



API-Guided Dataset Synthesis to Finetune Large Code Models

ZONGJIE LI, Hong Kong University of Science and Technology, China

DAOYUAN WU*, Hong Kong University of Science and Technology, China

SHUAI WANG*, Hong Kong University of Science and Technology, China

ZHENDONG SU, ETH Zurich, Switzerland

Large code models (LCMs), pre-trained on vast code corpora, have demonstrated remarkable performance across a wide array of code-related tasks. Supervised fine-tuning (SFT) plays a vital role in aligning these models with specific requirements and enhancing their performance in particular domains. However, synthesizing high-quality SFT datasets poses a significant challenge due to the uneven quality of datasets and the scarcity of domain-specific datasets.

Inspired by APIs as high-level abstractions of code that encapsulate rich semantic information in a concise structure, we propose DATA SCOPE, an API-guided dataset synthesis framework designed to enhance the SFT process for LCMs in both general and domain-specific scenarios. DATA SCOPE comprises two main components: DSLT and DGEN. On the one hand, DSLT employs API coverage as a core metric, enabling efficient dataset synthesis in general scenarios by selecting subsets of existing (uneven-quality) datasets with higher API coverage. On the other hand, DGEN recasts domain dataset synthesis as a process of using API-specified high-level functionality and deliberately constituted code skeletons to synthesize concrete code.

Extensive experiments demonstrate DATA SCOPE's effectiveness, with models fine-tuned on its synthesized datasets outperforming those tuned on unoptimized datasets five times larger. Furthermore, a series of analyses on model internals, relevant hyperparameters, and case studies provide additional evidence for the efficacy of our proposed methods. These findings underscore the significance of dataset quality in SFT and advance the field of LCMs by providing an efficient, cost-effective framework for constructing high-quality datasets, which in turn lead to more powerful and tailored LCMs for both general and domain-specific scenarios.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Theory of computation** → **Program constructs**.

Additional Key Words and Phrases: Code synthesis, API coverage, Large code models

ACM Reference Format:

Zongjie Li, Daoyuan Wu, Shuai Wang, and Zhendong Su. 2025. API-Guided Dataset Synthesis to Finetune Large Code Models. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 108 (April 2025), 30 pages. <https://doi.org/10.1145/3720449>

1 Introduction

Large language models (LLMs) have demonstrated remarkable performance across a wide range of tasks following extensive pre-training [32, 34, 40, 73]. In the domain of code-related tasks, large code models (LCMs) such as CodeLlama [66] and StarCoder [38] have exhibited impressive capabilities in program understanding and generation, supporting various real-world applications.

*Corresponding authors.

Authors' Contact Information: Zongjie Li, zligo@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong, China; Daoyuan Wu, daoyuan@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong, China; Shuai Wang, shuaiw@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong, China; Zhendong Su, zhendong.su@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/4-ART108

<https://doi.org/10.1145/3720449>

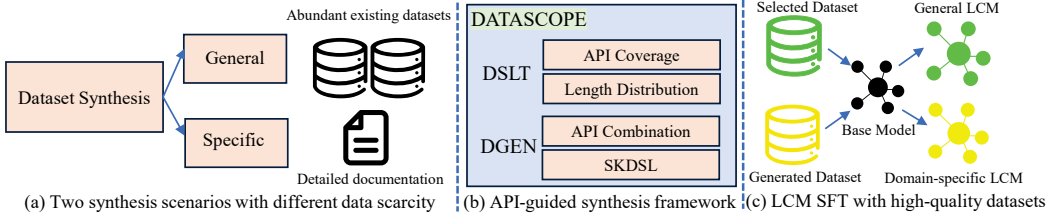


Fig. 1. Overview of the proposed API-guided dataset synthesis framework, from its (a) targeted scenarios to (b) methodology, and further to (c) usage for enhancing LCM SFT in general and domain-specific scenarios.

However, despite their vast knowledge acquired through training on enormous datasets, these base models may not achieve optimal performance across all use cases out-of-the-box. As illustrated in Fig. 1(a) and (c), to further align models with diverse requirements—including enhancing general code generation capabilities [39, 66] or specializing in specific codebases or domains (supporting commercial products like deep learning [56] or security-related [25] assists)—researchers often employ additional datasets to fine-tune base models, yielding more powerful and customized LCMs.

Among the various fine-tuning techniques proposed, supervised fine-tuning (SFT) has emerged as a critical approach for enhancing LLM capabilities. SFT leverages the knowledge acquired during pre-training while aligning models with human expectations [6, 12, 81]. This process involves further training the models on carefully curated instruction datasets, typically comprising formatted instruction-response pairs. These pairs, represented as (INSTRUCTION, RESPONSE), consist of human-provided tasks or queries (INSTRUCTION) and the corresponding desired outputs (RESPONSE) that the model should generate [9, 17, 54, 90].

Given the importance of high-quality SFT datasets for LCMs, various approaches are developed to create and curate such datasets. These methods include the collection of real-world code snippets [82] and the use of programming concepts and keywords (e.g., recursion and loops) to guide LLMs in dataset construction [46]. These efforts result in the generation of extensive datasets, as exemplified by Nemotron-4’s 800k examples [2]. While such large datasets offer potential benefits, they also present practical challenges in the SFT process. Notably, the computational resources required for processing extensive datasets can be substantial, which is particularly relevant given recent findings demonstrating that comparable performance can be achieved with as few as 2k high-quality examples [90]. Additionally, the generated data often focuses on solving problems using basic Python operations, potentially limiting its adaptability and efficacy in domain-specific scenarios, where we define a domain as a particular application area or specialized programming field (e.g., scientific computing and web development). Consequently, as illustrated in Fig. 1(a), researchers currently face a dichotomy: an *overabundance* of datasets with uneven quality for general scenarios, and a *scarcity* of high-quality, domain-specific datasets for specialized applications.

To address these challenges, we propose DATASCOPE, an API-guided dataset synthesis framework designed to enhance the fine-tuning performance of LCMs. DATASCOPE offers a comprehensive solution for dataset synthesis in both general and domain-specific scenarios, distinguishing itself from previous approaches. On the one hand, it aims to “distill” existing overly abundant SFT datasets to form concise, high-quality ones for general scenarios, thereby improving fine-tuning performance while simultaneously reducing SFT process’s cost. On the other hand, DATASCOPE supports the automated generation of domain-specific data to facilitate SFT in specialized contexts, enabling the efficient construction of high-quality SFT datasets without requiring real-world data or being constrained by particular powerful, proprietary LLMs.

The key observation underpinning DATASCOPE is the significant benefit APIs provide to LCMs. APIs, as high-level abstractions of code, encapsulate abundant semantic information within a *concise*

token structure. As will be shown in Sec. 3, our key insight is that such API-offered conciseness facilitates LCMs' comprehension, alters their internal behavior towards more accurate interpretation, and, consequently, improves their performance. Moreover, the power of APIs can further benefit the two scenarios in LCM SFTs: in contexts with overly abundant existing datasets, API coverage can serve as a core metric for *subset* selection, enabling LCMs to achieve superior performance with a compact dataset. Conversely, in data-scarce environments, combining diverse APIs provides an effective mechanism for constructing domain-specific datasets from scratch.

DATASCOPE concretizes the above insights and observations into two key components: DATASCOPE-SELECTION (DSL_T) and DATASCOPE-GENERATION (DGEN), which are designed to address dataset synthesis in general and domain-specific scenarios, respectively. As illustrated in Fig. 1(b), DSL_T employs API coverage as its primary metric, efficiently selecting subsets from existing datasets while also taking into account the length distribution of the code examples to ensure diversity in complexity. DGEN, on the other hand, leverages API information to control the generated program functionality, using a domain-specific language we designed in Sec. 4.3.2, called Skeleton Domain-Specific Language (SKDSL) to provide code snippet skeletons that guide the code structure, all without requiring real-world data.

To enable comprehensive evaluation of our approach, we develop LIBEN, a carefully curated benchmark of 345 high-quality instruction pairs across three specific domains. Our experimental results not only show the effectiveness of DATASCOPE, but also provide a series of valuable insights into API-guided dataset synthesis for LCM fine-tuning. DSL_T efficiently selects subsets with high API coverage from existing datasets, outperforming traditional selection methods like random sampling and clustering-based approaches. When fine-tuning models of varying sizes, it enables them to achieve performance comparable to or exceeding that of models fine-tuned on entire datasets. Notably, fine-tuning a 34 billion parameter LCM on just 5% of the data selected by DSL_T achieves 110% of the performance compared to fine-tuning on the full dataset.

Beyond selection, DGEN conceptualizes dataset generation as a process of transforming high-level requirements into concrete implementations. It facilitates domain-specific dataset synthesis through API combination and SKDSL utilization, without relying on real-world code snippets. This design enables automated generation of high-quality data using LLMs with limited capabilities at reduced cost. Specifically, domain-specific SFT datasets of 4,000 examples can be generated at a cost of only 3 USD, whereas a similar scale and quality human-written dataset requires over 334 volunteers [35]. This capability further enhances the SFT process in developing powerful models, offering a cost-effective and efficient pathway to domain-specific model enhancement. In summary, our contributions are as follows:

- Inspired by how APIs facilitate low-cost and effective LCM code comprehension, we propose DATASCOPE, a simple yet highly effective API-guided program dataset synthesis framework to enhance the LCM fine-tuning process from two key usage scenarios — general and domain-specific usage.
- For general SFT scenarios, we introduce DSL_T, which employs a greedy algorithm to efficiently select high-quality subsets from existing datasets. We propose well-designed, API-based objectives in the selection process, enabling LCMs to achieve superior performance with significantly less data.
- For domain-specific SFT scenarios, we present DGEN, which automates the generation of high-quality, tailored datasets. DGEN considers diverse APIs to achieve comprehensive coverage and uses SKDSL to provide code snippet skeletons, streamlining the creation process without relying on real-world data or proprietary LLMs.

- We demonstrate DATASCOPE’s effectiveness through comprehensive experiments. For domain-specific scenarios, we develop LIBEN, a specialized benchmark with 345 instruction pairs across three domains. Through extensive evaluation of both DSLT and DGEN on their respective scenarios and corresponding benchmarks, including analysis of LCMs’ performance and internal behavior, along with human assessment, we validate our method’s efficacy and provide insights into API-guided dataset synthesis for LCM SFT.

Due to space limitations, a full version of this paper is available in [42].

2 Background

This section introduces the background of LCMs and SFT. We explore the evolution of LCMs and clarify two distinct scenarios in LCM fine-tuning: general and domain-specific.

2.1 Large Code Models

The rapid advancements in deep learning (DL) techniques have led to the development of DL models capable of generating code with near-human or even superhuman performance [6, 12, 81]. These models are often integrated into development workflows through APIs (e.g., GPT-4 [53]) or IDE plugins (e.g., Codex [52]), revolutionizing the way developers write code by improving both efficiency and quality. The majority of state-of-the-art LLMs employ the Transformer [76] architecture, which relies on attention mechanisms to enable tokens to communicate and exchange information with each other. This architecture allows LLMs to generate text sequentially, predicting the next token based on the preceding context [61]. Building upon the success of LLMs, LCMs have emerged as a specialized variant tailored specifically for code-related tasks, leveraging the same underlying architecture. The development of a well-performed LCM typically involves a two-step process. First, a foundation model is selected and further pre-trained on a vast corpus of code, resulting in a “base model.” Second, the base model undergoes fine-tuning on a task-specific dataset using various fine-tuning techniques, ultimately yielding a fine-tuned model optimized for the desired task [6]. A notable example is the CodeLlama base model, which is derived from the Llama 2 foundation model and offers instructed version models [66].

2.2 Supervised Fine-Tuning

Formally, the SFT process of the target model can be outlined as follows: for the specific domain d with context c^d , each task example (x^d, y^d) is used to update the model parameters. This update aims at minimizing the loss function that measures the disparity between the data distribution and the target model distribution, as expressed below:

$$L_{\text{SFT}}(\theta) = -\log f_{\theta}(y^d | c^d, x^d), \quad (1)$$

Overall, this function seeks to minimize the negative log-likelihood of the target output y^d given the context c^d and input x^d , with respect to the model parameters θ . L_{SFT} converges when the generated response \hat{y} matches y^d , i.e., the distribution of fine-tuned model aligns with the task dataset distribution. Compared to other fine-tuning methods such as Reinforcement Learning from Human Feedback (RLHF) [54] or Direct Preference Optimization (DPO) [62], SFT is more efficient and effective, as it does not require a human preference dataset. Consequently, SFT becomes a standard procedure for developing high-quality general-purpose LLMs [5, 54] and has proven invaluable for customizing these models across numerous domains, such as medicine [68], finance [11], and various other fields, significantly enhancing their applicability and effectiveness in specialized contexts.

Notably, SFT methods can be further categorized into two main approaches: (1) full parameter supervised fine-tuning (SFT) and (2) parameter-efficient fine-tuning (PEFT). Although PEFT

demonstrates high performance while using fewer parameters, studies [23, 86] have shown that it primarily assists the model with response initiation and extracts most of the response from pre-trained knowledge. In other words, PEFT does not significantly contribute to the model's ability to acquire new knowledge. Therefore, in this study, we focus on the full parameter fine-tuning approach and refer to it as the SFT.

2.3 Two Mainstream LCM Fine-Tuning Scenarios

The primary purpose of fine-tuning LCMs is to enhance their performance on code generation tasks and align them with human instructions. Based on whether the fine-tuned models are intended for specific domains, LCM fine-tuning can be categorized into two mainstream scenarios, general and domain-specific, each facing distinct data scarcity, as illustrated in Fig. 1(a).

General Scenario. In this scenario, the fine-tuned model is designed as a universal code generation tool aimed at improving its general generation capabilities. A prime example is the instructed version of CodeLlama [66], which demonstrates superior performance across a wide range of code generation tasks compared to its base model. Researchers and practitioners working on fine-tuning general LCMs often face an abundance of existing datasets, such as OSS-instruct [82] and CodeExercise [13], which are typically derived from various online code repositories and programming forums using LLMs.

While the generation methods behind these datasets attempt to synthesize a diverse range of code snippets, they are inherently constrained by the quality of online code samples and the restricted number of programming forums. Consequently, these datasets often suffer from a paradox of quantity over quality, containing a large volume of data with inconsistent quality. For researchers aiming to construct SFT datasets to enhance a model's general performance, an efficient and cost-effective dataset synthesis approach would involve creating new SFT datasets by judiciously selecting from existing datasets.

Domain-Specific Scenario. In real-world applications, LCMs are often designed to meet the unique requirements of their target audience. For example, Pecan [56] focuses on generating machine learning code, while SQLCoder2 [15] specializes in generating SQL-related code. In such cases, fine-tuning aims to enhance the model's performance in specific domains by leveraging domain-specific datasets and integrating pertinent domain knowledge into the model.

Users aiming to tune domain LCMs frequently encounter a key challenge: the scarcity of high-quality, domain-specific datasets. Curating such datasets demands expertise in the target domain and often involves manual collection, cleaning, and annotation of code samples—a process that is both time-consuming and resource-intensive [14, 90]. Moreover, many valuable datasets may originate from proprietary company codebases, limiting their public availability and further exacerbating the data scarcity issue. Given these constraints, there is a pressing need for an efficient and adaptable framework that facilitates the synthesis of high-quality, domain-specific datasets. Such a framework would ideally minimize the reliance on extensive manual data generation while still capturing the nuances and complexities of the target domain, thereby enabling more effective and targeted fine-tuning of LCMs for specialized applications.

3 A Motivating Example

While APIs have been widely recognized for their effectiveness in program synthesis [70], this paper presents a novel perspective on their role in enhancing LCM performance through SFT dataset synthesis. To ease understanding, this section provides a motivating example to demonstrate how API-level abstractions can significantly improve an LCM's ability to understand and process code semantics, thereby underpinning our approach to SFT dataset synthesis.

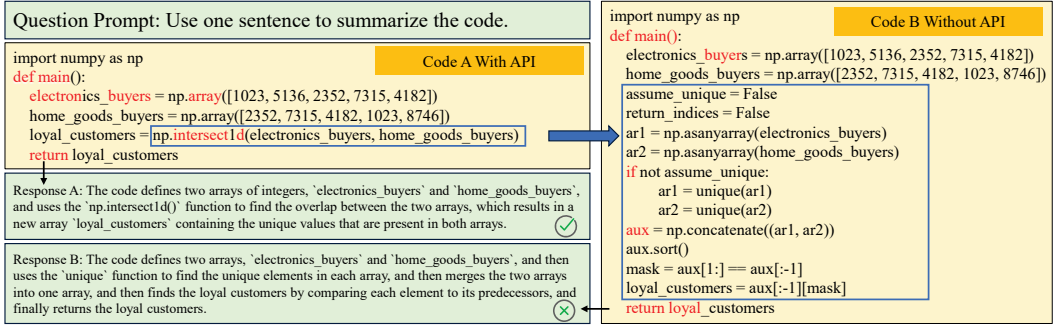


Fig. 2. An illustrative example showing the crucial role of APIs in helping LCMs understand programs. For readability, we have shortened the example code and simplified the API calls. The blue box on the left shows the original highly abstract code using APIs, while the right one displays the code after we manually expanded the API calls. The tokens highlighted in red are the top 20 tokens with the highest average attention scores across all attention heads in the model's ending layer.

Considering Fig. 2, we present two semantically equivalent code snippets that identify customers purchasing both electronics and home goods (loyal_customers). Code A employs the highly abstracted `np.intersect1d()` API call, while B replaces this with its official implementation. We input both snippets into the llama2-7B model, prompting it to summarize the program functionality. The model's responses for A and B are shown in the bottom left of Fig. 2.

Moreover, to gain insight into the model's internal processing, we analyze the attention mechanisms within the LCM. Attention scores, a key component of Transformer-based models, reflect the model's understanding of input importance [76]. We calculate the average attention score for each token across all attention heads in the model's ending layer. The top 20 tokens with the highest average attention scores are highlighted in red for both programs. This visualization offers a glimpse into how the LCM prioritizes different parts of the input, potentially revealing differences in its comprehension of highly abstracted API calls versus their expanded implementations. Based on the example, we summarize the following three key advantages of using API abstractions in LCM program comprehension:

1. **Token Efficiency:** The most apparent advantage of using APIs is the substantial reduction in token count. The API-abstracted code (snippet A) is significantly shorter than its expanded counterpart (snippet B), comprising only 49% of the original length. This reduction in token count directly impacts the computational intensity and resource requirements of LCM fine-tuning and inference processes, which are known to correlate positively with input token count [30, 76]. Thus, API abstractions potentially support more efficient LCMs.
2. **Semantic Comprehension:** A closer examination reveals the benefits of APIs in enhancing LCMs' understanding of program semantics. Real-world API names often encapsulate program functionality, enabling more concise and semantically rich code. Analyzing the model's responses, we observe that highly abstracted APIs more accurately reflect the program's functionality. Response A precisely captures the program's aim of finding customers who purchased both electronics and home goods, while response B merely describes the function's operations. This demonstrates how API abstraction swiftly mitigates LCMs' hurdles in understanding complex program semantics, thereby enhancing their performance.
3. **Internal Attention Patterns:** When examining the LLM's internal behavior, we notice that API calls typically receive higher attention scores. In contrast, manually expanded programs fail to allocate similar levels of focus on key tokens. This analysis further suggests that the

high-quality language abstraction provided by APIs augments the model's comprehension ability towards crucial semantic elements.

To ensure these observations are not specific to this single example, we additionally conduct a systematic study on 100 similar program pairs. Our statistical analysis confirms all three advantages: (1) Programs with APIs consistently use fewer tokens (average reduction of 68.76%); (2) Code summarization quality is significantly better when APIs are used, with 93% of summaries being judged as more accurate by LLM evaluators; and (3) API tokens demonstrate consistently high attention patterns, appearing among the top 20 tokens with the highest average attention scores in 88% of all cases. These findings convincingly support the advantages of API abstractions in LCM program comprehension.

Overall, our key insight, validated through both qualitative and quantitative analysis, is that APIs unleash a new dimension in SFT dataset synthesis by enhancing LCMs' comprehension capabilities while reducing their resource consumption through abstraction. This insight is particularly valuable given the pervasive use of APIs in modern programming, where the extra cost of incorporating APIs is often negligible compared to their substantial benefits. This insight is the cornerstone of our API-guided framework, which consists of two components to achieve well-performing and low-cost SFT dataset synthesis under different scenarios.

4 DATASCOPE

4.1 Overview

Building upon the insights from our motivating example (Sec. 3), which shows the significant benefits of APIs in enhancing LCM comprehension, we present DATASCOPE, a comprehensive framework designed to leverage these advantages in SFT dataset synthesis. As illustrated in Fig. 1(b), DATASCOPE comprises two key components: DSLT and DGEN, each tailored to address specific data availability contexts.

DSLT focuses on general scenarios where abundant SFT datasets exist. It employs API coverage as its primary metric for efficient subset selection, while also considering code example length distribution to ensure diversity. This approach optimizes the quality and representativeness of the selected data for fine-tuning, leveraging the semantic richness and token efficiency of API-based code, as demonstrated in our motivating example. DGEN addresses domain-specific scenarios characterized by data scarcity. It leverages API information to control the generated program functionality and uses a custom-designed Skeleton Domain-Specific Language (SKDSL) to guide code structure. This method enables creating high-quality, domain-specific datasets without relying on real-world data, while using the advantages of API abstraction. In the following sections, we provide detailed descriptions of DSLT (Sec. 4.2) and DGEN (Sec. 4.3).

4.2 DSLT: API-guided Dataset Selection for General Scenarios

This section introduces DSLT, our approach for optimizing SFT dataset selection. We begin by formulating the problem, followed by a detailed description of DSLT's core algorithm.

4.2.1 Problem Formulation. Given an SFT dataset $D = (x_i, y_i)_{i=1}^N$, where each example consists of an instruction x_i and its corresponding code y_i , our goal is to select a subset $D' \subseteq D$ of size n ($n \leq N$) such that a model trained on this subset achieves maximum performance on general code generation tasks, which can be formulated as:

$$\max_{D' \subseteq D} \text{Performance}(D') \quad \text{s.t.} \quad |D'| = n. \quad (2)$$

Despite the rather straightforward formulation, predicting the performance of a fine-tuned model is challenging. Drawing inspiration from our key observations in Sec. 3, which demonstrate the influence of APIs on LCM code comprehension, we propose using API coverage as a proxy measure for performance. Furthermore, building on previous research that highlights the importance of diversity in dataset quality [2, 57, 90], we also consider the diversity of code lengths in the selected subset. To quantify this aspect, we introduce a length diversity measure, $\text{LenDist}(D', D)$, which assesses the similarity between the length distributions of the subset D' and the original dataset D . Incorporating these design considerations, we reformulate our optimization problem as:

$$\max_{D' \subseteq D} \text{APICoverage}(D') \quad \text{s.t.} \quad |D'| = n, \text{LenDist}(D', D) \leq \tau, \quad (3)$$

where $\text{APICoverage}(D')$ represents the number of unique APIs covered in subset D' , and τ is a length diversity threshold. This formulation aims to maximize API coverage while maintaining a representative distribution of code lengths.

SFT Dataset Selection vs. Test Case Selection. Test case selection is a well-studied problem in software testing, with the objective to select a subset of test cases $T' \subseteq T$ from a large pool T while maintaining software quality [45, 85]. Although both test case selection and dataset selection can be formulated as optimization problems, there is a notable difference: as new, non-trivial examples are continuously added to the existing test case set, code coverage will monotonically increase, and the software will be tested more thoroughly. In contrast, such a monotonic behavior does *not* hold for SFT; adding more code snippets does not necessarily lead to better model performance on the task, as it may introduce more noise or cause overfitting [24], resulting in *performance degradation*.

4.2.2 Selection Algorithm. Although the optimization problem here can be solved through an exhaustive search approach, it quickly becomes infeasible for large datasets due to the combinatorial explosion of possible subsets. Thus, we propose an efficient sub-optimal greedy algorithm in DSLT. This algorithm iteratively selects examples to maximize API coverage while maintaining code length diversity. As shown in Algorithm 1, we first initialize the selected indexes subset as empty and the API set as empty (line 2). We also create a list bucket_{cnt} to track the number of examples per length bucket (line 3). This list is initialized by distributing the total number of examples to be selected (n) across buckets, based on the original dataset's length distribution (line 4).

In each iteration (lines 5-17), we select the example that adds the most uncovered APIs to the current API set. To ensure diversity, we prioritize examples from underrepresented length buckets. We traverse each non-empty bucket (line 7) and each example within the current bucket (line 8). For each valid example (i.e., not already selected and belongs to the current bucket), we calculate the number of new APIs it would add to the API set (lines 9-10). We update the best example if it adds more new APIs than the current best (lines 11-12). If the best example is found for the current bucket, we break the loop and move to the next iteration (lines 13-14).

After selecting the best example, we add its index to the selected indexes subset, update the API set with its APIs, and decrement the count for its corresponding length bucket (lines 15-17). This process continues until we reach n selections or exhaust API coverage improvements.

4.3 DGEN: API-Guided Dataset Generation for Specific Scenarios

Building on the insights from our motivating example (Sec. 3) and complementing DSLT for general scenarios (Sec. 4.2), we introduce DGEN, a framework that leverages API combinations to generate domain-specific SFT datasets. By combining API specifications to guide content generation and SKDSL to define structural constraints, DGEN can systematically generate high-quality datasets through different LLMs.

Algorithm 1 SFT data selection using DSLT.

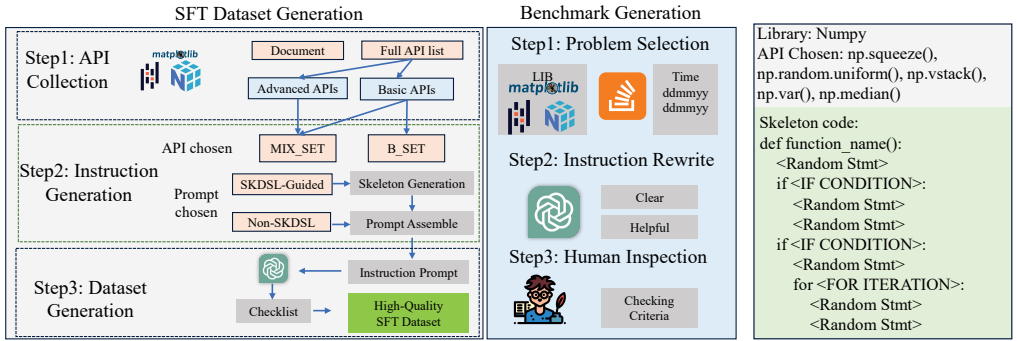
Require: n (Number of cases to select), api_stats (API statistics for each case), $case_lens$ (Lengths of each case), $buckets$ (Number of buckets for length distribution)

Ensure: Indices of selected cases

```

1: procedure SELECTAPI( $n, api\_stats, case\_lens, buckets$ )
2:   Initialize  $selected \leftarrow [], api\_set \leftarrow \{\}$  ▷ Init selected indices and API set
3:   Initialize  $bucket\_cnt \leftarrow [0] \times buckets$  ▷ Init bucket counts
4:   Calculate initial  $bucket\_cnt$  based on  $n, case\_lens$  and  $buckets$  ▷ Distribute  $n$  cases across buckets
5:   for  $i \leftarrow 1$  to  $n$  do
6:     Initialize  $best \leftarrow -1, max\_new \leftarrow -1$ 
7:     for  $bkt$  in prioritized order do ▷ Traverse each non-empty bucket
8:       for  $id, apis$  in  $api\_stats$  do ▷ Traverse each case for current bucket
9:         if  $id$  is valid for current  $bkt$  and not in  $selected$  then
10:           Calculate  $new\_apis$  for  $id$  ▷ Number of new APIs  $id$  would add to  $api\_set$ 
11:           if  $new\_apis > max\_new$  then
12:             Update  $best \leftarrow id, max\_new \leftarrow new\_apis$ 
13:         if  $best \neq -1$  then
14:           break
15:         if  $best \neq -1$  then
16:           Add  $best$  to  $selected$ , Update  $api\_set$  with APIs from  $api\_stats[best]$ 
17:           Decrement  $bucket\_cnt$  for the corresponding  $bkt$  by 1 ▷ Update bucket count
18:   return  $selected$ 

```



(a) Overview of DGEN for SFT dataset generation and LIBEN for evaluation. (b) An example of key components in DGEN's step2.

Fig. 3. Overview of DGEN for SFT dataset generation and LIBEN for evaluation, with an example of key components in DGEN's instruction generation process.

4.3.1 Problem Statement and Approach Overview. Extending our research beyond the scope presented in Sec. 4.2.1, we now confront a more complex challenge: the absence of a pre-existing high-quality dataset D . Our task evolves from subset selection to the creation of an entirely new dataset for domain-specific scenarios from scratch. Following the definition of domains as particular application areas or specialized programming fields in Sec. 1, we further concretize our approach in this section by using representative libraries to scope domains. Specifically, we deem one library representative enough to scope a domain - for instance, we consider Numpy as representative of the scientific computing domain, Pandas for data manipulation and analysis, and Matplotlib for data visualization. This pragmatic approach enables us to systematically generate domain-specific datasets by leveraging the APIs and documentation of these representative libraries.

Approach Overview. Fig. 3a illustrates the three-step process employed by DGEN for domain dataset synthesis: ① API collection: Given the documentation of a library (usually can be obtained by crawling the library’s official website), DGEN first extracts the complete API list, then categorizes them into “Advanced APIs” and “Basic APIs” based on their popularity and complexity. ② Instruction generation: DGEN further combines the APIs into B_SET and MIX_SET to specify the target functionality of the generated code. It also incorporates skeleton domain-specific language (SKDSL) to define the code structure, resulting in the assembly of comprehensive instruction prompts. ③ Dataset generation: Finally, DGEN queries LLMs with these constructed instruction prompts to generate data, validates the generated instances against a predefined checklist, and produces a high-quality SFT dataset.

SFT Data Generation vs. Test Case Generation. The generation of high-quality SFT datasets shares certain commonalities with test case generation in software testing, despite their divergent objectives. While test case generation aims to produce scenarios that trigger software failures, SFT dataset generation seeks to create exemplars for improved model fine-tuning. In test case generation, complex cases that traverse deeper code branches are generally considered more valuable than trivial ones [21, 55]. Similarly, in SFT dataset generation, the quality of instructions and code examples takes precedence over mere quantity. Zhou et al. [90] demonstrate that using 1,000 high-quality, manually crafted instructions by domain experts yields significantly greater performance improvements compared to larger but less curated datasets for SFT. However, a key distinction lies in the emphasis on diversity. For SFT datasets, achieving a diverse range of functional code completions is more crucial than solely pursuing complex corner cases.

4.3.2 Generation Algorithm. This section details DGEN’s three-step process for generating high-quality SFT datasets: API collection, instruction generation, and dataset generation. We first introduce the key insights of DGEN’s design, followed by giving the details of each step.

Key Insights. DGEN incorporates two key insights. First, it frames the task as a transformation process from high-level requirements to concrete implementations. By providing the LLM with specific and clear requirements for code generation, DGEN reduces the need for extensive domain-specific knowledge, enabling the model to focus on translating clear specifications into actual code. Second, DGEN decomposes complex generation problems into simpler sub-problems using API sets and SKDSL, lowering the capability threshold required at each step.

As a result of these insights, DGEN reduces the strict demand for a “high-capability” model, potentially enabling a *weak-to-strong* [7] generation paradigm where less capable models could be used for data generation, which is then used to further improve the model performance. DGEN’s novel approach enhances flexibility and cost-effectiveness, establishing it as a scalable framework for specific dataset synthesis, adaptable to varying model capabilities and domain requirements.

API Collection. As illustrated in Fig. 3a, we first extract APIs from the library’s official documentation, following the method proposed by [10]. This process yields 4,089, 3,296, and 3,683 APIs from Matplotlib, Pandas, and Numpy respectively, including their parameters and functional descriptions. Then, we rule out certain APIs based on the following criteria: (1) APIs starting with “_” or “c.” are excluded, as they are typically internal or low-level APIs related to the underlying C implementation; (2) For method implementations in both base and derived classes, we only retain the base class implementation as the API call; (3) any API invocation with a method chain [50] longer than three is excluded from further consideration.

Finally, we categorize the kept APIs into basic and advanced types. We select basic APIs directly from the examples in the official tutorials of each library, limiting their number to 50. These basic APIs are frequently used and easily understood, such as `numpy.sum` and `pandas.read_csv`. The

[System prompt]: You are a teacher who is good at {Library}. You are exceptionally skilled at crafting high-quality programming problems and offering precise solutions.

[User prompt]: Please take inspiration from the following list of application interfaces and their definitions to create a quality programming problem. Requirement: Use all APIs in the list. Present your output in two distinct sections: [Problem Description] and [Solution]. API list for inspiration: {API chosen} You will be given a Python code skeleton, and you need to follow the structure to complete your solution. Example Python code skeleton: {SKDSL code}

Guidelines for each section: 1. [Problem Description]: This should be "completely self-contained", providing all the contextual information one needs to understand and solve the problem. Assume common programming knowledge, but ensure that any specific context, variables, or code snippets pertinent to this problem are explicitly included. 2. [Solution]: Offer a comprehensive, "correct" solution that accurately addresses the [Problem Description] you provided.

Program	P	::=	S*
Stmt	S	::=	F I W R T M
FunctionDef	F	::=	def func_name() : S*
IfStmt	I	::=	if C : S*
			(elif C : S*)*
			(else : S*)?
WhileStmt	W	::=	while C : S*
ForStmt	R	::=	for E : S*
TryStmt	T	::=	try : S*
			(except X : S*)?
			(else : S*)?
SimpleStmt	M	::=	break continue
			return V D
Condition	C	::=	{IF CONDITION}
ForIter	E	::=	{FOR ITERATION}
Exception	X	::=	{Exception Type}
ReturnValue	V	::=	{RETURN VALUE}
RandomStmt	D	::=	{RANDOM STMT}

Fig. 4. The prompt template with two slots ({Library} and {API chosen}). Text in red will only show up when SKDSL is enabled.

Fig. 5. Selected syntax of SKDSL

remaining APIs are classified as advanced, which are typically less common and often require specialized knowledge, as exemplified by `numpy.linalg.eig` and `pandas.DataFrame.groupby`.

Notably, while this extraction and filtering process is straightforward for most public libraries with comprehensive documentation, we assume similar completeness for proprietary libraries considered for SFT dataset construction. Typically, only high-quality internal codebases typically require SFT, and organizations with substantial codebases may customize their filtering rules to best suit their specific needs and codebase characteristics.

Instruction Generation. As shown in Fig. 3a, the second step of DGEN involves preparing instructions for querying the LLM to synthesize the SFT dataset. Specifically, this step is divided into two separate parts: choosing the appropriate APIs and selecting the prompt template. For the chosen APIs, we consider two different strategies to create API sets of varying difficulty levels:

$$\begin{aligned} B_SET &= \text{RandomSample}(\text{BasicAPIList}, N) \\ \text{MIX_SET} &= \text{RandomSample}(\text{BasicAPIList} \cup \text{AdvancedAPIList}, N) \end{aligned} \quad (4)$$

B_SET focuses on commonly used APIs, ensuring high coverage of basic functionalities. MIX_SET introduces higher difficulty by incorporating advanced APIs, simulating real-world scenarios where complex APIs are used alongside basic ones. The selected APIs, along with their relevant information extracted from documentation, are then incorporated into the prompt template (see Fig. 4).

While API sets of varying difficulties control content complexity, we design SKDSL, a domain-specific language, to govern code structure by allowing rapid prototyping of Python's high-level logic flow through specified Python keywords (e.g., `def`, `if`, `else`). Given a random keyword list, we incrementally incorporate each keyword into the code, randomly injecting valid statements (e.g., `a = 1`) between keywords. These injected statements serve to create a more complete code skeleton, enabling subsequent validation by a syntax checker. The formal syntax of SKDSL is illustrated in Fig. 5. In the implementation, while our skeleton generation approach may occasionally produce invalid code skeletons, we maintain it for its simplicity and efficiency. If an invalid skeleton is detected, we discard and regenerate it. This process of generating and validating is significantly faster than querying the LLM for generation, often by a factor of thousands. Consequently, the generation of invalid skeletons has a negligible impact on overall efficiency.

After generating the skeletons, SKDSL integrates with a grammar checker to perform basic validation, catching syntactic errors before full implementation. This process enhances the quality of the generated skeletons. Subsequently, we standardize these skeletons by replacing random statements with the special token `<Random Stmt>` and conditions in control flow keywords (e.g.,

if, while) with <Corresponding Keyword + Condition>. These refined and standardized code skeletons then serve as the example code part in the prompt template (see Fig. 4), which is ultimately assembled into the final prompt for LLM querying. Fig. 3b shows an example, where a skeleton code incorporating two if statements and one for loop is generated using five chosen Numpy APIs, demonstrating how these elements are assembled to create the final prompt.

Dataset Generation. The final stage of DGEN uses the generated instruction prompts to query the LLM, producing library-specific instruction and response code pairs. Each pair undergoes format and length checks, discarding those that fail to meet specified criteria. These criteria include the presence of a code snippet and content length requirements, where pairs with fewer than 32 tokens or more than 4,096 tokens are excluded. Subsequently, we perform content validation on the remaining pairs. Since the inserted random valid statements enable grammatical correctness checks, we define a threshold T ($0 < T < 1$) to determine code acceptance. If the number of detected APIs in the generated code exceeds $N \times T$, where N is the required number of APIs, the pair passes content validation. Only the pairs that successfully pass all the aforementioned checks are included in our SFT dataset.

Notably, DGEN is designed to establish an efficient and automated SFT dataset generation process. The entire procedure does not necessitate any manual intervention, from API collection through instruction generation to dataset creation. It guarantees the scalability and reproducibility of the dataset generation pipeline, facilitating the creation of extensive, high-quality SFT datasets for a wide range of domain-specific libraries. By minimizing human involvement, we not only increase efficiency but also reduce the potential for human-induced biases or errors, ensuring consistent quality across large-scale dataset synthesis efforts.

5 Experimental Setup

In this section, we first introduce LIBEN, a benchmark we developed to evaluate domain-specific code generation. We then detail the experimental configuration for evaluating DATASCOPE.

5.1 LIBEN: A Benchmark for Evaluating Code Generation in Specific Domains

Before proceeding with the evaluation of DATASCOPE, it is crucial to establish an appropriate benchmark for testing. While the community has developed numerous benchmarks for assessing code generation, existing options either focus on general programming problems (e.g., HumanEval [10]) or present tasks of relatively low difficulty (e.g., DS-1000 [36]). To address these limitations, we develop LIBEN, a novel benchmark designed to assess model performance on specific domains. As outlined in Fig. 3a, the creation of LIBEN follows a three-step approach: ① problem selection through a highly automated collection process, ② instruction rewriting using an automated rewriting process, and ③ human inspection with rigorous selection criteria. This comprehensive process ensures the quality and relevance of the included questions, facilitating further research and assessment of LCMs in specific domains. In the following subsections, we detail each step of our approach.

5.1.1 Problem Selection. To demonstrate the versatility and effectiveness of DGEN across a range of practical programming scenarios, we focus on three distinct domains: scientific computing, data manipulation and analysis, and data visualization. These domains are represented by NumPy, Pandas, and Matplotlib, respectively, chosen for their comprehensive API collections and well-maintained codebases. Specifically, we use the Stack Exchange platform [71] to identify popular library-specific questions, searching with the target library name as the keyword. We prioritize questions with accepted answers or, in their absence, those with the highest-voted responses. To ensure quality and relevance, we apply a filtering criterion based on the average monthly vote count, considering only questions with at least 5 votes per month. We also verify the presence of

the target library in the answer code snippets. This process yields 439, 642, and 391 Python-related question-answer pairs for Numpy, Matplotlib, and Pandas, respectively, encompassing a diverse range of domain-specific tasks. Furthermore, it is important to note that the proposed technique is general and can be applied to other well-documented domains.

5.1.2 Instruction Rewriting. To enhance the quality of the selected question-answer pairs, we employ a rewriting process similar to previous works [44, 90]. First, we extract code snippets from raw answers, removing unrelated information, indentation symbols, and HTML tags (e.g., <p>). Then, we apply GPT-4 to generate refined instruction pairs based on the questions and answers. This process involves improving clarity, eliminating unnecessary details, and ensuring the instructions are easily comprehensible. The result is a set of high-quality instruction pairs that effectively capture the essence of the original question-answer pairs while significantly enhancing their clarity and usefulness for evaluating LCMs in specific libraries.

5.1.3 Human Inspection. In the final step, we conduct a human inspection of the rewritten instruction pairs. This process ensures the quality and relevance of each pair to the target library. We follow a set of predefined criteria that cover aspects such as contextualization, feasibility, readability, relevance, correctness, usefulness, and safety. Our inspection approach is adapted from [44] to suit code generation requirements. Through this rigorous evaluation, we curate a final benchmark dataset of 115 high-quality instruction pairs for each library.

Data Leakage. Notably, to mitigate the risk of data leakage and maintain the integrity of the evaluation, we exclusively consider questions posted between August 2023 and April 2024, ensuring that the selected data is temporally distinct from the training data of the LCM model used in our experiments. This temporal separation guarantees that the benchmark accurately assesses the model’s ability to generalize to unseen domain-specific tasks, providing a fair and unbiased evaluation of its performance. We conduct additional Membership Inference Attack (MIA) [67] tests to justify that LIBEN indeed does not appear in the model’s memory.

5.2 Experimental Configuration

SFT Datasets. We use two datasets in our experiments: CODEE and OSS. CODEE is primarily sourced from the CodeExercise dataset [13], which contains programming exercises covering a wide range of Python-related topics, including basic syntax, data structures, algorithm applications, and database queries. To further enhance the dataset’s diversity and quality, we supplement it with the MEIC dataset [77], which follows a similar data generation process and template, providing a substantial number of high-quality instruction pairs. In contrast, OSS is derived from the MagicCoder’s OSS-Instruct dataset [82]. This dataset generates a large number of instruction pairs by querying GPT-3.5 with a combination of real-world code snippets. To ensure data quality and consistency, we apply a further processing step to both datasets. We select Python-related examples, remove instances failing syntax checks or exceeding length thresholds, and eliminate potentially invalid or excessively long code snippets. This filtering process helps to eliminate any potentially invalid or excessively long code snippets that may adversely affect the training process. After processing, we obtain 76,512 examples from CODEE and 37,284 examples from OSS, forming the basis for evaluating our proposed SFT subset selection method.

Models. For our SFT experiments, we select CodeLlama, a family of LCMs based on Llama 2, as our starting point. CodeLlama has demonstrated strong performance in code-related tasks and has been widely adopted in previous works [16, 74, 87], establishing itself as a reliable benchmark model. To investigate the impact of model size on the effectiveness of SFT, we consider three different sizes of the CodeLlama base model: 7B, 13B, and 34B parameters. As noted in Sec. 2, META provides both base and instructed versions of CodeLlama. We opt for the base version to ensure that our

evaluation is not influenced by the fine-tuning process used in CodeLlama’s instructed version, as the details of their SFT process are not clearly disclosed [66]. Using the base version allows us to isolate the effects of DSLT and maintain a fair comparison across different model sizes, without potential confounding factors introduced by the instructed version’s fine-tuning process.

Benchmarks. For evaluating DSLT, we employ HumanEval [10], a widely accepted benchmark for assessing code generation capabilities [3, 16, 51, 72]. It offers a diverse set of programming problems with human-written instructions and solutions. For DGEN, we use LIBEN, as introduced in Sec. 5.1, to assess performance on domain-specific scenarios.

Metrics. For DSLT, we employ two primary metrics: Pass@k and Jensen-Shannon (JS) divergence. Pass@k assesses the accuracy of LCMs in solving programming problems, while JS divergence measures the similarity between code length distributions of the selected subset and the original dataset. Following [38, 47, 83], we set $k = 1$ for the evaluation.

For DGEN, we primarily use CodeBLEU[64] to assess the quality of generated code snippets. To provide a more thorough evaluation, we supplement this with three additional metrics: cyclomatic complexity [48], Silhouette Coefficient [65], and Calinski-Harabasz Index (CH-Index) [8]. These metrics offer insights into code complexity and clustering quality. Despite the prevalence of evaluation methods based on test cases (e.g., HumanEval) in assessing code generation, CodeBLEU serves as our primary metric due to two key factors. First, the StackOverflow answers forming our dataset typically lack accompanying test cases, making consistent implementation of test-based evaluations challenging. Second, for libraries like Matplotlib that deal with data visualization, designing meaningful test cases is problematic due to the difficulty in asserting program behavior for graphical outputs. To address the limitations of relying solely on CodeBLEU, we incorporate human inspection and LLM-based evaluation in Sec. 6.3.2, providing qualitative insights into the generated code’s effectiveness and correctness. This approach ensures a more comprehensive assessment of DGEN’s performance by combining quantitative metrics with qualitative analysis.

Hyperparameters. DSLT and DGEN share most hyperparameter configurations, which can be found in the full version. Specifically, for DSLT, we set 40 buckets and vary the budget constraint n as 2.5%, 5%, 10%, 20%, and 25% of the training data size. For DGEN, we adjust the training batch size from 64 to 32 for datasets containing fewer than 2,000 examples to enhance training stability and ensure reliable results.

6 Evaluation

We conduct extensive experiments to evaluate DATASCOPE in both data selection for general scenarios (RQ1 and RQ2) and data generation for specific scenarios (RQ3-RQ5):

- **RQ1:** How effective is DSLT in selecting representative examples from SFT datasets?
- **RQ2:** How does DSLT impact the performance of SFT models?
- **RQ3:** How effective is DGEN in generating high-quality SFT datasets and improving model performance?
- **RQ4:** How does DGEN-generated data affect the internal representations and discriminative abilities of models?
- **RQ5:** How does DGEN perform under different hyperparameter settings and models?

Additionally, we compare DGEN with alternative strategies in Sec. 6.6 to validate our design.

6.1 RQ1: Effectiveness of DSLT in Selecting Representative Examples

6.1.1 Main Results. We conduct experiments on two datasets (OSS and CODEE) and compare DSLT against two baselines: random selection (Random) and clustering-based selection (CaR) [22]. For the Random baseline, we randomly select three sets of examples using different random seeds, perform

Table 1. Comparative analysis of subset selection methods on LIBEN. API coverage (higher is better) and Jensen-Shannon (JS) divergence (lower is better) are shown for different subset sizes using Random selection, Clustering-and-Retrieve (CaR), our proposed DSLT, and its length-constrained version (TOPAPI).

Dataset	Size	API Coverage (%)				JS Divergence			
		Random	CaR	DSL _T	TOPAPI	Random	CaR	DSL _T	TOPAPI
OSS	2.5%	6.84	4.05	18.95	12.09	0.0787	0.0886	0.0763	0.3945
	5%	13.08	11.55	39.84	21.95	0.0760	0.0739	0.0734	0.4032
	10%	26.53	22.45	55.33	28.34	0.0678	0.0781	0.0677	0.4105
	20%	25.99	40.06	67.23	29.58	0.0772	0.0801	0.0659	0.4119
	25%	31.67	56.26	77.82	29.58	0.0765	0.0798	0.0546	0.4119
CODEE	2.5%	6.74	5.70	23.37	11.00	0.0546	0.0560	0.0512	0.3604
	5%	11.86	11.70	36.98	17.59	0.0538	0.0539	0.0514	0.3452
	10%	19.45	19.09	56.45	26.77	0.0543	0.0550	0.0503	0.3357
	20%	32.17	32.21	89.00	26.77	0.0547	0.0539	0.0310	0.3357
	25%	38.21	37.97	100.00	26.77	0.0543	0.0540	0.0208	0.3357

a preliminary study, and choose the seed that yields the best results for subsequent experiments. In the CaR baseline, we follow the same setting, where we use the Sentence-BERT model [63] to embed the instruction pairs and apply K-means [31] method on the feature vectors processed by PCA [1]. We then select the top k examples from each cluster as representatives. The results are in Table 1, where API coverage is calculated based on the unique APIs in the selected subset.

API Coverage Analysis. DSL_T consistently achieves higher coverage compared to both Random and CaR across all subset sizes and datasets. Notably, at the 25% subset size, DSL_T covers 77.82% and 100% of the unique APIs for the OSS and CODEE datasets, respectively, outperforming the baselines by 33.85% and 41.80%. This shows the effectiveness of DSL_T in maximizing API coverage through its iterative greedy selection approach. As the dataset size increases, the API coverage of all three methods increases. However, the convergence speed is slower on OSS compared to CODEE, with DSL_T achieving a coverage increase of 58.87% from 2.5% to 25% subset size on OSS, compared to a 76.63% increase on CODEE. This indirectly reflects the challenge of finding a well-covering subset for OSS, as it stems from complex real-world code crawled from the internet. Interestingly, all methods' API coverage values are higher than the corresponding dataset percentages. We attribute this to that different examples contain varying numbers of APIs, and they could share some common APIs, such as `max()`, `min()`, etc. Such usage overlap leads to higher API coverage values.

JS Divergence Analysis. DSL_T consistently maintains lower divergence compared to the baselines, especially at larger subset sizes. On average, DSL_T achieves JS divergence that is 0.0128 lower than Random and 0.0203 lower than CaR. At the 25% subset size, DSL_T achieves JS divergence of 0.0546 and 0.0208 for OSS and CODEE, respectively. These values are 0.0219 and 0.0252 lower than Random, and 0.0252 and 0.0332 lower than CaR for OSS and CODEE, respectively. This highlights DSL_T's ability in selecting subsets that are close to the original dataset. Similar to the API coverage results, the JS divergence decreases more slowly on OSS compared to CODEE, with a decrease of 0.0217 from 2.5% to 25% subset size on OSS, compared to a decrease of 0.0304 on CODEE. This further reflects the complexity of the data distribution and the effectiveness of DSL_T in handling it.

6.1.2 Ablation Study on Length Diversity. To investigate how length diversity affects performance, we conduct an ablation study by modifying our selection algorithm to prioritize API coverage while constraining program length. Specifically, we set a threshold at the median length of the dataset and only select programs below this threshold while maintaining high API coverage. We name this approach TOPAPI because we prioritize the programs with more APIs. This approach aims to maximize API coverage while systematically excluding longer programs.

Coverage Analysis. The results in Table 1 demonstrate the impact of length constraints on selection performance, where higher API coverage and lower JS divergence indicate better performance. For

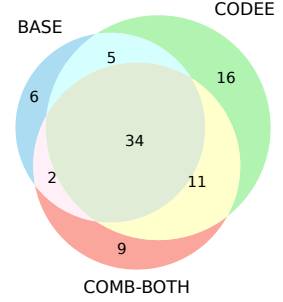
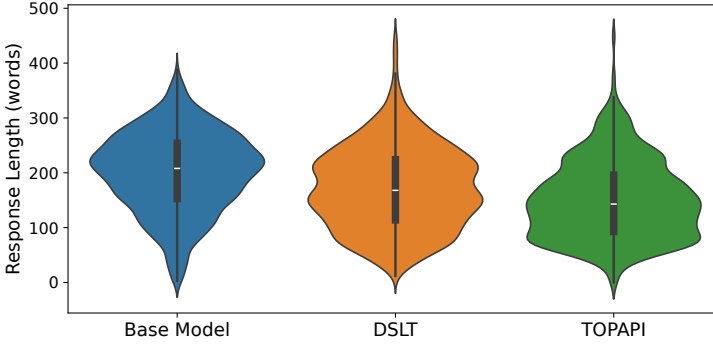


Fig. 6. Response length distribution comparison across different models. Fig. 7. Overlap of passed test-cases among selected models.

the OSS dataset, the length-constrained TOPAPI achieves significantly lower API coverage (29.58% vs. 77.82%) and higher JS divergence (0.4119 vs. 0.0546) compared to DSLT. Similarly, for the CODEE dataset, TOPAPI shows substantial degradation in both metrics, with API coverage dropping from 100% to 26.77% and JS divergence increasing from 0.0208 to 0.3357.

Notably, TOPAPI's performance plateaus after 10% subset size for both datasets. This saturation occurs because the length constraint effectively prevents the selection of longer programs that could potentially contribute new APIs. At this point, the remaining shorter programs in the candidate pool either contain APIs that are already covered or are rejected due to the length threshold, creating a coverage ceiling. These results demonstrate the critical role of length diversity in example selection. Constraining program length not only limits API coverage but also distorts the natural distribution of the training data, highlighting the importance of DSLT's length-diverse selection strategy.

Impact on Model Generation Length. To evaluate the impact of length diversity on models' generation behavior, we fine-tune CodeLlama-7B using 5% of CODEE dataset selected by both DSLT and TOPAPI. We then analyze the length distributions of responses generated by these models and the base model on 1,000 randomly sampled OSS test cases (Fig. 6).

The violin plot reveals distinct generation patterns: while the base model shows the highest average length, model fine-tuned on DSLT produces moderately shorter responses with similar distribution shape. In contrast, the model fine-tuned on TOPAPI exhibits a clear bias toward shorter generations. It demonstrates that length-constrained examples significantly limit the model's generation flexibility, while DSLT's diverse length selection strategy helps maintain the model's ability to generate responses in a wide range of lengths. This finding shows the importance of preserving length diversity in training examples for developing more capable LCMs.

Answer to RQ1: DSLT consistently outperforms baseline methods in both API coverage and distribution similarity across different subset sizes and datasets. Our ablation study further reveals that maintaining length diversity in example selection is crucial for achieving comprehensive API coverage and developing models with flexible generation capabilities, demonstrating the effectiveness of DSLT in selecting representative examples from SFT datasets.

6.2 RQ2: Impact of DSLT on SFT Model Performance

Fig. 8 presents the evaluation results of DSLT on HumanEval using CodeLlama models of different sizes (7B, 13B, and 34B), in which the score is represented by pass@1 score. Additionally, we evaluate DSLT on the MBPP benchmark [4] under the same setting. Overall, we view the results as

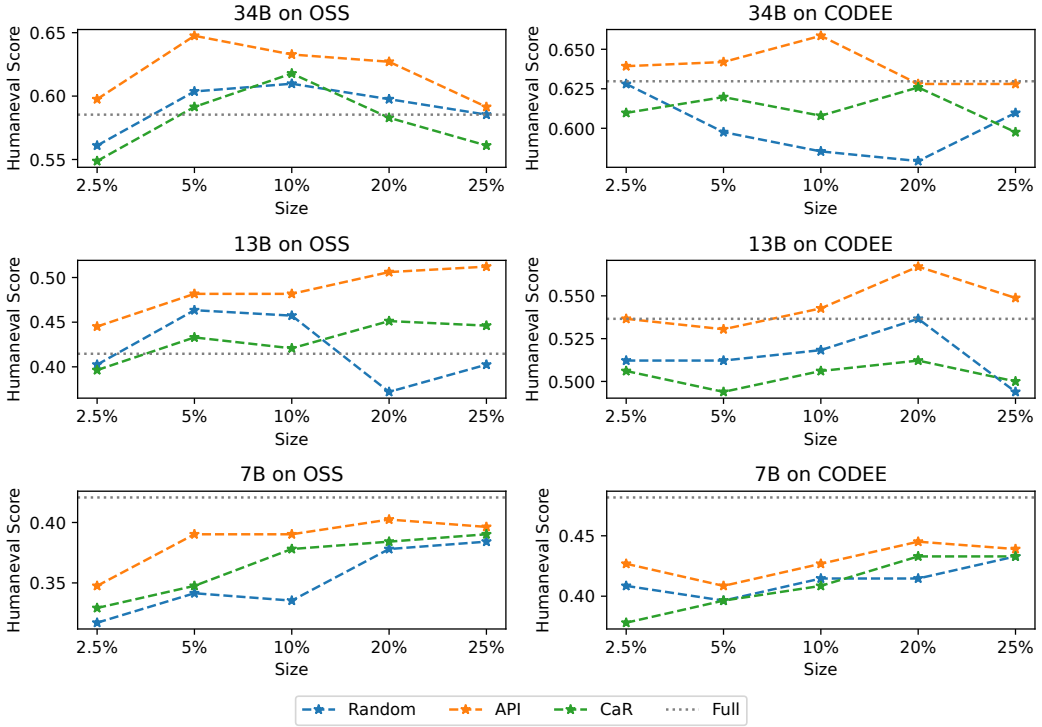


Fig. 8. Pass@1 score of DSLT, Random, and CaR on the HumanEval benchmark using CodeLlama models of different sizes trained on subsets of varying sizes from the OSS and CODEE datasets.

highly promising. Before delving into a detailed analysis of DSLT’s effectiveness, it is important to highlight its most direct benefit to developers: the significant reduction in training costs. When using only 5% of the data, the required resources often decrease proportionally to around 5% or even lower (due to faster model convergence). For instance, training a 34B model on the complete CODEE dataset costs approximately 170 USD, whereas training on 5% of the data costs around 8 USD.¹ This substantial cost reduction demonstrates the immediate practical advantages of DSLT in making large-scale model SFT more accessible and economically viable.

We first investigate how models fine-tuned using DSLT-selected subsets perform compared to those trained on the full SFT dataset (denoted as “FULL” in Fig. 8). We observe that for the 7B model, using the full SFT dataset yields better results than using subsets selected by DSLT. However, this trend reverses for the 13B and 34B models, where using fewer examples selected by DSLT leads to better performance. Moreover, we find that the larger the model, the smaller the number of examples required to achieve superior performance compared to training on the full dataset.

This phenomenon can be attributed to several factors: The 7B model may not have effectively learned code-related knowledge during pretraining, thus requiring more data to assist in its learning process during SFT. On the other hand, the 34B model has already acquired enough knowledge, enabling it to learn more effectively with only a small amount of carefully selected data during SFT. This finding has significant implications, as larger models incur higher fine-tuning costs in terms of both time and hardware requirements. The effectiveness of DSLT in improving the larger

¹Calculated based on H800 service provider pricing provided by the authors’ institution, excluding data storage costs.

models' performance while using fewer examples demonstrates its potential to greatly reduce the computational resources needed for SFT, making large-scale LCMs more feasible.

Besides the full dataset results, DSLT largely improves the performance of SFT models across all model sizes and datasets compared to the random selection (Random) and clustering and ranking (CaR) baselines. On average, DSLT outperforms Random and CaR by 7.96% and 7.14%. In general code generation scenarios where syntax and semantics correctness are required, this relative improvement means approximately 7-8% more user queries can be correctly processed on the first attempt. Even considering just a single round of re-querying (i.e., simply asking again to obtain the correct result), this improvement translates to at least a 7% reduction in computational costs, which can be significant in large-scale applications. Similar patterns of improvement are also observed on the MBPP benchmark, although different SFT datasets show varying degrees of absolute improvement on different benchmarks.

Impact of Model Size and Dataset Quality. First, we observe that as the model size increases from 7B to 34B, the relative improvement also increases. For instance, on the CODEE dataset, compared to the base model fine-tuned with two baselines, DSLT achieves an average Pass@1 improvement of 4.29% for the 7B model, while the improvement increases to 7.06% and 5.45% for the 13B and 34B models, respectively. This suggests that larger models generally benefit more from DSLT. Second, we notice that fine-tuned models' performance on the OSS dataset is consistently lower than those tuned on the CODEE dataset by a margin of 8.29%. This significant gap highlights the importance of dataset quality and the limitations of using open-source code to construct SFT datasets. Finally, we find that for larger models (13B and 34B), increasing the subset size does not always lead to better performance, which is consistent with the findings in [17].

Answer to RQ2: DSLT significantly reduces training costs by using only a small portion of the dataset. It enables larger models to achieve better performance with fewer examples compared to using the full dataset, and consistently outperforms baseline methods across different model sizes and datasets.

6.3 RQ3: Effectiveness of DGEN in Improving SFT Models

In this section, we examine DGEN's performance on LIBEN and analyze the factors that potentially affect the results. Additionally, we employ both LLM-based and human evaluations to assess the quality of DGEN's output.

6.3.1 Main Results. Table 2 presents the evaluation results on LIBEN using CodeBLEU as the metric for CodeLlama models of 7B and 13B sizes. We compare the performance of base models without fine-tuning, models fine-tuned on the full CODEE dataset (76K examples), and models fine-tuned using our generated datasets. "DSL-Guided" indicates whether SKDSL is used in dataset generation. The "Dataset" column specifies the prompt selection method: BASIC uses only the B_SET, MIX uses only the MIX_SET, COMB combines both BASIC and MIX, and COMB-BOTH randomly selects 2K examples from each of the two COMB datasets, resulting in a total of 4K examples. The "# Examples" column is the number of examples in each SFT dataset.

The results in Table 2 demonstrate that fine-tuning the models using our generated datasets consistently improves their performance on all three libraries compared to the base versions without fine-tuning. For instance, fine-tuning the 13B and 7B models with COMB-BOTH (4K examples) achieves CodeBLEU scores of 0.3973 and 0.3795 on Matplotlib, representing relative improvements of 82.4% and 163.2% over their respective base models. Moreover, using COMB-BOTH achieves better results than fine-tuning with the full 76K examples from the CODEE dataset. For example,

Table 2. Performance comparison of 7B and 13B models on LIBEN. Scores are reported using the CodeBLEU metric. Bold values indicate the highest scores for each model size and library combination.

Model Size	Dataset	DSL-Guided	# Examples	Numpy	Pandas	Matplotlib
13B	-	-	-	0.1917	0.2014	0.2177
13B	CODEE	-	76k	0.3218	0.3183	0.3436
13B	MIX	No	2k	0.3380	0.3151	0.3679
13B	BASIC	No	2k	0.3493	0.3319	0.3699
13B	COMB	No	4k	0.3563	0.3470	0.3722
13B	BASIC	Yes	2k	0.3546	0.3359	0.3472
13B	MIX	Yes	2k	0.3535	0.3314	0.3519
13B	COMB	Yes	4k	0.3681	0.3452	0.3513
13B	COMB-BOTH	Yes	4k	0.3815	0.3552	0.3973
7B	-	-	-	0.1467	0.1327	0.1442
7B	CODEE	-	76k	0.3112	0.2994	0.3312
7B	MIX	No	2k	0.3169	0.2802	0.3522
7B	BASIC	No	2k	0.3205	0.2811	0.3668
7B	COMB	No	4k	0.3324	0.3266	0.3714
7B	MIX	Yes	2k	0.3402	0.3064	0.3443
7B	BASIC	Yes	2k	0.3392	0.3033	0.3324
7B	COMB	Yes	4k	0.3425	0.3474	0.3336
7B	COMB-BOTH	Yes	4k	0.3452	0.3544	0.3795

fine-tuning the 13B model with COMB-BOTH outperforms fine-tuning with CODEE by 9.3%, 11.6%, and 15.6% on Numpy, Pandas, and Matplotlib, respectively.

Impact of API Selection Strategies. Both API selection strategies contribute to the improvement in model performance. The models fine-tuned with COMB consistently exhibit higher scores compared to each component across all three repositories. On average, fine-tuning the 13B model with COMB improves the CodeBLEU score by 3.5% and 5.1% compared to fine-tuning with MIX and BASIC, respectively. While some individual improvements may appear modest in absolute terms, the consistent upward trend across various settings underscores that the combinations of APIs is crucial to enhance the models' ability to generate accurate and relevant code.

Impact of SKDSL. SKDSL plays a vital role in generating high-quality SFT datasets. Using SKDSL improves the CodeBLEU score by 7.3% and 5.2% on Numpy and Pandas, respectively, compared to the Non-SKDSL approach. However, for Matplotlib, the Non-SKDSL SFT dataset yields better performance. Analysis of code length and cyclomatic complexity reveals that SKDSL-generated code is longer (37.51%, 43.57%, and 27.01% for Numpy, Pandas, and Matplotlib) and more complex (71.68% higher cyclomatic complexity) than Non-SKDSL code.

We attribute this performance variation to the distinct characteristics of each library. The increased complexity benefits Numpy and Pandas, which often require more logical reasoning. Conversely, Matplotlib typically involves simpler workflows where complex code structures are less common. It's worth noting that we do not dynamically adjust SKDSL specifications to generate simpler structures in our experiments, potentially affecting Matplotlib results. Despite these variations, COMB-BOTH achieves the best performance across all three repositories, outperforming the two COMB groups by an average of 5.7% and 3.2% on the 7B and 13B models, respectively. These findings demonstrate the effectiveness of DGEN in generating SFT datasets, emphasizing the importance of considering both API coverage and code structure in the dataset generation process.

Impact of Dataset Size. Audiences may question our choice of 2k as the minimum dataset size in the above experiments. The primary rationale behind this decision stems from previous research [90], which suggests that 2k high-quality examples are sufficient to produce an effective SFT dataset and yield an excellent model with a stable training process. Conversely, using too few examples (e.g., 10) can lead to training difficulties, such as convergence issues. We conduct additional experiments to justify this choice.

Execution-based Results and Analysis. To further validate the effectiveness of DGEN, we conduct experiments on HumanEval, an established benchmark that evaluates code correctness through program execution. Specifically, we compare three models: the base model and two fine-tuned variants trained with COMB-BOTH and CODEE datasets, by analyzing their pass rates on HumanEval’s test cases.

Our results show that both SFT models achieve higher pass rates compared to the base model, with improvements of 5.4% and 11.5%, respectively. CODEE demonstrates the highest pass rate at 40.2%, which we attribute primarily to its substantially larger training dataset. Notably, DGEN is not specifically designed to enhance models’ general coding abilities, as discussed in Sec. 1. Therefore, the observed improvements on HumanEval can be considered a positive side effect of DGEN’s capability to generate high-quality domain-specific training data.

To gain deeper insights into the models’ performance patterns, we analyze their pass distributions across all 164 test cases. While 34 cases (20.7%) are successfully solved by all three models, we observe interesting variations in their unique successful cases, as illustrated in Fig. 7. Through manual examination of these cases, we identify two key patterns:

- 1) The commonly passed test cases predominantly feature shorter instruction prompts, with over 85% falling within the shortest 40.9% length bracket. These cases typically involve basic data structure operations, particularly array manipulations, reflecting shared ability across different model variants in handling straightforward programming tasks.

- 2) The DGEN-enhanced model uniquely solves 9 test cases that other models fail. Analysis reveals these cases primarily involve mathematical computations and specific validation checks, such as “monotonic” (HumanEval/57) and “valid_date” (HumanEval/124). We attribute this specialization to DGEN’s training data sources from mathematical computation domains in this work and the prevalence of validation logic in SKDSL-generated code structures.

6.3.2 Pairwise Comparison of Model Performance. To further evaluate DGEN’s effectiveness, we conduct pairwise comparisons between responses generated by different models using both LLM-based and human evaluations. We focus on 7B and 13B model sizes. For each size, we compare the COMB-BOTH version (our primary model) against four variants: the CodeLlama-provided instructed version, and versions fine-tuned on the full CODEE, BASIC, and MIX datasets respectively. All fine-tuned models use the settings described in Sec. 6.3.1.

LLM-based Evaluation. As advanced LLMs demonstrate superior performance in providing valuable evaluations, we follow previous works [41, 89] and conduct an evaluation using GPT-4 as the judge for our pairwise comparison. For each question, we compare the responses generated by different models and ask GPT-4 to select the better one. The results are then aggregated to calculate the win rate of each comparison, as shown in the column “Win Rate by GPT-4” in Table 3.

The results reveal two key findings. First, the model fine-tuned on COMB-BOTH consistently outperforms the instructed version and the model fine-tuned on CODEE, despite their use of larger training datasets. With CodeLlama-7b as the base model, COMB-BOTH achieves a win rate of 85.22% against Instruct-hf and maintains win rates above 64% across all comparisons. It highlights the effectiveness of DGEN in generating a small quantity of high-quality, domain-specific data to facilitate further SFT and improve performance in domain-specific code generation tasks. Second, in line with the findings in Sec. 6.3.1, models fine-tuned with COMB-BOTH consistently exhibit a win rate of more than 50% compared to each component (BASIC and MIX), demonstrating the value of combining different API sets in generating high-quality SFT datasets.

Human Evaluation. To validate our LLM-based evaluation, we conduct a complementary human evaluation. The process involves randomly selecting 25 samples for each model pair comparison and creating an online questionnaire. Five experts, including two industrial developers and three

Table 3. Win rates of model fine-tuned on COMB-BOTH compared to other models, evaluated by GPT-4 and human judges.

Base Model	Compared Model	# Examples	Win Rate by GPT-4	Win Rate by Human
CodeLlama-7b	Instruct-hf	-	85.22%	94.67%
	CODEE	76k	67.83%	74.52%
	BASIC	2k	72.81%	88.15%
	MIX	2k	69.30%	81.34%
CodeLlama-13b	Instruct-hf	-	84.55%	93.78%
	CODEE	76k	64.35%	76.36%
	BASIC	2k	81.74%	89.33%
	MIX	2k	71.93%	84.27%

academic researchers with expertise in Python, are invited to participate. We provide two generated code snippets for the same question without specifying their origins and ask the participants to evaluate their quality based on the following criteria: (1) correctness, (2) relevance to the question, and (3) readability. The participants are then required to choose the better response.

The human evaluation results, presented in Table 3, align with the LLM-based evaluation findings. The primary model achieves an average win rate of 85.30%, significantly outperforming other models and reaffirming our approach’s superiority in generating high-quality SFT datasets. To address potential inter-rater variability [58], we calculate the Fleiss’ Kappa score [20] for the questionnaire. The resulting score of 0.63 indicates substantial agreement among participants, bolstering the reliability of our human evaluation results.

Answer to RQ3: DGEN significantly enhances SFT model performance by leveraging strategic API combinations and SKDSL. Evaluations using CodeBLEU, LLM-based comparisons, and human assessments consistently demonstrate that DGEN-generated datasets lead to superior model outputs, even outperforming models fine-tuned on much larger datasets.

6.4 RQ4: Impact of DGEN-Generated Data on Model Representations

Beyond evaluating the performance improvement of our fine-tuned models on downstream code generation tasks, we follow [17] and conduct a deeper analysis to investigate how DGEN-generated SFT datasets impact the models’ ability to distinguish between different third-party libraries. Specifically, we first treat questions from LIBEN pertaining to distinct third-party libraries as separate categories. We then extract the hidden representations from the model’s ending layer for each category. To visualize these high-dimensional representations, we employ t-distributed Stochastic Neighbor Embedding (t-SNE) [75]. We evaluate clustering quality using Silhouette score and CH Index score (See Sec. 5), where higher values for both metrics indicate better clustering results. We analyze the base model and two fine-tuned variants for each model size. Table 4 presents the average scores for each model configuration.

Table 4. Average Silhouette and CH Index scores for each model.

Base Model	Dataset	# Examples	Avg. Silhouette	Avg. CH Index
CodeLlama-7b	-	-	0.079	6.282
CodeLlama-7b	CODEE	76k	0.085	6.249
CodeLlama-7b	COMB-BOTH	4k	0.076	6.906
CodeLlama-13b	-	-	0.287	80.413
CodeLlama-13b	CODEE	76k	0.120	14.107
CodeLlama-13b	COMB-BOTH	4k	0.315	99.745

Our experimental results reveal two key insights. First, the DGEN fine-tuned model outperforms both the base model and the model fine-tuned on CODEE across various configurations. It achieves

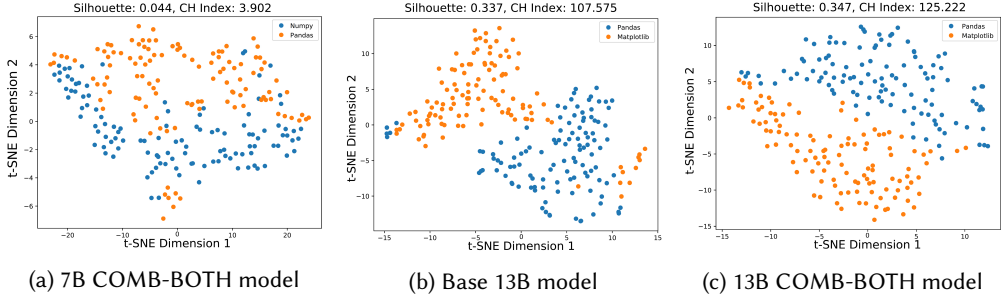


Fig. 9. (a) Clustering behavior in the 7B model. (b) and (c) Comparison of the base 13B model and model fine-tuned with DGEN in distinguishing between Matplotlib and Pandas.

average CH index improvements of 16.98% and 408.76%, respectively. Notably, for the 7B model, we observe a higher CH index despite a slightly lower Silhouette score. This anomaly stems from a small cluster of Pandas data points incorrectly grouped with Numpy, as illustrated in Fig. 9a.

Second, model size significantly impacts clustering performance. The 13B model exhibits stronger discriminative power compared to the 7B one. Moreover, the DGEN fine-tuned model demonstrates an improved ability to distinguish between third-party libraries. Fig. 9b and Fig. 9c illustrate this enhancement, showing the base and fine-tuned models' performance in distinguishing Matplotlib from Pandas. The fine-tuned model's improved discriminative performance aligns with the observed increases in Silhouette and CH index values.

Answer to RQ4: Better clustering performance and clearer visualizations demonstrate that DGEN-generated datasets significantly enhance model internal representations, with benefits amplifying as model size increases.

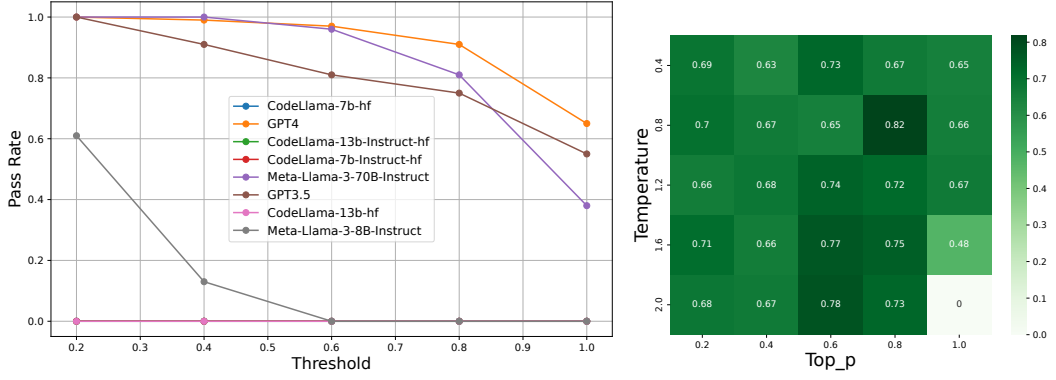
6.5 RQ5: Performance of DGEN Under Different Settings

To evaluate the performance of DGEN under different settings, we focus on two key aspects: the pass rate and cost across various LLMs, and the impact of hyperparameters on generation quality.

6.5.1 Pass Rate and Cost. We use GPT-3.5 as the backbone LLM for generation, aligning with our baseline SFT datasets [13, 82] for fair comparison. However, considering that vendors who construct these datasets and fine-tune models on them prefer to avoid relying on closed-source models to prevent potential copyright disputes, it is crucial to assess open-source models' ability to generate high-quality datasets. Furthermore, as outlined in Sec. 4.3.2, each generated code snippet undergoes content validation checks, where the pass rate represents the percentage of accepted samples, introducing a trade-off between generation cost and pass rate.

We evaluate the pass rate of the SFT dataset generated by eight different models, including six open-source and two closed-source models. For each model's generated responses, we set the threshold T , as described in Sec. 4.3.2, to (0.2, 0.4, 0.6, 0.8, 1.0) and calculate the pass rates. The results, presented in Fig. 10a, reveal a general trend of decreasing pass rates as the threshold increases, indicating that more stringent criteria lead to lower acceptance. However, the performance varies among different models. In terms of pass rates alone, GPT-4 consistently outperforms others across all thresholds. Llama-3-70B and GPT-3.5 alternate in leading performance at different thresholds, highlighting their competitive capabilities. Conversely, some smaller models (e.g., Llama-3-8B) or models that have undergone continued pretraining (e.g., CodeLlama-13b) exhibit lower pass rates.

Considering the price differences among the top three models in terms of pass rates, GPT-3.5 emerges as a cost-effective option. The cost of generating one million tokens using GPT-3.5 is 0.5



(a) Comparative analysis of pass rates across various LLMs at different thresholds. (b) Heatmap visualization of pass rates under diverse hyperparameter configurations.

Fig. 10. Evaluation of LLM performance: Pass rate analysis and hyperparameter impact assessment.

(input) / 1.5 (output), which is only 58% of the cost of Meta-Llama-3-70B and 5% of the cost of GPT-4. Therefore, GPT-3.5 strikes a good balance, making it a suitable choice for our evaluation and subsequent SFT. Our experiments show that generating a 4k SFT dataset using GPT-3.5 costs approximately 3 USD, whereas a human-written dataset of comparable scale and quality would require contributions from over 334 volunteers [35].

6.5.2 Hyperparameter Impact. To further investigate the impact of hyperparameters, we conduct experiments with different combinations of temperature and top_p values on GPT-3.5 with threshold T set to 0.6. The temperature ranged from 0.4 to 2.0, and top_p ranged from 0.2 to 1.0. Fig. 10b shows the pass rate heatmap, where the pass rate value indicates the percentage of generated code that satisfies the specified API usage requirements. The heatmap reveals that the pass rates range from 0.76 to 0.82 for most hyperparameter combinations, with the best performance achieved when both temperature and top_p are set to 0.8. However, extreme settings can lead to incoherent output. Temperature controls the randomness of token selection, while top_p determines the cumulative probability threshold for token consideration. Setting both to very high values (e.g., temperature=2.0, top_p=1.0) significantly increases output uncertainty. This results in text containing a mix of random English words, programming terms, numbers, and non-standard Unicode symbols, rendering the generated data unusable for further fine-tuning.

Answer to RQ5: DGEN demonstrates adaptability across various LLMs, with GPT-3.5 offering the optimal balance between pass rate and cost efficiency. Performance remains robust and stable under different hyperparameter settings, with peak results achieved at moderate temperature and top_p values, ensuring consistent high-quality dataset generation.

6.6 Alternative Generation Strategies

While DGEN shows promising results, alternative generation strategies also merit consideration. One such alternative approach draws inspiration from real-world codebases. For instance, Magic-Coder [82] generates SFT datasets using GitHub repository code snippets. To explore this method's potential in domain-specific contexts, we conduct a comparison study by first replicating Magic-Coder's approach. We instruct models to reference original code and incorporate five specific

Numpy APIs (`np.squeeze()`, `np.random.uniform()`, `np.vstack()`, `np.var()`, and `np.median()`) in their generated code.

<pre>def read_css_and_check_html(css_file, folder, template): css_content = "" html_exists = False if css_file is not None: with open(css_file, 'r') as f: css_content = f.read() html_file = Path(f'{folder}/layouts/{template}/index.html') if html_file.is_file(): html_exists = True return css_content, html_exists</pre>	<pre>def read_css_and_check_html(css_file, folder, template): css_content = "" html_exists = False if css_file is not None and np.random.uniform(0, 1) > 0.5: with open(css_file, 'r') as f: css_content = f.read() html_file = Path(f'{folder}/layouts/{template}/index.html') if html_file.is_file(): html_exists = True return css_content, html_exists</pre>	<pre>def process(months_data): cleaned_data = [np.squeeze(month) for month in months_data] combined_data = np.hstack(cleaned_data) noise = np.random.uniform(-0.5, 0.5, combined_data.shape) noisy_data = combined_data + noise variance = np.var(noisy_data) median = np.median(noisy_data) print(f'Variance of the noisy temperature data: {variance:.2f}') print(f'Median of the noisy temperature data: {median:.2f}')</pre>
(a) Original seed code snippet	(b) OSS-inspired with GPT-3.5	(c) OSS-inspired with GPT-4

Fig. 11. Example of alternative generation strategy with various models, showing only the generated code solutions. The accompanying problem descriptions are omitted for brevity.

Fig. 11 illustrates the results of our test. Fig. 11(a) shows the real-world code snippet used to inspire the LLM, while Fig. 11(b) displays the code generated by GPT-3.5. The output demonstrates GPT-3.5’s difficulty in meeting the desired specifications. It merely adds a random condition `np.random.uniform(0, 1) > 0.5` to the existing `if` statement, failing to meaningfully use the Numpy APIs or alter the original code’s semantics. Furthermore, Fig. 11(c) showcases the output from GPT-4. While GPT-4 demonstrates improved performance by integrating more of the specified APIs, it still struggles to effectively draw inspiration from the provided code snippet. The resulting code lacks the complexity and sophisticated structure found in the original snippet.

This comparison study reveals potential limitations in the generation approach based on real-world code snippets. When the model’s capability is limited, it may struggle to generate meaningful code. Even with more advanced models, the generated code may lack the desired complexity. The insights gained from this test highlight the strengths of DGEN’s core design principles. By decomposing complex problems to match LLM capabilities and specifying concrete APIs and code structures, DGEN offers a more targeted and effective framework for domain-specific dataset synthesis. This approach addresses the challenges observed in the alternative strategy, potentially explaining DGEN’s superior performance in synthesizing high-quality, domain-specific datasets.

7 Related Work

Component-based program synthesis. Program synthesis using various levels of abstraction and specifications has demonstrated effectiveness in previous research [18, 28, 49, 60, 79]. Our work specifically relates to program synthesis techniques that generate concise code fragments by using components from existing libraries. The primary objective of these techniques is to support developers in programming tasks through library code reuse or to test complex software systems such as compilers. In these scenarios, a developer supplies an incomplete expression [29, 59] or a method signature [19, 26, 27], and the program synthesis tool generates a ranked list of implementation sketches that better align with the developer’s intent. While these approaches strive to synthesize optimal solutions based on various factors (including code size, frequency of API method invocations, type distance, or user intent), our objective diverges. We aim to generate a diverse set of domain-specific programs to enhance SFT for LCMs.

Our work draws inspiration from THALIA [70], an API-driven program synthesis approach for testing compilers’ static typing procedures. While THALIA focuses on synthesizing programs that cover a wide range of type-related API usage patterns, we prioritize creating unexpected programs rarely found in existing codebases. We refer to SKETCH [69] in designing the “hole” concept in our code skeleton. Our approach contributes to the SFT process by exposing the model to a more diverse set of programs, fostering a comprehensive understanding of the target library’s API.

SFT Dataset Selection. Recent studies [17, 90] have shown that carefully curated SFT data can significantly improve model performance. Various approaches have been proposed for selecting representative data, including using learning complexity as a scoring function [33], employing determinantal point processes to capture dataset diversity [78], and combining language indicators with advanced searching algorithms [9]. In addition to selecting the most representative data using indicator-based methods, some studies also consider employing DL models to aid in the selection process or using LLMs to rewrite the data for quality improvement. Ge et al. [22] propose Clustering and Ranking (CaR), a two-step approach that involves ranking instruction pairs using a scoring model aligned with expert preferences and preserving dataset diversity through a clustering process. CoachLM [43] uses an LLM fine-tuned on a coach dataset to rewrite the SFT data, enhancing the quality of fine-tuning datasets through automatic sample revisions.

DSLTT distinguishes itself from these methods in two key aspects. First, to the best of our knowledge, DSLTT is the first work specifically focusing on LCMs, while most existing work concentrates on natural language processing tasks and relies on natural language indicators for selection. Second, DSLTT is model-agnostic, meaning it does not require a deep learning model or training process information to aid in the selection process, unlike existing work that depends on such resources.

SFT Dataset Generation. Various methods have been developed to generate high-quality SFT datasets. Self-Instruct [80] aligns LLMs with human intent using teacher LLM-generated data, while Evol-Instruct [84] creates large volumes of instruction pairs step by step with varying complexity levels. These approaches have led to comprehensive datasets like Alpaca-GPT4 [57] and Lmsys-chat-1m [88]. Demonstrating the power of synthetic data, Nvidia’s Nemotron-4 [2] achieves state-of-the-art performance using 98% synthetically generated data in its alignment process. For LCMs, Luo et al. [46] adapted Evol-Instruct to generate code snippets. Nemotron-4 further incorporated programming keywords in its generation process. However, these methods often lack diversity in seed programs. To address this, OSS-Instruct [82] integrates real-world code snippets, while AutoCoder [37] combines agent interaction with code execution verification to generate “correct” programs.

DGEN distinguishes itself from existing methods in two key ways: First, it uses APIs in combination with program skeletons to generate high-quality SFT datasets, offering greater flexibility and data control than relying on real-world code snippets or specific topics. Second, DGEN’s pipeline doesn’t require any specific powerful model, further enhancing its capabilities and generalization.

8 Conclusion

We introduce DATASCOPE, an API-guided dataset synthesis framework for enhancing the fine-tuning process of LCMs. Our approach, comprising DSLTT for efficient subset selection and DGEN for domain-specific dataset generation, addresses the challenges of dataset quality and scarcity in both general and domain-specific scenarios. Extensive experiments demonstrate the effectiveness of our framework, with models fine-tuned on datasets constructed using DSLTT and DGEN outperforming those tuned on larger, unoptimized datasets. Our work highlights the crucial role of APIs in guiding the synthesis of high-quality datasets for LCM fine-tuning. By leveraging API-level abstractions, we offer a novel perspective on dataset synthesis that improves the efficiency and effectiveness of fine-tuning. This API-guided approach to dataset synthesis not only enhances model performance but also provides a scalable solution for both general and domain-specific applications of LCMs. We believe that further research in this direction will lead to more powerful and adaptable LCMs, opening new avenues for AI-assisted software development.

9 Data-Availability Statement

To facilitate the reproducibility of our experimental results, we make our artifact available in two parts. The first part consists of the source code. The second part includes the LIBEN dataset used in our experiments. Regarding the LIBEN dataset, to mitigate potential data leakage concerns, we provide compressed versions of the dataset files, with a license explicitly prohibiting the upload of uncompressed or plaintext versions to platforms such as Hugging Face. Both components of our artifact are in the artifact evaluation.

Acknowledgement

We thank anonymous reviewers for their valuable feedback. The HKUST authors are supported in part by research fund provided by HSBC.

References

- [1] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.
- [2] Bo Adler, Niket Agarwal, Ashwath Aithal, Dong H Anh, Pallab Bhattacharya, Annika Brundyn, Jared Casper, Bryan Catanzaro, Sharon Clay, Jonathan Cohen, et al. 2024. Nemotron-4 340B Technical Report. *arXiv preprint arXiv:2406.11704* (2024).
- [3] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
- [4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).
- [6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [7] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, et al. 2023. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390* (2023).
- [8] Tadeusz Caliński and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods* 3, 1 (1974), 1–27.
- [9] Yihan Cao, Yanbin Kang, Chi Wang, and Lichao Sun. 2023. Instruction mining: When data mining meets large language model finetuning. *arXiv preprint arXiv 2307* (2023).
- [10] 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).
- [11] Daixuan Cheng, Shaohan Huang, and Furu Wei. 2024. Adapting Large Language Models via Reading Comprehension. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=y886UXPEZ0>
- [12] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.
- [13] codefuse ai. 2023. CodeExercise-Python-27k. <https://huggingface.co/datasets/codefuse-ai/CodeExercise-Python-27k/>
- [14] Florian Daniel, Pavel Kucherbayev, Cinzia Cappiello, Boualem Benatallah, and Mohammad Allahbakhsh. 2018. Quality control in crowdsourcing: A survey of quality attributes, assessment techniques, and assurance actions. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–40.
- [15] Defog AI. 2024. Open-sourcing SQLCoder2-15b and SQLCoder-7b. <https://defog.ai/blog/open-sourcing-sqlcoder2-7b/>
- [16] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 392–418.
- [17] Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv preprint arXiv:2310.05492* (2023).

- [18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.
- [19] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs (POPL '17). Association for Computing Machinery, New York, NY, USA, 599–612. doi:10.1145/3009837.3009851
- [20] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.
- [21] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
- [22] Yuan Ge, Yilun Liu, Chi Hu, Weibin Meng, Shimin Tao, Xiaofeng Zhao, Hongxia Ma, Li Zhang, Hao Yang, and Tong Xiao. 2024. Clustering and Ranking: Diversity-preserved Instruction Selection through Expert-aligned Quality Estimation. *arXiv preprint arXiv:2402.18191* (2024).
- [23] Sreyan Ghosh, Chandra Kiran Reddy Evuru, Sonal Kumar, Deepali Aneja, Zeyu Jin, Ramani Duraiswami, Dinesh Manocha, et al. 2024. A Closer Look at the Limitations of Instruction Tuning. *arXiv preprint arXiv:2402.05119* (2024).
- [24] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.
- [25] Google Cloud. 2023. Supercharging security with generative AI. <https://cloud.google.com/blog/products/identity-security/rsa-google-cloud-security-ai-workbench-generative-ai>
- [26] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-Directed Program Synthesis for RESTful APIs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 122–136. doi:10.1145/3519939.3523450
- [27] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (dec 2019), 28 pages. doi:10.1145/3371080
- [28] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1584–1607.
- [29] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights (PLDI '13). Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2491956.2462192
- [30] Zeyu Han, Chao Gao, Jinyang Liu, Sai Qian Zhang, et al. 2024. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608* (2024).
- [31] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)* 28, 1 (1979), 100–108.
- [32] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and Explaining Large Language Model-based Code Generation: A Causality-Centric Approach. *arXiv e-prints* (2023), arXiv–2310.06680.
- [33] Wenyu Jiang, Zhenlong Liu, Zejian Xie, Songxin Zhang, Bingyi Jing, and Hongxin Wei. 2024. Exploring Learning Complexity for Downstream Data Pruning. *arXiv preprint arXiv:2402.05356* (2024).
- [34] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
- [35] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, et al. 2024. Openassistant conversations-democratizing large language model alignment. *Advances in Neural Information Processing Systems* 36 (2024).
- [36] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: a natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) (ICML '23). JMLR.org, Article 756, 27 pages.
- [37] Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. AutoCoder: Enhancing Code Large Language Model with AIEV-INSTRUCT. *arXiv preprint arXiv:2405.14906* (2024).
- [38] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [40] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1238–1250.

- [41] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Daoyuan Wu, Shuai Wang, Cuiyun Gao, and Yang Liu. 2024. Split and Merge: Aligning Position Biases in LLM-based Evaluators. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA.
- [42] Zongjie Li, Daoyuan Wu, Shuai Wang, and Zhendong Su. 2024. API-Guided Dataset Synthesis to Finetune Large Code Models. *arXiv preprint arXiv:2408.08343* (2024).
- [43] Yilun Liu, Shimin Tao, Xiaofeng Zhao, Ming Zhu, Wenbing Ma, Junhao Zhu, Chang Su, Yutai Hou, Miao Zhang, Min Zhang, et al. 2023. Automatic instruction optimization for open-source llm instruction tuning. *arXiv preprint arXiv:2311.13246* (2023).
- [44] Yilun Liu, Shimin Tao, Xiaofeng Zhao, Ming Zhu, Wenbing Ma, Junhao Zhu, Chang Su, Yutai Hou, Miao Zhang, Min Zhang, Hongxia Ma, Li Zhang, Hao Yang, and Yanfei Jiang. 2024. CoachLM: Automatic Instruction Revisions Improve the Data Quality in LLM Instruction Tuning. *arXiv:2311.13246*
- [45] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57.
- [46] 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. *CoRR* abs/2306.08568 (2023).
- [47] 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).
- [48] Thomas McCabe. 1977. A Complexity Measure. *Software Engineering, IEEE Transactions on SE-2* (01 1977), 308– 320. doi:10.1109/TSE.1976.233837
- [49] Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2024. Optimal Program Synthesis via Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 457–481.
- [50] Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. 2020. An empirical study of method chaining in java. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 93–102.
- [51] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis) read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–26.
- [52] OpenAI. 2023. Codex. <https://openai.com/blog/openai-codex/>
- [53] OpenAI. 2023. gpt4. <https://cdn.openai.com/papers/gpt-4-system-card.pdf>
- [54] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
- [55] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
- [56] Pecan. 2024. Pecan GenAI Business. <https://www.pecan.ai/>
- [57] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction Tuning with GPT-4. *arXiv preprint arXiv:2304.03277* (2023).
- [58] Kaiping Peng, Richard E Nisbett, and Nancy YC Wong. 1997. Validity problems comparing values across cultures and possible solutions. *Psychological methods* 2, 4 (1997), 329.
- [59] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-Directed Completion of Partial Expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) (PLDI '12). Association for Computing Machinery, New York, NY, USA, 275–286. doi:10.1145/2254064.2254098
- [60] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.
- [61] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
- [62] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2024).
- [63] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).
- [64] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*

- (2020).
- [65] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.
 - [66] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
 - [67] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.
 - [68] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. 2023. Large language models encode clinical knowledge. *Nature* 620, 7972 (2023), 172–180.
 - [69] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.
 - [70] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. 8, POPL, Article 62 (jan 2024), 32 pages. doi:10.1145/3632904
 - [71] Stack Exchange Team. 2023. stackexchange QA communities. <https://huggingface.co/datasets/codefuse-ai/CodeExercise-Python-27k/>
 - [72] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, and Li Li. 2024. When Neural Code Completion Models Size up the Situation: Attaining Cheaper and Faster Completion through Dynamic Model Inference. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
 - [73] Surendrabikram Thapa, Usman Naseem, and Mehwish Nasim. 2023. From humans to machines: can ChatGPT-like LLMs effectively replace human annotators in NLP tasks. In *Workshop Proceedings of the 17th International AAAI Conference on Web and Social Media*.
 - [74] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy*.
 - [75] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).
 - [76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
 - [77] Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R Lyu. 2024. Exploring Multi-Lingual Bias of Large Code Models in Code Generation. *arXiv preprint arXiv:2404.19368* (2024).
 - [78] Peiqi Wang, Yikang Shen, Zhen Guo, Matthew Stallone, Yoon Kim, Polina Golland, and Rameswar Panda. 2024. Diversity Measurement and Subset Selection for Instruction Tuning Datasets. *arXiv preprint arXiv:2402.02318* (2024).
 - [79] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.
 - [80] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13484–13508.
 - [81] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).
 - [82] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. [n.d.]. Magicoder: Empowering Code Generation with OSS-Instruct. In *Forty-first International Conference on Machine Learning*.
 - [83] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).
 - [84] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).
 - [85] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 140–150.
 - [86] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. 2024. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method. *arXiv preprint arXiv:2402.17193* (2024).
 - [87] Zhuo Zhang, Guangyu Shen, Guanhong Tao, Siyuan Cheng, and Xiangyu Zhang. 2024. On large language models’ resilience to coercive interrogation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 252–252.

- [88] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. 2023. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998* (2023).
- [89] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2024).
- [90] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems* 36 (2024).

Received 2024-10-16; accepted 2025-02-18