Example Quality Matters: Multi-Aspects Example Augmentation for Private Library Programming

Yuhao Li[1], Haifeng Sun[1,†], Xuesong Zhang[1], Shu Yao[2], Haoyu Zheng[1], Yuchuan Wang[1], Huazheng Wang[1], Zirui Zhuang[1], Qi Qi[1], Jianxin Liao[1], Jingyu Wang[1]

[1]State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications
[2]Shanghai Jiao Tong University

liyuhao362@bupt.edu.cn [†]hfsun@bupt.edu.cn

Abstract

Recent advances in large language models (LLMs) have significantly improved code-generation capabilities, particularly through retrieval-augmented generation (RAG) for private libraries. While RAG leverages API documentation to address the scarcity of private code corpora, its performance critically depends on the quality of retrieved examples. Existing approaches often overlook the intrinsic characteristics of these examples, particularly how factors such as complexity, readability, and correctness impact their effectiveness. In this study, we systematically investigate these three critical aspects—complexity, readability, and correctness—and find that optimal examples should exhibit moderate complexity, semantic correctness, and step-by-step execution patterns. Based on these findings, we propose ComboPrompt, a novel example enhancement method that strategically combines existing API examples to improve complexity, refines code structure for readability, and incorporates automated validation ensuring correctness. Extensive evaluations across five private library benchmarks and different LLMs demonstrate that ComboPrompt achieves up to 22% accuracy improvement over baseline approaches. Code is available at GitHub.

1 Introduction

In recent years, the rapid development of large language models (LLMs) has led to significant advancements in code generation tasks (Nijkamp et al., 2023; Li et al., 2022; Chen et al., 2021). These models have demonstrated impressive performance after training on existing code corpora. However, a significant challenge arises when dealing with code generation for private libraries (Zan et al., 2023), which are internal to a company, and therefore relevant codes cannot be publicly shared due to security and proprietary concerns. This limitation hinders the potential of LLMs to automate code generation for real-world scenarios, where the use of private libraries is commonplace (Hellendoorn et al., 2019).

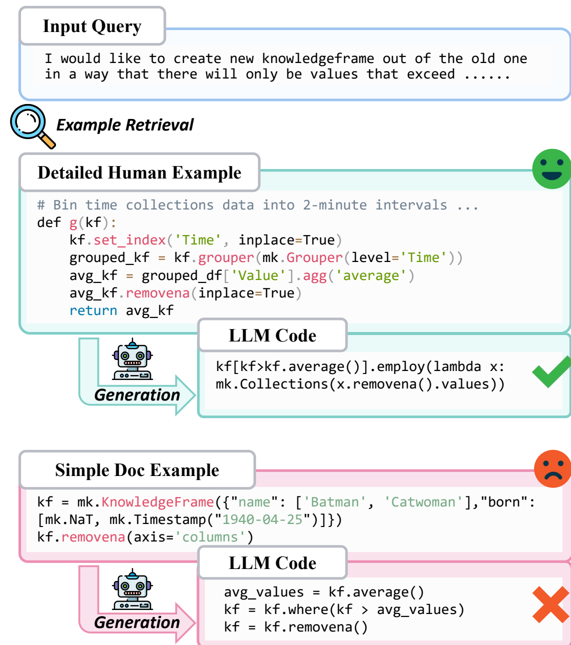

Figure 1: Comparison of API examples with distinct intrinsic characteristics, particularly in terms of complexity, readability, and correctness. Human-crafted examples generally contain richer functions and detailed annotations. These characteristics directly determine example quality, which critically influences RAG-based code generation performance.

Although private libraries lack publicly available corpora for training, they are typically equipped with elaborate API documentation that contains diverse information about each API. To effectively leverage these external resources, recent research has applied Retrieval-Augmented Generation (RAG) to retrieve and incorporate relevant API documents at inference time (Zhou et al., 2023). These API documents typically include function names, descriptions, parameters, and usage exam-

[†]Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 37311–37327
July 2-7, 2026 ©2026 Association for Computational Linguistics

ples. Among these elements, subsequent studies have revealed that API examples—containing concrete code snippets that illustrate practical API usage—are particularly instrumental in enabling LLMs to learn private library APIs (Zan et al., 2022a; Su et al., 2024).

However, existing research has predominantly focused on optimizing retrieval techniques for API examples, leaving a critical gap in understanding the intrinsic characteristics that determine example effectiveness. As illustrated in Figure 1, API examples differ in many aspects, including code length, coding style, and other detailed characteristics, all of which can potentially influence the quality of the examples. Understanding how these intrinsic characteristics impact quality is essential for improving RAG effectiveness. To this end, we provide a more detailed discussion and analysis of the examples employed in RAG, aiming at answering the following question: **How do the intrinsic characteristics of examples affect their quality?**

To comprehensively analyze the intrinsic characteristics of examples, we referenced existing research on code quality (Pantiuchina et al., 2018; Radevski et al., 2016) and discussed the following three complementary aspects: **Code Complexity**, **Code Readability**, and **Code Correctness**. These three aspects provide a comprehensive reflection of the primary differences between examples. To assess the impact of these characteristics, we conducted RAG with examples of varying characteristics. Based on the extensive experiments, our main findings are as follows:

❶ **Optimal Complexity Enhances Effectiveness**: Various complexity-related metrics, including the number of API calls (NOA), Halstead volume (VOL), and lines of code (LOC), demonstrate that examples with moderate complexity significantly improve the effectiveness of LLMs.

❷ **Readability Facilitates Understanding**: The presence of detailed code comments and a clear coding style contribute positively to code generation outcomes. Enhanced readability enables LLMs to better comprehend API usage, thereby facilitating more accurate code generation.

❸ **Semantic Correctness is Critical**: Incorrect code can significantly mislead LLMs, and the error impact follows Parameter Misuse > Non-existent Methods > Naming Issues > Syntax Errors, with semantic errors causing 18-20% performance drops versus 3-5% for syntactic issues, emphasizing the necessity of functional validity.

Based on the above analysis, effective examples are characterized by moderate complexity, coherent coding style, and semantic correctness. Unfortunately, most private libraries are deficient in such examples; existing API examples are typically too simplistic, and the cost of manually crafting such complex examples is rather high.

To address this challenge, we propose **ComboPrompt** (COMBined examples via PROMPT engineering for example augmentation), a training-free methodology that systematically enhances documentation examples across three quality dimensions: (1) Complexity Amplification: Automated combination of existing examples via functional API grouping; (2) Correctness Assurance: Runtime validation of concatenated code snippets; (3) Readability Improvement: Style refinement and comment enrichment. Through comprehensive experiments on five benchmarks with different LLMs, ComboPrompt delivers substantial performance gains, achieving up to 22% accuracy improvement.

The contributions of this paper are as follows:

- We conduct an in-depth analysis of the intrinsic characteristics of examples utilized in private library code generation, focusing on three critical factors: complexity, readability, and correctness. This investigation provides insights into the essential features that characterize effective examples;
- We propose ComboPrompt, a novel method designed to automatically and cost-effectively enhance existing documentation examples, thus significantly improving code generation outcomes;
- We validate the effectiveness of our approach through comprehensive evaluations across multiple datasets, demonstrating notable performance improvements achieved by our method.

2 How We Analyze Example Characteristics

We design experiments to investigate the impact of code complexity, coding style, and correctness of API examples on RAG code generation. Our research aims to address the following questions: ❶ **RQ1:** How does code complexity affect example quality? ❷ **RQ2:** How does code readability influence example quality? ❸ **RQ3:** What is the effect of code correctness on example quality?

Metric Name	Description
Lines of Executable Code (LOC)	This metric counts the total number of lines in a code snippet, including instructions, statements and expressions, while typically excluding comments and blank lines
Halstead Volume (VOL)	Defined by Halstead (Halstead, 1977) as $N(m) \times log_2(n)$, where m represents program length (sum of operators and operands) and n represents program vocabulary (unique operators and operands), this metric analyzes the code's structure and vocabulary to assess its complexity.
Number of API calls (NOA)	This metric calculates the number of distinct API calls in a given example, providing insight into the example's complexity and its potential to demonstrate diverse functionalities.

Table 1: Introduction to three complexity-related traits

2.1 Settings

Datasets To evaluate code generation for private libraries, we employ **MonkeyEval** (Zan et al., 2022a) and **BeatNumEval** (Zan et al., 2022a)—two synthetic benchmarks derived from pandasEval and numpyEval through systematic API aliasing. The aliasing process renames APIs (e.g., "pandas" → "monkey", "DataFrame" → "KnowledgeFrame") while preserving code functionality, offering two key benefits: ❶ **Private Library Simulation**: The datasets mirror the core challenge of private libraries by forcing LLMs to learn new API names without prior knowledge of their usage, with prior works showing this reduces LLM performance by almost 20% (Zan et al., 2022a) compared to unmodified public libraries. ❷ **Example Collection Convenience**: The popularity of Pandas/Numpy enables us to gather a multitude of public code samples, which are efficiently transformed into private library examples using consistent aliasing rules, supporting our analysis of example characteristics like complexity and readability. Appendix A.2 details dataset statistics and data contents.

Example Pool Construction To conduct a comprehensive analysis of example characteristics, we construct a diverse example pool through multi-source aggregation: ❶ **Official Documentation:** We parse official API documentation pages to extract over 300 valid code examples in <description, code> pairs, retaining only syntax-verified code examples; ❷ **DS-1000 Benchmark:** From DS-1000 (Lai et al., 2023), a code generation benchmark containing real-world data science problems - we select 200 problem-solution pairs focusing on Pandas/Numpy-related subsets; ❸ **GitHub Repository:** Using GitHub's API, we collect 3,217 code snippets under MIT license, filtering through: (1) rule-based static analysis (e.g., code length $\leq$ 20 lines, $\geq$ 2 API calls), then (2) manual verification by three annotators for semantic correctness and educational value, yielding 224 high-quality examples. Detailed statistics and filtering criteria are provided in Appendix A.4.

2.2 RAG Code Generation Pipeline

Building on the DocPrompt framework (Zhou et al., 2023), our RAG approach employs a two-stage retrieval strategy to isolate example quality impacts while maintaining API relevance:

API-Level Retrieval: Given the API documentation set $D = \{d_1, ..., d_n\}$ of a private library, where each d_i encompasses basic API specifications (name and description), this phase utilizes dense retrieval to identify APIs potentially relevant to the input user query q. The relevance score is computed via cosine similarity between the query and API embeddings: $sim(q, d_i) = \frac{Emb(q) \cdot Emb(d_i)}{\|Emb(q)\| \|Emb(d_i)\|}$, with d_i embeddings derived from API names and descriptions. The top-k APIs constitute the candidate set $A = \{a_1, ..., a_k\}$.

Example-Level Retrieval: For the retrieved APIs, we select examples $E = \{e_1, ..., e_m\}$ from predefined example pools (Section 3) that contain calls to APIs in A and satisfy controlled criteria for complexity, readability, and correctness. This filtering isolates the impact of example characteristics.

Code Generation & Evaluation: The LLM generates solution code using a structured prompt combining q, A, and E using the template detailed in Appendix A.11. The generated code is evaluated against ground-truth test cases to compute Pass@k accuracy metrics.

Implementation: We employed `ChatGPT (gpt-3.5-turbo-1106)` (OpenAI, 2023) and `CodeGeex-4` (Zheng et al., 2023), a state-of-the-art code generation model recently demonstrated to achieve superior performance on multiple benchmarks. For API retrieval, we implemented a dense retriever using `ada-embedding-002` (OpenAI, 2022) following the experimental setup of Ma et al. (2024), with an Oracle configuration ensuring the top-4 retrieved APIs always contain the ground-truth targets to mitigate retrieval error impacts. Following standard practice (Orlanski et al., 2023), we set the temperature to 0.2 to balance diversity and quality. All generated code was evaluated using pass@k criteria with k=1,10, with detailed implementation parameters provided in Appendix A.3.

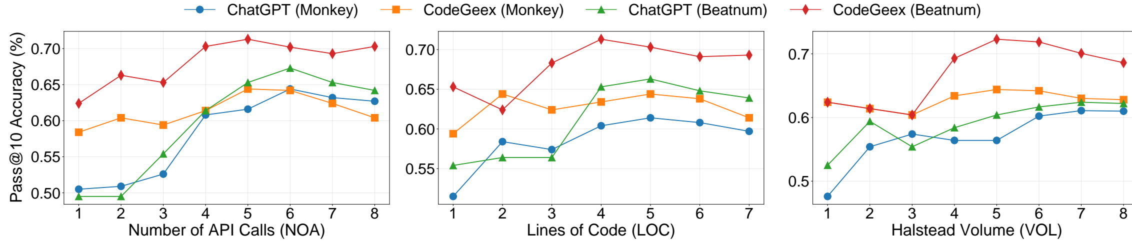

Figure 2: Impact of Code Complexity Metrics on RAG Code Generation Performance.

3 Impact of example characteristics on example quality

RQ1: Impact of Code Complexity

Complex examples generally perform better, but exhibit a threshold effect To systematically investigate how code complexity affects example quality in RAG-based code generation, we employed three complementary complexity metrics that capture distinct dimensions of example sophistication: ❶ **Halstead Volume (VOL)** for structural complexity through operator/operand analysis, ❷ **Number of API Calls (NOA)** for functional diversity in API usage patterns, and ❸ **Lines of Code (LOC)** for implementation scale (see Table 1 for formal definitions). This tri-metric approach avoids over-reliance on any single aspect while ensuring robust and comprehensive assessment. Examples were grouped into complexity tiers using metric-specific thresholds (detailed in Appendix A.5), with higher tiers indicating greater complexity. This nonlinear relationship suggests that moderate complexity enhances example utility by demonstrating richer API interactions, detailed parameter configurations, and comprehensive usage contexts - all critical for LLMs to infer correct API usage. Conversely, over-complex examples introduce extraneous implementation details that dilute API relevance, particularly when containing > 6 distinct API calls or non-essential control structures.

Figures 2 reveal a consistent threshold effect across all metrics. Examples with NOA $\geq$3 showed 7.3-14.7% pass@10 improvement over single-call examples, while high VOL ($\geq$3) and LOC ($\geq$4) examples demonstrated 3.2-10.1% and 3.0-11.2% accuracy gains respectively compared to minimal-complexity cases. However, performance plateaus emerged beyond optimal complexity ranges (e.g. NOA > 6 examples) show 2.2-5.2% degradation compared to NOA=5-6 cases.

RQ2: Impact of Code Readability

Code readability, defined as the quality of code that makes it easy to understand, is proved to be crucial for human users (Scalabrino et al., 2018). However, its influence on Large Language Models (LLMs) remains unclear.We mainly explore code readability from the perspectives of code comments and code style:

Simple examples are more sensitive to comments. Code comments in examples serve multiple purposes: they may summarize the functionality of the code, explain individual steps, or display the content of result variables (Steidl et al., 2013). Chen et al. (2024) demonstrates that code comments are crucial for bridging the gap between code functionality and its implementation, thereby enhancing code comprehension.

Model	Complexity	Comments	MonkeyEval		BeatNumEval	
			Pass@1	Pass@10	Pass@1	Pass@10
ChatGPT	simple	✓	35.9	55.4	30.9	47.5
	simple	✗	32.9	47.5	26.5	42.5
	complex	✓	47.7	64.4	46.9	67.3
	complex	✗	46.5	63.4	47.1	65.3
CodeGeex4	simple	✓	36.3	63.4	30.5	62.4
	simple	✗	33.3	61.4	24.1	58.4
	complex	✓	41.2	66.3	38.6	71.3
	complex	✗	40.2	66.3	36.4	68.3

Table 2: Results of code generation using examples with different settings of example comments

We established two comparative groups to assess the impact of comments on example effectiveness: with comments and without comments. Furthermore, we accounted for complexity by performing experiments on examples with varying levels of complexity. Our experimental results (Table 2) indicate that for simple examples (NOA $\leq$ 3), code comments are vital. In contrast, for inherently complex examples (NOA $\geq$ 4), the absence of code comments has minimal effect. This may be attributed to complex examples inherently containing sufficient contextual information, compensating for the lack of code comments and mitigating their impact on the outcomes.

A consistent code style enhances the effectiveness of examples.

Code style refers to the conventions and patterns used in writing code (Lee et al., 2013). Consistent code style is considered beneficial for code

Model	Style Refinement	MonkeyEval		BeatNumEval	
		Pass@1	Pass@10	Pass@1	Pass@10
ChatGPT	✓	49.5	63.3	50.7	69.3
	✗	47.7	61.6	46.9	65.6
CodeGeex4	✓	38.6	66.3	37.4	73.3
	✗	39.9	64.4	38.6	71.3

Table 3: Results of code generation using examples with different code style

Model	Error Type	MonkeyEval		BeatNumEval	
		Pass@1	Pass@10	Pass@1	Pass@10
ChatGPT	Correct	47.7	64.4	46.9	67.3
	Syntax Errors	45.5	63.2	42.3	62.4
	Inconsistent Naming	42.8	61.4	41.6	59.4
	Non-existent Methods	39.7	57.3	35.6	57.4
	Parameter Misuse	32.4	42.6	21.4	36.6
CodeGeex4	Correct	41.2	66.3	38.6	71.3
	Syntax Errors	40.3	65.8	35.6	66.3
	Inconsistent Naming	38.6	60.4	36.6	59.4
	Non-existent Methods	34.7	54.5	31.7	58.4
	Parameter Misuse	24.7	39.6	29.7	42.6

Table 4: Results of code generation using examples with different error types

readability (Wyrich et al., 2021). To investigate the impact of code style on the results, we refined the code style of existing examples for comparison. Specific optimizations included using a line-by-line format, incorporating additional intermediate steps, and optimizing variable naming to enhance clarity. Appendix Figure 5 illustrates a demonstration of code style rewriting. The results of Table 3 indicate that adopting a more readable code style can effectively enhance performance. This demonstrates that code style contributes to the overall effectiveness of the code.

These findings underscore that code readability does influence example effectiveness. Highly readable code more clearly and effectively presents API usage and logical steps, thereby enhancing the performance of LLM-generated outputs.

RQ3: Impact of Code Correctness

Semantic errors severely degrade example quality while syntactic errors show limited impact.

To systematically analyze error impacts, we follow prior work (Radevski et al., 2016) and categorize example errors into four types: ❶ **Syntax Errors:** Surface-level lexical issues including typos and missing semicolons. ❷ **Inconsistent Naming:** Mismatched variable/API references in examples. ❸ **Non-existent Methods:** Calls to undefined library functions or classes. ❹ **Parameter Misuse:** Incorrect argument types or boundary violations.

Through controlled mutations of correct examples, we generate corrupted variants preserving original functionality while introducing specific error types. Table 4 quantifies three key findings from this analysis: The results of Table 4 demonstrate three key patterns: (1) Syntax errors cause relatively minor degradation (1-5% pass@1 reduction), as LLMs can often self-correct surface-level issues; (2) Semantic errors like parameter misuse induce severe performance drops (11-22% pass@1 reduction), directly misleading API understanding; (3) Error impacts follow consistent severity rankings across models: Parameter Misuse > Non-existent Methods > Inconsistent Naming > Syntax Errors. This hierarchy suggests that errors disrupting API semantics (parameter configurations and method existence) are more detrimental than those affecting implementation details.

4 Proposed Method: ComboPrompt

Our analysis in Section 3 demonstrates that effective examples require moderate complexity, semantic correctness, and step-by-step execution patterns. However, examples in existing documentation-based pools typically fail to meet these standards. To bridge this gap, we propose ComboPrompt, a data augmentation framework designed to enhance documentation examples before any RAG procedure is applied. Our method comprises:

(1) API Combination Search: Exploring diverse API combinations through cross-functional sampling to discover complex examples;

(2) Quality Verification: Filtering candidates via execution check and complexity threshold to ensure correctness and quality;

(3) Code Style Refinement: Enhancing code readability through style optimization.

4.1 API Combination Search

This phase aims to generate complex examples by linking distinct API calls through shared variables or data flows. The key challenge is in the exponentially vast API Example Space where random combinations typically fail to produce valid code.

We employ functional grouping to guide the search: combining APIs from different functional categories (e.g., data loading and filtering) reduces invalid combinations while providing semantic cues that help LLMs recognize natural composition patterns. Our approach consists of three steps:

Step 1—Functional Grouping: Partition APIs into groups by parsing documentation index pages, which typically group APIs by functionality (example from TorchData in Appendix A.11, Table 12).

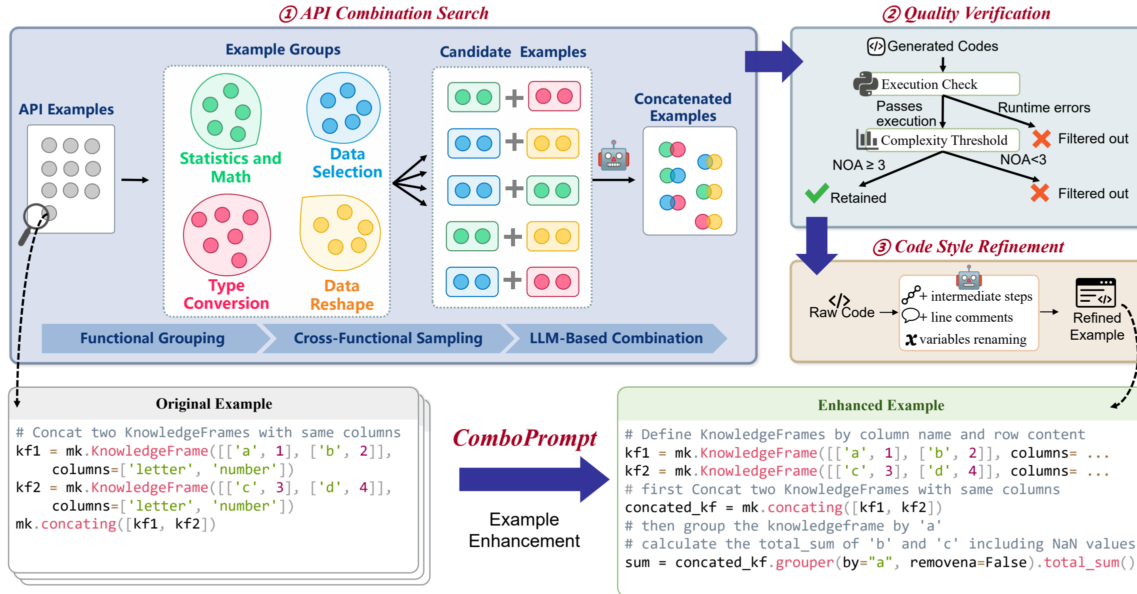

Figure 3: Method Overview: We systematically enhances documentation examples by generating diverse API combinations, verifying their quality through execution and complexity checks, and refining their code style for improved readability.

Step 2—Cross-Functional Sampling: For each search iteration randomly select two distinct functional groups to ensure diversity. From each group, randomly select two existing API examples to serve as the building blocks for the LLM's search.

Step 3—LLM-Based Combination: Instruct an LLM to concatenate the selected examples into coherent multi-API code. The prompt specifies functional context (group names/descriptions) and includes a concatenation demonstration (prompt in Appendix A.11, Prompt 14), allowing LLM to identify composition points (shared data types, compatible I/O signatures) and generates valid combinations

4.2 Quality Verification

The search phase generates diverse API combinations, but not all satisfy our quality criteria established in Section 3. We implement a two-stage verification process:

Verification via Execution: We run each generated example and discard those with runtime errors or semantic inaccuracies (e.g., parameter misuse), ensuring correctness.

Verification via Complexity Threshold: We compute the Number of API calls (NOA) for each validated example and retain only those meeting a minimum threshold (NOA $\geq$ 3) to guarantee sufficient complexity.

4.3 Code Style Refinement

Beyond complexity and correctness, we enhance readability through LLM-based style optimization, including: (1) inserting explanatory intermediate steps, (2) adding detailed line-by-line comments, and (3) implementing semantic variable renaming (prompt in Appendix A.11, Prompt 15).

5 Experiments

5.1 Experiment Setup

Baselines To evaluate the effectiveness of our enhanced examples, we utilize the following example pool for RAG code generation as baselines for comparison: **Vanilla**: This baseline directly obtains outputs from LLMs based on the coding question without augmenting external knowledge. **DocPrompting (Zhou et al., 2023)**: This approach leverages code documentation by first retrieving relevant documentation pieces given a natural language (NL) intent, and then generating code based on the NL intent and the retrieved documentation. **Evor-Code (Su et al., 2024)**: This method involves prompting LLMs to write short scripts demonstrating sample usage of each function in the documentation corpus. The generated scripts are then executed. If a code snippet reports errors during execution, it is included as a pair of (code, error); otherwise, it is regarded as a code snippet that demonstrates syntax and function usage. **Human**: This refers to the comprehensive example pool used in Section 2,

Model	Method	MonkeyEval		BeatNumEval		TorchData		Ring		Pony	
		Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10
ChatGPT	Vanilla	28.4	37.6	25.4	38.8	47.7	58.7	3.7	10.3	1.8	4.7
	DocPrompt	35.9	55.4	30.9	47.5	52.5	63.7	8.4	20.8	2.7	7.5
	Evor-code	27.9	47.5	31.7	51.5	48.5	61.8	7.6	20.4	3.8	7.6
	ComboPrompt	**39.6**	**60.4**	**41.2**	**62.3**	**56.2**	**67.5**	**10.6**	**23.2**	**5.4**	**9.6**
	Human	47.7	64.4	46.9	67.3	–	–	–	–	–	–
CodeGeex4	Vanilla	28.5	47.3	29.3	58.1	49.8	60	5.1	12.6	2	6.3
	DocPrompt	36.3	63.4	31.2	62.4	54	71.3	9.3	22.5	4.4	10.3
	Evor-code	35.2	60.4	32.1	61.3	51.1	66.5	9.1	21.9	5.3	14.8
	ComboPrompt	**37.8**	**64.4**	**34**	**66.3**	**57.1**	**73.2**	**11.3**	**23.5**	**7.5**	**20.3**
	Human	41.2	66.3	38.6	71.3	–	–	–	–	–	–
DeepSeek-R1	Vanilla	34.3	53.5	51.3	71.7	52.6	67.5	7.5	19.4	6.2	11.4
	DocPrompt	47.0	61.9	64.2	76.2	51	63.9	16.7	25.4	9.6	13.1
	Evor-code	48.5	52.6	62.3	77.6	**51**	**65**	17.6	27.1	11.4	20.8
	ComboPrompt	**49.23**	**61.9**	**67**	**79.0**	50	63.4	**20.3**	**33.7**	**13.7**	**29.6**
	Human	51.2	63.4	63.7	78.9	–	–	–	–	–	–
GPT-4o	Vanilla	21.1	41.3	25.7	46.8	48	59.2	7.2	17.8	3.9	8.5
	DocPrompt	**33.2**	**58.7**	**34.7**	64.9	54.5	65.9	12.9	24.3	7.3	18.4
	Evor-code	28.1	49.3	31.5	65.2	55.2	62.8	15.8	30.6	8.7	22.34
	ComboPrompt	30.6	54.6	33.6	**68.8**	**55.2**	**69.8**	**18.5**	**41.4**	**13.2**	**25.4**
	Human	39.9	62.9	36.4	70.4	–	–	–	–	–	–

Table 5: Evaluation of pass@k metrics for ComboPrompt in comparison with other baseline methods on five distinct private library benchmarks. The results demonstrate that ComboPrompt, although inferior to human-generated examples, outperforms other methodologies.

which serves as the upper bound.

Dataset We adhere to the same experimental setup as outlined in Section 2. To conduct a more extensive evaluation of our method, we utilized the torchdata dataset (Zan et al., 2022b), which serves as a private library equivalent and was launched after ChatGPT. Additionally, we used Pony (Su et al., 2024) and Ring (Su et al., 2024), two non-Python languages with little publicly available data, to further evaluate our method's effectiveness on other programming languages. Dataset statistics are provided in Appendix A.2.

5.2 Overall Results

Table 5 presents the results demonstrating that ComboPrompt consistently outperforms the two baseline models across all five private library datasets. In comparison to directly generating code with gpt-3.5-turbo, incorporating concatenated examples via ComboPrompt yields a 13% improvement in pass@1 and a 19% improvement in pass@10. On the MonkeyEval benchmark, ComboPrompt surpasses DocPrompt by 4% in pass@1 and 5% in pass@10, and exceeds Evor-code by 12% in pass@1. In most experimental configurations, ComboPrompt achieves superior performance, indicating its effectiveness in enhancing LLMs' capability to generate code for private code libraries. Notably, ComboPrompt shows considerable gains over DocPrompt, implying that the examples embedded within the official documentation are often insufficient to illustrate the usage of the API. More importantly, for long-context reasoning models such as DeepSeek, ComboPrompt demonstrates a pronounced advantage. By injecting more relevant and informative content into the prompt, ComboPrompt enables the model to more effectively leverage its reasoning capabilities, thereby facilitating a deeper understanding of API functionalities and usage patterns, ultimately increasing the likelihood of correct API invocation. Additionally, when applied to long-tail programming languages such as Ring and Pony, ComboPrompt proves to be a simple yet effective solution for automatically synthesizing informative and context-rich prompts. This allows LLMs to rapidly learn the syntactic structures and API invocation patterns of the target language, leading to significant improvements in pass@k accuracy. In summary, the findings demonstrate that ComboPrompt is a promising method for enhancing the utility of existing documentation examples and improving code generation performance in both common and under-resourced programming environments.

5.3 Comprehensive Analysis

To better understand the differences in LLM outputs, we introduce metrics spanning code complexity, task proficiency, API selection precision, and error distribution: ❶ NOA: Number of distinct API calls per solution; ❷ LOC: Executable lines of code (LOC) per implementation; ❸ Single-step Accu-

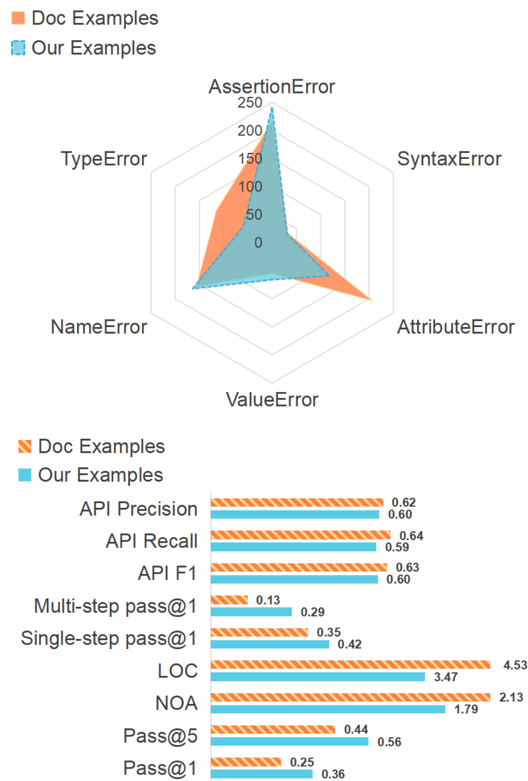

Figure 4: Error distribution and metrics of outputs on Monkey-ChatGPT

racy: Pass@1 for simple tasks solvable with 0-1 API calls; ❹ Multi-step Task Accuracy: Pass@1 for complex tasks requiring ≥ 2 API calls; ❺ API Selection Metrics (F1 Score, Recall, Precision): Evaluate the model's ability to select appropriate APIs; ❻ Error Type Distribution: Contrasts semantic errors (e.g., AttributeError, TypeError) with syntactic errors (e.g., SyntaxError).

As shown in Figure 4 , ComboPrompt examples demonstrate three advantages: ❶ **Enhanced Complex Task Execution**: ComboPrompt achieves 72.3% multi-step accuracy (18.7% higher than documentation examples' 53.6%), indicating superior capacity to guide LLMs through intricate API workflows. This improvement stems from the examples' explicit encoding of API interaction patterns and execution sequences, enabling more effective learning of API composition strategies. ❷ **Improved Implementation Efficiency**: Despite comparable API selection performance (F1=0.597 vs. 0.629), ComboPrompt yields 18.9% fewer API calls (1.786 vs. 2.129) and 23.3% shorter implementations (3.479 vs. 4.537 LOC) than documentation examples. This demonstrates its ability to focus LLM outputs on essential functionalities while eliminating redundant operations. ❸ **Strengthened Semantic Understanding**: ComboPrompt reduces critical semantic errors by 43% fewer API hallucinations (118 vs. 207 AttributeErrors) and 49% less parameter misuse (59 vs. 115 TypeErrors). These reductions confirm improved comprehension of API specifications, particularly regarding method existence and type constraints.

6 Related Work

Retrieval-augmented generation (RAG) has become a key paradigm for enabling large language models (LLMs) to leverage external knowledge, particularly in code generation tasks (Zhang et al., 2023a; Lu et al., 2022; Li et al., 2024a). Recent work includes documentation-driven methods (Zhou et al., 2023; Zan et al., 2022a; Gao et al., 2024), which retrieve API documentation to guide code generation, and library adaptation frameworks that align LLMs with private APIs. Several methods specifically utilize API examples: Evor (Su et al., 2024) implements active retrieval across code snippets and documentation, while EpiGEN (Li et al., 2024b) synthesizes prompts using API names, descriptions, and examples. Current studies primarily focus on example selection mechanisms and retrieval strategies (Gao et al., 2023; Wu et al., 2023; Zhang et al., 2023b), as evidenced by their demonstrated performance improvements. However, these methods largely overlook the intrinsic characteristics that determine example quality, such as code complexity (Ajami et al., 2017), code readability (Buse and Weimer, 2008), and code correctness (Radevski et al., 2016). Our work shifts focus from how to select examples to analyzing what makes examples effective, proposing automated methods for quality enhancement.

7 Conclusion

In this paper, we systematically investigated the intrinsic characteristics of examples, focusing specifically on complexity, readability, and correctness, and their effects on code generation for private libraries. Our research elucidates the influence of these factors and delineates the defining traits of effective examples. Specifically, effective examples are characterized by moderate complexity, semantic correctness, and step-by-step execution patterns. Building on these insights, we introduced ComboPrompt, a method that enhances existing document examples to elevate the performance of private library code generation. This work contributes to a

deeper understanding of the quality determinants of examples in RAG, facilitating enhanced accuracy and automation in private library code generation.

Limitations

Scope of Example Characteristics. While we analyzed three fundamental characteristics (complexity, readability, and correctness), other dimensions may influence example quality. These include: 1) *modular design* - whether examples are properly encapsulated as functions/classes, 2) *error handling patterns* - presence of input validation or exception handling, 3) *data diversity* - coverage of edge cases in test data, and 4) *context dependency* - requirements for external variables or environment configurations. Future work should investigate these additional factors.

Suboptimal API Selection. Our functional grouping approach uses random sampling from API clusters, which may increase computational overhead. More sophisticated selection strategies (e.g., mining frequent API call patterns or leveraging library dependency graphs) could improve concatenation efficiency.

Acknowledgements

This work was supported in part by the National Key R&D Program of China 2024YFE0200800, the National Natural Science Foundation of China under Grants (62321001, 62471055, U23B2001, 62101064, 62201072), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM107), the High-Quality Development Project of the MIIT (2440STCZB2584), the Ministry of Education and China Mobile Joint Fund (MCM20200202, MCM20180101), and the 2025 Education and Teaching Reform Project Funding at Beijing University of Posts and Telecommunications (2025YZ005).

References

Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2017. Syntax, predicates, idioms: what really affects code complexity? In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, page 66–76. IEEE Press.

Raymond P.L. Buse and Westley R. Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 121–130, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yijie Chen, Yijin Liu, Fandong Meng, Yufeng Chen, Jinan Xu, and Jie Zhou. 2024. Comments as natural logic pivots: Improve code generation via comment perspective. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7040–7051, Bangkok, Thailand. Association for Computational Linguistics.

Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs? . In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 761–773, Los Alamitos, CA, USA. IEEE Computer Society.

Xinyu Gao, Yun Xiong, Deze Wang, Zhenhan Guan, Zejian Shi, Haofen Wang, and Shanshan Li. 2024. Preference-Guided Refactored Tuning for Retrieval Augmented Code Generation . In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 65–77, Los Alamitos, CA, USA. IEEE Computer Society.

Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier North-Holland, Inc., Amsterdam.

Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 960–970.

International Organization for Standardization. 2011. ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO 25000 Portal. Accessed: 2026-04-17.

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*, ICML'23. JMLR.org.

Taek Lee, Jung-Been Lee, and Hoh In. 2013. A study of different coding styles affecting code readability. *International Journal of Software Engineering and Its Applications*, 7:413–422.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024a. Acecoder: An effective prompting technique specialized in code generation. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

Sijie Li, Sha Li, Hao Zhang, Shuyang Li, Kai Chen, Jianyong Yuan, Yi Cao, and Lvqing Yang. 2024b. EpiGEN: An efficient multi-api code GENeration framework under enterprise scenario. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 6206–6215, Torino, Italia. ELRA and ICCL.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional api recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, page 87–98, New York, NY, USA. Association for Computing Machinery.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

OpenAI. 2022. Ada embedding. Accessed: 2026-04-17.

OpenAI. 2023. New models and developer products announced at devday. Accessed: 2026-04-17.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26619–26645. PMLR.

Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91.

Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. 2016. Towards building api usage example metrics. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 619–623.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *J. Softw. Evol. Process*, 30(6).

Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. EvoR: Evolving retrieval for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 2538–2554, Miami, Florida, USA. Association for Computational Linguistics.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Di Wu, Xiao-Yuan Jing, Hongyu Zhang, Yang Feng, Haowen Chen, Yuming Zhou, and Baowen Xu. 2023. Retrieving api knowledge from tutorials and stack overflow based on natural language queries. *ACM Trans. Softw. Eng. Methodol.*, 32(5).

Marvin Wyrich, Andreas Preikschat, Daniel Graziotin, and Stefan Wagner. 2021. The mind is a powerful place: How showing code comprehensibility metrics influences code understanding. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 512–523. IEEE Press.

Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-library-oriented code generation with large language models. *Preprint*, arXiv:2307.15370.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022a. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 277–288, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022b. Cert: Continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2369–2375. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

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

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda.

A Appendix

A.1 Models

We utilize `GPT-3.5 (gpt-3.5-turbo-1106)`, `CodeGeex-4`, `GPT-4o` and `Deepseek-R1 (671B)` through API calls. The retriever model leverages `all-MiniLM-L6-v2` (Wang et al., 2020). All experiments are conducted on a server equipped with A100 and A800 GPUs, running Ubuntu.

A.2 Benchmarks

MonkeyEval and BeatNumEval: Derived from PandasEval (Zan et al., 2022b) and NumpyEval (Zan et al., 2022b) respectively, these benchmarks test LLMs' ability to generate code for unseen libraries through systematic API renaming. Specifically, all library-specific keywords are replaced with synthetic equivalents (e.g., "pandas" → "monkey", "dataframe" → "knowledgeframe"), effectively creating private libraries with zero training data leakage.

	MonkeyEval	BeatNumEval	TorchData	Ring	Pony
problem num	101	101	55	107	113
language	python	python	python	Ring	Pony

Table 6: Dataset statistics

TorchDataEval: Based on the TorchData library in Python, this benchmark evaluates models against an unseen library containing 50 test examples.

Ring and Pony: For long-tail programming languages, we select Ring and Pony. These languages have little public data and are adapted from LeetCode.

A.3 Metrics

Pass@k (Functional Correctness): Following previous studies [3, 1, 29], we assess the functional correctness of programs by executing test cases and compute the unbiased Pass@k. Specifically, we generate n > k programs per requirement, count the number of correct programs c < n that pass test cases, and calculate the Pass@k as follows:

$$Pass@k := E_{Requirements}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (1)$$

where $n \geq k$ samples are generated per requirement, and c denotes correct solutions passing all test cases. We run $n = 15$ to calculate Pass@1 and Pass@10

A.4 Example Pool Construction

We construct two example pools for Monkey and BeatNum respectively.

DS-1000: We adapt data from DS1000 (Lai et al., 2023). Each DS-1000 data entry includes fields such as prompt, reference_code, problem_id, library, test_case_cnt, and code_context. We combine prompt and reference_code to form an API usage example.

GitHub: We crawl a certain number of codes and process them as follows: extract all functions using AST operations, and filter functions based on the following criteria: (1) contain more than 2 target library APIs, (2) other APIs apart from

API library	Example Source	count
Monkey	document	366
	DS-1000	179
	GitHub	121
BeatNum	document	247
	DS-1000	142
	GitHub	103

Table 7: Statistics of example pool

target library APIs and Python native APIs should be less than 2, (3) pure code lines should be less than 20, and (4) syntactically correct (can be successfully parsed by AST). Then, three experienced PhD students select functions with clear and explicit functionality that contain logic related to the target library and are not overly trivial.

Table 1 details the distribution of these example sources. The combined pool ensures diversity across complexity levels, with documentation examples being concise (avg 2.17-2.36 LOC) and DS-1000 examples demonstrating higher complexity (avg 3.76-5.09 LOC). This variety enables comprehensive analysis of example characteristics' impact on code generation performance.

A.5 Grouping based on Complexity

For the Number of distinct API calls (NOA), we divided the pool of examples into eight groups according to the varying NOA values. Each group was then used separately for code generation tasks. A similar grouping strategy was applied for the Lines of Executable Code (LOC). For the Halstead Volume (VOL), due to its wide range of values from 0 to 1088, we first sorted the examples by their VOL values and then evenly distributed them into five distinct groups. This approach ensures a balanced representation of complexity levels across the groups.

A.6 Broader Discussion of Potential Factors

To clarify why we focus on **Complexity**, **Readability**, and **Correctness**, we provide the design rationale behind this choice. Our selection follows a systematic filtering process inspired by ISO/IEC 25010 (International Organization for Standardization, 2011), while adapting the quality model to the specific role of API examples in retrieval-augmented code generation. Unlike production software artifacts, API examples are primarily intended to demonstrate correct and effective API usage in a concise and learnable form. This difference substantially changes which quality characteristics are most relevant for evaluation.

Table 8 summarizes how major software quality characteristics relate to API examples. The analysis shows that the three selected dimensions cover the most critical properties for pedagogical code snippets: **Correctness** captures whether the demonstrated usage is functionally valid, **Readability** reflects how easily the usage pattern can be understood, and **Complexity** reflects whether the example contains sufficiently informative interactions rather than oversimplified API invocations. At the same time, other characteristics such as performance efficiency, reliability, and portability are less central in this setting because API examples are not intended to serve as production-ready software components. This filtering process therefore yields a focused yet comprehensive framework for evaluating example quality.

A.7 Code Style Rewrite

Below is a concrete code style rewrite example:

Before style rewrite

```
def g(kf, row_list, column_list):
    result = kf[column_list].iloc[row_list].total_sum(axis=0)
    return result.remove(result.index[result.argmax()])
```

After style rewrite

```
def g(kf, row_list, column_list):
    # Select specified rows and columns
    selected_data = kf[column_list].iloc[row_list]
    # Sum the selected data row-wise (axis=0)
    sum_result = selected_data.total_sum(axis=0)
    # Find index of maximum value and remove this row
    max_index = sum_result.argmax()
    result_with_no_max = sum_result.remove(max_index)
    return result_with_no_max
```

Figure 5: comparison of code style

A.8 Ablation Study

In this ablation study, we aim to evaluate the contribution of each component of our ComboPrompt approach to the overall performance improvements in RAG-based code generation. We focus on three key components: functional grouping for API selection, example filtering, and code style restructuring.

Functional Grouping Analysis: Replacing our functionality-based API grouping with random selection of four candidate examples causes an 13.8%

Quality Characteristic	Representative Sub-characteristics	Rationale for Selection or Exclusion
Functional Suitability	Functional Correctness, Functional Appropriateness	**Partially Selected.** Functional correctness is essential and directly corresponds to our **Correctness** dimension. Functional appropriateness, such as whether an example matches a user query, is mainly handled by the RAG retriever rather than being an intrinsic property of the example itself.
Maintainability	Analyzability, Modularity, Testability, Reusability	**Partially Selected.** Analyzability, namely the ease with which code can be understood, is central to API examples and is captured by our **Readability** dimension. In contrast, modularity, testability, and reusability are less relevant for short illustrative snippets that are not designed for long-term maintenance.
Performance Efficiency	Time Behavior, Resource Utilization	**Excluded.** The primary goal of an API example is clarity rather than runtime optimization. In many cases, a more explicit step-by-step example is pedagogically preferable even if it is less efficient.
Reliability	Fault Tolerance, Maturity	**Excluded.** API examples usually demonstrate ideal-path usage and are not expected to include comprehensive exception handling or production-level robustness mechanisms.
Usability	Learnability, Operability	**Covered by the selected dimensions.** For API examples, learnability largely depends on whether the code is easy to follow (**Readability**) and whether it demonstrates sufficiently meaningful interactions (**Complexity**).
Portability	Adaptability, Installability	**Excluded.** API examples are inherently library-specific and are not intended to support cross-platform or cross-environment deployment.

Table 8: Mapping general software quality characteristics to the evaluation of API examples.

Models	Variants	MonkeyEval		BeatNumEval	
		Pass@1	Pass@10	Pass@1	Pass@10
ChatGPT	Ours	**39.6**	**60.4**	**41.2**	**62.3**
	with random group	31.8	50.5	35.0	56.6
	without correctness filter	35.9	53.4	36.3	58.3
	without complexity filter	37.1	58.4	36.3	60.3
	without style refine	37.3	55.5	40.7	60.5
CodeGeex 4	Ours	**37.8**	**64.4**	**34.0**	**66.3**
	with random group	28.4	50.6	32.3	58.4
	without correctness filter	35.6	57.4	30.6	56.4
	without complexity filter	32.7	60.5	29.9	63.2
	without style refine	33.2	59.7	31.8	64.3

Table 9: Ablation study assessing the contributions of three essential components: API grouping, example filtering, and code style refinement within the ComboPrompt

pass@10 drop. This substantial degradation confirms that functional grouping helps LLMs better comprehend the library's operational logic, enabling more meaningful API combinations. **Filtering Mechanism Impact**: Results of disabling the two-stage filtering pipeline (execution validation + NOA > 3 complexity check) reveals: (1) Omitting execution validation alone causes 9.9% degradation, as 58% of unfiltered examples contain runtime errors (e.g., parameter misuse); (2) Removing complexity filtering yields 3.1% reduction, with 42% examples falling below NOA $= 4$.

Style Refinement Effect: Disabling code restructuring (semantic renaming, step comments) results in a modest 4.9% accuracy decrease, suggesting readability improvements provide secondary but measurable benefits.

These findings align with our Section 3 conclusions about example quality requirements. The results particularly emphasize the necessity of functionality-aware grouping – without API contextualization, LLMs struggle to synthesize coherent workflows from random API combinations. Meanwhile, the filtering mechanism proves essential for eliminating misleading examples, as evidenced by the high error rate (58%) in raw LLM outputs.

A.9 Cost Analysis of ComboPrompt

To further clarify the practical cost of ComboPrompt, Table 10 reports a detailed token and monetary analysis for augmenting the MonkeyEval dataset using the `gpt-3.5-turbo-1106` API. The results show that the primary cost arises from the *API Combination Search* stage. This is mainly because the LLM-based search explores a large number of candidate combinations, while a substantial portion of these candidates are subsequently re-

moved by the quality verification stage for failing to satisfy correctness or complexity requirements. By contrast, the *Code Style Refinement* stage is only applied to the retained candidates and therefore consumes significantly fewer tokens.

Component	Token Usage	Cost (USD)	ΔPass@10
API Combination Search	342,884	$0.34	+9.9%
Code Style Refinement	47,306	$0.06	+4.9%
Total (per example created)	6,293	$0.0065	–
RAG (per query)	1,330	$0.0016	–

Table 10: Analysis of token usage, cost, and performance gain per component

Overall, the analysis indicates that ComboPrompt provides a favorable balance between cost and effectiveness. The augmentation cost is modest, while the resulting performance gains are substantial. More importantly, this cost is incurred only once in an offline manner during construction of the enhanced example pool. After the augmented pool has been created, it can be reused for subsequent RAG-based code generation without repeating the augmentation process, making the additional cost at inference time negligible.

A.10 Quality Improvement After ComboPrompt

Dimension	Metric	Original Doc Example	After ComboPrompt
Complexity	Avg. NOA	1.65	5.44 (+229.7%)
	Avg. LOC	2.42	7.98 (+229.8%)
Readability	Comment Coverage	40.5%	100% (+147.0%)
Correctness	Execution Success Rate	66.3%	100% (+50.8%)

Table 11: Comparison of example quality before and after applying ComboPrompt.

To further justify the proposed evaluation framework, we examine whether ComboPrompt improves examples specifically along the three selected dimensions. ComboPrompt is explicitly designed to enhance **Complexity**, **Readability**, and **Correctness**; therefore, if these dimensions indeed capture meaningful aspects of example quality, they should also reflect the effect of the method. Table 11 quantifies the changes from the original documentation examples to the augmented examples produced by ComboPrompt on the Monkey dataset using ChatGPT. The results show consistent gains across all three dimensions, directly linking the proposed framework to the observed effectiveness of the method.

A.11 Method Details

Group Name	Group Description	Group APIs
Archive DataPipes	These DataPipes help opening and decompressing archive files of different formats.	Decompressor, RarArchiveLoader, TarArchiveLoader, XzFileLoader, ZipArchiveLoader
Augmenting DataPipes	These DataPipes help to augment your samples.	Cycler, Enumerator, IndexAdder
Combinatorial DataPipes	These DataPipes help to perform combinatorial operations.	Sampler, Shuffler
Combining/Spliting DataPipes	These tend to involve multiple DataPipes, combining them or splitting one to many.	Concater, Demultiplexer, Forker, IterKeyZipper, MapKeyZipper, Multiplexer, SampleMultiplexer, UnZipper, Zipper
Grouping DataPipes	These DataPipes have you group samples within a DataPipe.	Batcher, BucketBatcher, Collator, Grouper, UnBatcher
Mapping DataPipes	These DataPipes apply a given function to each element in the DataPipe.	FlatMapper, Mapper
Selecting DataPipes	These DataPipes help you select specific samples within a DataPipe.	Filter, Header

Table 12: Function groups crawled from TorchData documents

A.12 Prompt Templates

Here are the prompt instructions used for the LLM.

Prompt for Code Generation

Role. You are an API programming assistant to help me complete the following task.

Task Description. The user will provide the previous code, the code requirement (in comments), and the API information to be used. You need to write the remaining code using the provided APIs to complete the user's requirements.

Task Instructions.

- First analyze the user requirement and API information, and think about which APIs to use and how to use them.
- Analyze the reasons for your API choices and parameter settings.
- Your output should be divided into two parts: **Analysis** and **Rest code**.
- You may proceed step by step, and you are allowed to use multiple lines of code to gradually solve the task.
- Do not invent APIs; you can only use the ones provided by me.
- Not all provided APIs will be used, and you should select them according to the query.
- Finally, only output the remaining code. Do not output test functions.

Begin!

- **API Information:** `{api_info}`
- **API Examples:** `{api_examples}`
- **Input Previous Code and Requirement:** `{complete_user_query}`

Output Format. Just output **Analysis** and **Rest code**.

Answer:

Table 13: The prompt for code generation

Prompt for Code Concatenation

Task. Your task is to combine and rewrite existing code to create more complex functionality.

Library Context. The code library I am using is `{lib_name}`, a Python library designed for data processing. To develop more intricate code, you need to use the APIs from `{lib_name}` and logically integrate these APIs.

Demonstration.

```
<Demonstration>
{demonstration_for_concatenation}
</Demonstration>
```

I will provide specific examples from the `{lib_name}` library, and you can rewrite them based on these examples.

Task Begin!

Candidate Code Snippets. Here are some example APIs to be combined, each belonging to a different functional group. You can integrate them based on their functionalities:

````
```py
{candidate_api_examples}
```
````

Documentation for the APIs Used: `{api_info}`

You are only allowed to use the APIs that have been mentioned above; do not invent any new APIs. Generate a code snippet with more complex functionality that includes additional API calls and more intricate logic.

Your analysis and output:

Table 14: The prompt for code concatenation

Prompt for Code Style Refinement

Task. I am working with some Python code and need your help to refactor it for better readability.

Improvement Techniques.

- Break down complex code calls into a line-by-line format.
- Use more assignment statements, create intermediate variables, and execute in steps.
- Add more comments to explain each step of the operation.
- Optimize the naming of intermediate variables to make them more intuitive about the variable's function and purpose.

Example. Based on the following example, please refactor the input code.

````
<example>
Input:
```py
def g(df, row_list, column_list):
   result = df[column_list].iloc[row_list].sum(axis=0)
   return result.drop(result.index[result.argmax()])
```

Output:
```py
def g(df, row_list, column_list):
   # Select specified rows and columns
   selected_data = df[column_list].iloc[row_list]
   # Sum the selected data row-wise (axis=0)
   sum_result = selected_data.sum(axis=0)
   # Find the index of the maximum value and remove this row
   max_index = sum_result.argmax()
   result_with_no_max = sum_result.drop(max_index)
   # Return the result
````
````

```
    return result_with_no_max
```
```
</example>
```

Task Begin!

Input.

```
```py
{input_code}
```
```

Output:

Table 15: The prompt for code style refinement