of our results.* Evaluating private-library-oriented code generation requires benchmarks whose target libraries are unseen by the tested models. Ideally, such evaluation should be conducted on real enterprise private libraries. However, due to the closed-source nature of real-world private libraries, obtaining such libraries for research is diffucult and infeasible [41, 48]. To mitigate this threat, we construct two new benchmarks, `NdonnxEval` and `NumbaEval`. Both benchmarks are built on libraries that are unfamiliar to the evaluated models and can thus be viewed as private libraries from the models' perspective, thereby providing a reliable proxy for evaluation.

❸ *Replicability of our experiments.* To ensure reproducibility, we provide comprehensive details regarding our experimental settings within the *Supplementary Material*. In addition, we have open-sourced our code and the newly constructed benchmarks. These efforts significantly enhance transparency and guarantee that researchers can easily reproduce the results of our experiments.

9 Conclusion

In this paper, we study private-library-oriented code generation and show that, even given accurate required knowledge, LLMs still struggle to effectively invoke private libraries. To address this limitation, we propose *PriCoder*, an approach that teaches LLMs to invoke private-library APIs through automatically synthesized training data. To support rigorous evaluation, we further construct two benchmarks based on recently released libraries that are largely absent from the training corpora of modern LLMs. Extensive experiments show that *PriCoder* substantially improves private-library-oriented code generation and introduces negligible impact on general code generation capability. Additional ablation studies further confirm the effectiveness of both components in *PriCoder*.

References

[1] Shivam Adarsh, Kumar Shridhar, Çağlar Gülçehre, Nicholas Monath, and Mrinmaya Sachan. 2025. Siked: Self-guided iterative knowledge distillation for mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*. 9868–9880.

[2] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

[3] Liyi Cai, Yijie Ren, Yitong Zhang, and Jia Li. 2025. AI-Driven Self-Evolving Software: A Promising Path Toward Software Automation. *arXiv preprint arXiv:2510.00591* (2025).

[4] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Hyo Jin Do, Zahra Ashktorab, Jasmina Gajcin, Erik Miehling, Martín Santillán Cooper, Qian Pan, Elizabeth M Daly, and Werner Geyer. 2025. Generate, Evaluate, Iterate: Synthetic Data for Human-in-the-Loop Refinement of LLM Judges. *arXiv preprint arXiv:2511.04478* (2025).

[7] Yuanning Feng, Sinan Wang, Zhengxiang Cheng, Yao Wan, and Dongping Chen. 2025. Are We on the Right Way to Assessing LLM-as-a-Judge? *arXiv preprint arXiv:2512.16041* (2025).

[8] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[9] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–22.

[10] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. 2024. DeepSeek-Coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[11] Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2026. LLM-as-a-Judge for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* (2026).

[12] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *Iclr* 1, 2 (2022), 3.

[13] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[14] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[15] Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, et al. 2025. aiXcoder-7B: A Lightweight and Effective Large Language Model for Code Processing. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 215–226.

[16] Xue Jiang, Jiaru Qian, Xianjie Shi, Chenjie Li, Hao Zhu, Ziyu Wang, Jielun Zhang, Zheyu Zhao, Kechi Zhang, Jia Li, et al. 2026. KOCO-BENCH: Can Large Language Models Leverage Domain Knowledge in Software Development? *arXiv preprint arXiv:2601.13240* (2026).

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel,

et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[19] Chengze Li, Yitong Zhang, Jia Li, Liyi Cai, and Ge Li. 2025. Beyond autoregression: An empirical study of diffusion large language models for code generation. *arXiv preprint arXiv:2509.11252* (2025).

[20] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.

[21] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. Acecoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–26.

[22] Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, et al. 2026. What Papers Don't Tell You: Recovering Tacit Knowledge for Automated Paper Reproduction. *arXiv preprint arXiv:2603.01801* (2026).

[23] Sijie Li, Sha Li, Hao Zhang, Shuyang Li, Kai Chen, Jianyong Yuan, Yi Cao, and Lvqing Yang. 2024. Epigen: An efficient multi-api code generation framework under enterprise scenario. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. 6206–6215.

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[25] Jiaxin Liu, Yating Zhang, Deze Wang, Yiwei Li, and Wei Dong. 2025. THINK: Tackling API Hallucinations in LLMs via Injecting Knowledge. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 229–240.

[26] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 434–445.

[27] Wei Liu, Siya Qi, Yali Du, and Yulan He. 2026. Self-Play Only Evolves When Self-Synthetic Pipeline Ensures Learnable Information Gain. *arXiv preprint arXiv:2603.02218* (2026).

[28] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[29] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[30] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional API recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 87–98.

[31] Somshubra Majumdar, Vahid Noroozi, Mehrzad Samadi, Sean Narenthiran, Aleksander Ficek, Wasi Ahmad, Jocelyn Huang, Jagadeesh Balam, and Boris Ginsburg. 2025. Genetic instruct: Scaling up synthetic generation of coding instructions for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*. 208–221.

[32] Meta. 2021. torchdata. https://pypi.org/project/torchdata/.

[33] Multi-Linguality Multi-Functionality Multi-Granularity. 2024. M3-Embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216* (2024).

[34] Noor Nashid, Taha Shabani, Parsa Alian, and Ali Mesbah. 2024. Contextual api completion for unseen repositories using llms. *arXiv preprint arXiv:2405.04600* (2024).

[35] NVIDIA. 2026. numba-cuda (Version 0.27.0). https://pypi.org/project/numba-cuda/0.27.0/.

[36] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*. 15174–15186.

[37] QuantCo. 2025. ndonnx (Version 0.17.1). https://pypi.org/project/ndonnx/0.17.1/.

[38] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[39] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267* (2025).

[40] Chenlong Wang, Zhaoyang Chu, Zhengxiang Cheng, Xuyi Yang, Kaiyue Qiu, Yao Wan, Zhou Zhao, Xuanhua Shi, and Dongping Chen. 2025. Codesync: Synchronizing large language models with dynamic code evolution at scale. *arXiv preprint arXiv:2502.16645* (2025).

[41] Yunkun Wang, Yue Zhang, Zhen Qin, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. 2025. ExploraCoder: Advancing code generation for multiple unseen APIs via planning and chained exploration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 18124–18145.

[42] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120* (2023).

[43] Jiajun Wu, Jian Yang, Wei Zhang, Lin Jing, Yuqing Ma, Ensheng Shi, Yuchi Ma, Zhoujun Li, and Xianglong Liu. 2025. UCoder: Unsupervised Code Generation by Internal Probing of Large Language Models. *arXiv preprint arXiv:2512.17385* (2025).

[44] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[45] Lekang Yang, Yuetong Liu, Yitong Zhang, and Jia Li. 2025. DiffTester: Accelerating Unit Test Generation for Diffusion LLMs via Repetitive Pattern. *arXiv preprint arXiv:2509.24975* (2025).

[46] Jiarui Yuan, Tailin Jin, Weize Chen, Zeyuan Liu, Zhiyuan Liu, and Maosong Sun. 2026. SE-Bench: Benchmarking Self-Evolution with Knowledge Internalization. *arXiv preprint arXiv:2602.04811* (2026).

[47] Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2025. Private-library-oriented code generation with large language models. *Knowledge-Based Systems* 326 (2025), 113934.

[48] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. 277–288.

[49] Daoguang Zan, Ailun Yu, Bo Shen, Bei Chen, Wei Li, Yongshun Gong, Xiaolin Chen, Yafen Yao, Weihua Luo, Bei Guan, et al. 2024. DiffCoder: Enhancing large language model on API invocation via analogical code exercises. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 406–426.

[50] Yitong Zhang, Yongmin Li, Yuetong Liu, Jia Li, Xiaoran Jia, Zherui Li, and Ge Li. 2026. Lookahead-then-Verify: Reliable Constrained Decoding for Diffusion LLMs under Context-Free Grammars. *arXiv preprint arXiv:2602.00612* (2026).

[51] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Association for Computational Linguistics, Bangkok, Thailand. http://arxiv.org/abs/2403.13372

[52] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.