LongCodeU: Benchmarking Long-Context Language Models on Long Code Understanding

Jia Li[1,2], **Xuyuan Guo**[3], **Lei Li**[4], **Kechi Zhang**[1,2], **Ge Li**[1,2*], **Jia Li ♂**[5], **Zhengwei Tao**[1,2], **Fang Liu**[6], **Chongyang Tao**[6], **Yuqi Zhu** [1,2], **Zhi Jin**[1,2*]

[1]Key Lab of High Confidence Software Technology (PKU), Ministry of Education
[2]School of Computer Science, Peking University, China
[3]School of Software & Microelectronics, Peking University, China
[4]The University of Hong Kong [5]College of AI, Tsinghua University
[6]School of Computer Science and Engineering, Beihang University
{lijiaa,lige,zhijin}@pku.edu.cn, {guoxy26,nlp.lilei}@gmail.com

Abstract

Current advanced long-context language models offer great potential for real-world software engineering applications. However, progress in this critical domain remains hampered by a fundamental limitation: the absence of a rigorous evaluation framework for long code understanding. To gap this obstacle, we propose a long code understanding benchmark LongCodeU from four aspects (8 tasks) to evaluate LCLMs' long code understanding ability required for practical applications, including code unit perception, intra-code unit understanding, inter-code unit relation understanding, and long code documentation understanding. We evaluate 9 popular LCLMs on LongCodeU (*i.e.,* 6 general models and 3 code models). Our experimental results reveal key limitations in current LCLMs' capabilities for long code understanding. Particularly, the performance of LCLMs drops dramatically when the long code length is greater than 32K, falling far short of their claimed 128K∼1M context windows. In the four aspects, inter-code unit relation understanding is the most challenging for LCLMs. Our study provides valuable insights for optimizing LCLMs and driving advancements in software engineering.

1 Introduction

Recent advances in long-context language models (LCLMs) like Gemini-1.5 (Team et al., 2024) and GPT-4o (GPT, 2024), which support context windows exceeding hundreds of thousands of tokens, offer unprecedented potential for real-world software engineering applications. These models promise transformative improvements in related downstream tasks, such as repository-level code generation (Zhang et al., 2024b; Bi et al., 2024), real-world GitHub issues resolution(Jimenez et al., 2023), and long code summarization (Dhulshette et al., 2025). However, progress in this critical area remains hampered by a fundamental limitation: **the lack of rigorous evaluation frameworks for long code understanding**—a capability essential for real-world software development tools that require accurate code unit perception, intra-code unit understanding, inter-code unit relation understanding, and long code documentation understanding in code repositories.

```
def calculate_positive_average(lst):
  pos_n = [i for i in lst if i > 0]
  return sum(pos_n) / len(pos_n) if pos_n else 0

def upper_and_sort(str_list):
  upper_str = []
  for string in str_list:
    upper_str.append(string.upper())
  return sorted(upper_str, key=len)
(more lines . . )
```

(a) A long code with independent functions

```
import codecs
def decode_utf7(enc):
  return codecs.decode(enc, 'utf-7')

def namespace(self):
  data = self._command_and_check("namespace")
  parts = []
  for item in parse_response(data):
  (more lines . . )
    for prefix, separator in item:
      prefix = decode_utf7(prefix)
      parts.append((prefix, to_unicode)
  return Namespace(*parts)
```

(b) A real-world long code with non-standalone functions

Figure 1: Examples of a synthetic long code with independent functions and a real-world long code with non-standalone functions. Dependencies are highlighted.

Current benchmarks generally fall into two categories, which face five fundamental limitations that hinder comprehensive evaluation of LCLMs' long code understanding capabilities. The first category includes studies such as RepoQA (Liu et al., 2024), whose task design has insufficient diversity like only focusing on needle function search. While these evaluations are useful, ❶ **they do not capture the real-world full range of code understanding capabilities** needed for practical coding scenarios. Additionally, benchmarks like L-Eval An et al. (2023) further compound these limitations by using synthetic "long code" through a

*Corresponding authors.

simple joining of independent code snippets. ❷ This approach **overlooks the natural dependencies between code segments**, as shown in Figure 1. ❸ These studies also face issues with **data contamination**. They neither enforce temporal constraints on code release dates nor address potential model pre-exposure through evaluating on previously published datasets. ❹ Their maximum supported context length of 36.5K tokens also **falls far short of stress-testing the claimed 128K-1M context windows of modern LCLMs**. The second category includes benchmarks like Long Code Arena (Bogomolov et al., 2024), LongBench (Bai et al., 2023), SWE-bench (Jimenez et al., 2023), and DevEval (Li et al., 2024b), which evaluate long-context understanding based on performance in downstream tasks. ❺ However, these benchmarks **entangle code understanding with other task-specific challenges** like code generation and bug fixing. This makes it hard to determine whether performance limitations are due to a lack of code understanding or other factors.

To address these limitations, we propose LONGCODEU, a benchmark designed to isolate and comprehensively evaluate LCLMs' capacity to understand and reason about real-world, dependency-rich, long code contexts. Our benchmark offers the following key features:

- **Comprehensive Tasks stem from Practical Applications.** We evaluate LCLMs from four aspects (8 tasks) to evaluate the LCLMs' long code understanding capability required for practical applications, including code unit perception, intra-code unit understanding, inter-code unit relation understanding, and long documentation understanding.
- **Extra-long Code Context.** Each task contains around 500 gathered long codes. The lengths of examples change in 0∼8K, 8∼16K, 16∼32K, 32∼64K, and 64∼128K following the normal distribution, which far exceeds the maximum length of 36.5K supported by existing benchmarks (An et al., 2023).
- **Real-world Repository.** The benchmark is collected from real-world code repositories. Long code consists of one or more real code file contents, instead of being composed of multiple independent code units like current benchmarks (Liu et al., 2024).
- **Reducing Data Contamination.** We collect up-to-date code repositories that are created after 2024-06 from GitHub[1], which are later than most prevailing LCLMs' cut-off dates thus reducing the risk of data contamination.

We evaluate 9 popular LCLMs, which contain 6 general models (*i.e.,* GPT-4o (GPT, 2024), Claude-3.5-Sonnet (cla, 2024), Gemini-1.5-Flash (Team et al., 2024), DeepSeek-V2.5 (Bai et al., 2023), Mistral-v0.3 (Jiang et al., 2023), and Phi-3.5 (Abdin et al., 2024)) and 3 code models (*i.e.,* DeepSeek-Coder-V2 (Bai et al., 2023), Qwen2.5-Coder (Hui et al., 2024), CodeLlama (Roziere et al., 2023)) on LONGCODEU. The experimental results reveal key limitations in current LCLMs' capabilities for long code understanding. Especially, **LCLMs' performance drops dramatically when the long code length is greater than 32K, falling far short of their claimed context windows such as 128K-1M tokens**. In the four aspects, **inter-code unit relation understanding is the most challenging for LCLMs.** Our findings provide valuable insights for optimizing LCLMs and driving advancements in software engineering.

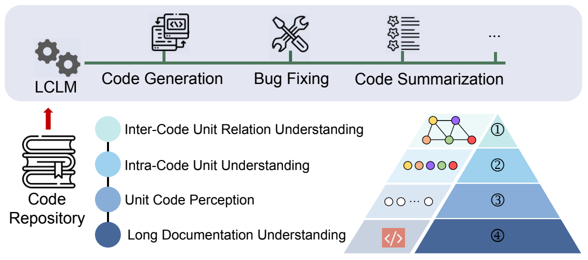
2 Related Works

Benchmarks on Long Code Understanding. Existing studies can be mainly categorized into two types. The first category predominantly explores the long code understanding ability required in downstream applications like the needle-in-a-haystack task, but they face significant limitations. RepoQA (Liu et al., 2024) is a pioneer introduced needle function retrieval task. However, the single task is insufficient to evaluate the complex long code understanding ability. L-Eval (An et al., 2023), a widely-used benchmark, has limitations as well. It constructs artificial "long code" by concatenating independent code snippets, ignoring context dependency in real-world source code. These limitations highlight the need for a more comprehensive and reliable benchmarking approach for evaluating LCLMs' long code understanding ability. The second category includes research like LCArea (Bogomolov et al., 2024). They verify LCLMs' long code understanding ability by measuring their performance on downstream tasks such as LongBench (Bai et al., 2023), SWE-bench (Jimenez et al., 2023), and EvoCodeBench (Li et al., 2024a). Although they provide an intuitive way to assess LCLMs, it suffers from a significant drawback. The performance of downstream tasks is confounded by

[1]https://github.com/

Table 1: The comparison between existing benchmarks and LONGCODEU. #Num is the abbreviation of number. #Div Tasks refers to diverse tasks. #High Disp represents high dispersion. #Max-L and #Avg-L mean the maximum length and the average length of long code. #Trunk-L means whether each example has the length label. #Doc refers to documentation related to repositories. #Task represents the number of tasks (*i.e.,* examples).

Benchmark	Comprehensive Code Tasks			Extra-long Data			Real-world Repository			Reduce Data Leaking	Data Scale
	#Num	#Div Tasks	#High Disp	#Max-L	#Avg-L	#Length-L	Code	#Doc	#Num	Data Time	#Task
The second category benchmarks (Only some benchmarks are listed)											
LongBench [6]	2	✗	✗	–	0.4K	✗	Function	✗	–	2023.02–2023.08	1,000
LC-Arena [8]	6	✗	✗	–	–	✗	File	✗	62	2023.01–2024.05	–
LONGPROC [30]	1	✗	✓	–	2K	✓	Function	✗	0	No Limit	200
DevEval [21]	1	✗	✓	–	0.3K	✓	File	✗	164	2023.11-2024.02	1,825
The first category benchmarks											
RepoQA [22]	1	✗	✗	16K	–	✗	Function	✗	50	No Limit	500
L-Eval [5]	1	✗	✗	36.5K	31.5K	✗	Function	✗	0	No Limit	90
LONGCODEU	8	✓	✓	128K	54.8K	✓	File	✓	116	2024.06–2024.11	3,983


LCLM
Code Generation
Bug Fixing
Code Summarization
...
Code Repository
Inter-Code Unit Relation Understanding
Intra-Code Unit Understanding
Unit Code Perception
Long Documentation Understanding
①
②
③
④
Figure 2: Four understanding aspects in LONGCODEU.

multiple factors, and these works do not decouple the long code understanding ability independently, which is orthogonal to our objective-evaluating LCLMs' capacity to understand and reason about real-world, dependency-rich, long code.

Long Context Language Models. Recent studies have explored diverse ways to extend large language models' context window size. The direct way is to fine-tune models on long sequences (Wu et al., 2021), but they are often effort-costing. Some approaches involve additional fine-tuning for a longer context (Xiong et al., 2023; Chen et al., 2023a,b,c; Peng et al., 2023). They down-scale the input position indices to match the original context window size of models with several training steps. There are also some training-free studies, which use window attention to clip the long sequences (Han et al., 2023; Ding et al., 2023; Xiao et al., 2023). Concurrently, a series of approaches modify the relative distance to extend the extrapolation length (Zhang et al., 2024a; Jin et al., 2024). In this paper, we construct a comprehensive benchmark to evaluate LCLMs' long code understanding ability.

3 LONGCODEU Benchmark

In this section, we first introduce long code understanding tasks (§3.1). Then, we describe the construction process of LONGCODEU (§3.2). Finally, we present the evaluation metrics (§3.3).

3.1 Tasks

In real-world software development, long code understanding is usually oriented towards code repositories. These repositories take functions as basic code units, establish relations among units to achieve complex applications, and introduce documentation to describe code-related information. LCLMs with good long code understanding abilities should be able to perceive and understand basic code units, relations among units, and associated code documentation, which is essentially required for dealing with downstream tasks such as repository-level code generation, issue resolving, and long code summarization. In this paper, we propose LONGCODEU to comprehensively evaluate LCLMs' long code understanding ability from four aspects: code unit perception, intra-code unit understanding, inter-code unit relation understanding, and long documentation understanding. Four aspects of LONGCODEU are shown in Figure 2

These long code understanding tasks share the following procedure: given an instruction, long code, and anchor input, LCLMs output the desired answer. The instruction briefly describes the request of each task. The anchor input is the detailed demand such as a code unit. In LONGCODEU, instruction and anchor input are generally short, while long code is long containing 0∼128K tokens.

3.1.1 Code Unit Perception

Understanding long code, particularly in code repositories, requires identifying its numerous functions, as they form the foundational code unit for comprehending long code's overall functionality. In this paper, we treat a function as the code unit and introduce the code unit perception task to evaluate LCLMs' code unit identifying ability in long code. Concretely, this task requires LCLMs to identify all defined functions in long code and return their corresponding function names, where long code is composed of one or more code file contents collected from real-world repositories. This ability is the cornerstone for downstream tasks.

3.1.2 Intra-Code Unit Understanding

Based on code unit perception, we further evaluate LCLMs' ability to understand the internal logic and semantics of code units.

Code Unit Data Flow Analysis. In this section, we propose the code unit data flow analysis task that verifies whether LCLMs can understand the internal logic of code units by tracking data flow to a certain extent. Given a code unit in long code and a variable name in the code unit, LCLMs are required to figure out lines where the value of the given variable changes by tracing data flow. For example, after executing the line "upper_string += 2", the value of variable "upper_string" changes. The given code unit is randomly selected from long code and its position in long code is also random. Evaluating code unit understanding ability is significant for LCLMs to discover potential vulnerabilities, repair vulnerabilities, and optimize code.

Code Unit Semantic Analysis. In addition to analyzing the data flow of code units, we propose the code unit semantic analysis task to further verify LCLMs' intra-code unit understanding ability. The task asks LCLMs to return a code unit from the long code that satisfies the given description. The long code contains real-world code files, where all descriptions of code units are removed to prevent LCLMs from acquiring clues in them. This task requires LCLMs understand the semantics of code units and then return the desired code unit.

3.1.3 Inter-Code Unit Relation Understanding

In real-world long code, especially in code repositories, code units are non-standalone. Grasping code unit relations is essential for LCLMs to understand the complex functionality of long code, where code unit relations mainly containing dependency relations and semantic relations. The dependency relation indicates the calls among code units. The semantic relation focuses on the functional similarities of code units.

Dependency Relation Analysis. **(T1)** Given a code unit, this task requires LCLMs to find code units that are invoked by the given unit from long code, where the long code covers one or multiple code files and is collected from the same repository with the given code unit. The dependency relation analysis ability can assist LCLMs correctly identifying other code units related to vulnerable units and determining vulnerability scopes in real-world applications. **(T2)** Considering that in real-world applications, apart from long code as LCLMs' input, developers usually use natural language requirements to interact with LCLMs. Thus, LCLMs also need to understand dependency relation between long code and requirements. Given a natural language description, this task requires LCLMs to find code units from the long code that are invoked for generating the desired code that satisfies the given description. The ability can ensure that LCLMs sucessfully invoke existing code units in repository-level code generation and correctly integrate generated code into the current repository.

Semantic Relation Extraction. Even if two code units have no dependency relationship, they might be semantically similar such as having similar implementation or logic. This section analyzes semantic relations of code units in long code. **(T1)** Given a code unit, this task asks LCLMs to extract semantically similar code units with the given unit from long code. Extracting semantic relations of units can effectively help LCLMs to improve software development efficiency by reusing similar code units when programming, and enhance software maintainability such as finding potentially similar vulnerabilities among semantically similar units. **(T2)** As described in dependency relation analysis (T2), understanding the semantical relations of code units and natural language requirements is more in line with practical development scenarios. For instance, in repository-level code generation, developers input a requirement to LCLMs. LCLMs can find semantically similar units with the given requirement from the current repository and then refer to these units to generate desired codes. In this task, LCLMs need to extract semantically similar code units to a given requirement, which challenges LCLMs to understand the semantics of units in long code and reason their semantic relations.

3.1.4 Long Documentation Understanding

In real-world software engineering, code documentation also plays a crucial role. It encompasses a diverse range of code-related information including descriptions of code units, usage patterns, architecture designs, and more. Consequently, it is essential not merely to verify the long code under-

Table 2: Statistics of LongCodeU. #Num means the number of examples in each task. #C-File represents whether the output can be obtained by aggregating cross-file content. #Avg-L is the average length of the output. #Gran means the granularity of the output.

Task	Input		Output		
	#Num	Format	#C-File	#Avg-L	#Gran
CU_P	487	Code	✗	0.4K	Name
CU_SA	500	Code	✗	0.2K	Function
CU_DFA	500	Code	✗	0.03K	Line
DRA_T1	500	Code	✓	0.3K	Function
DRA_T2	500	Code	✓	0.3K	Function
SRE_T1	500	Code	✓	1.0K	Function
SRE_T2	500	Code	✓	1.1K	Function
LDU	500	Document	✗	0.7K	NL

standing ability, but also to analyze the long code documentation understanding ability of LCLMs. We introduce the long documentation understanding task. Given long documentation and a code unit name such as a function name contained in the documentation, this task requires LCLMs to extract the related information to the unit name. We ensure that the long documentation contains the related knowledge of the given unit name, aiming to effectively evaluate LCLMs. Analyzing long documentation is critical to verify the performance of LCLMs in real-world software development.

3.2 Benchmark Construction

The pipeline for constructing LongCodeU encompasses 6 stages as follows.

Stage ❶: Repository Selection. Referring to the TIOBE index (Tio, 2024) for programming language popularity, the most popular language is Python. Thus, we conduct experiments on Python and will expand to other languages in the future. In Python source platforms, PyPI is a rich Python package index tool. We identify the top 10 popular programming topics in PyPI and obtain the top 50 packages with the most stars in each topic. We then select repositories following four criteria: open-source repositories, created after 2024-06, non-fork and non-malicious, and more than 50 stars. Finally, we crawl these repositories from GitHub and obtain 116 real-world repositories.

Stage ❷: Code Parse. We use *tree-sitter*[2] to design a code parser. This parser identifies code units defined in repositories, traces the definition of code units, and analyzes unit relations: First, it performs static analysis of each file in a repository and extracts unit names defined in it. Then, it executes unit symbol navigation, finding the definitions of all code units in the repository. Finally, it extracts unit names invoked in a code unit to grasp their dependency relations. Combining the three steps, the parser can traverse predefined code units within a repository and obtain intricate dependencies. To obtain semantic relations among units, we apply an advanced embedding model to encode code units, and use the cosine similarities of their representations to measure semantic relations.

Stage ❸: Requirement Collection. We extract requirements contained in the signature of code units with *tree-sitter* and invite two developers to check requirements. Not that it is enough to construct examples for T2-type tasks even if some code units do not contain requirements in repositories.

Stage ❹: Documentation Annotation. We collect documentation from collected code repositories and select the documentation with standard: being easy to distinguish the related information of code units. Then we invite two developers to manually label 500 examples.

Stage ❺: Deduplication. For tasks whose input is code unit, we exclude trivial units (*e.g.,* empty or initialization functions). To ensure the quality of LongCodeU, we randomly select files from repositories as long codes, and ensure long codes of all examples in each task are non-repetitive.

Stage ❻: Benchmark Construction. Based on the above stages, we construct around 500 examples for each task supporting the maximum length of 128K tokens. We make LongCodeU satisfy following goals: comprehensive tasks from practical applications, extra-long code context, real-world repositories, and reducing data contamination.

3.3 Automatic Evaluation

We focus on evaluating the long code understanding ability required for LCLMs to complete downstream tasks. Tasks in LongCodeU commonly require LCLMs to retrieve dispersive snippets from long code and execute reasoning. Thus, we mainly adopt metrics used in retrieval tasks to measure LCLMs, including recall and precision.

For proper evaluation, we refine existing metrics according to the output granularity of tasks: ❶ *Output with code lines* refers to the code unit data flow analysis task, which extracts several lines from long code. We first use exact match (**EM**) to measure each line in outputs. Then, we calculate recall and precision based on EM, acquiring **EM-R** and **EM-P** metrics. ❷ *Output with code unit names*

[2]https://tree-sitter.github.io/tree-sitter/

is related to the code unit perception task. We first find the correct names in output and then calculate the longest common subsequence (LCS) of unit names between output and ground truth, since it not only reflects whether the unit name is correct but also indicates LCLMs' ability to perceive the position of units in long code. Then, we calculate recall and precision based on LCS, named **LCS-R** and **LCS-P**. ❸ *Output with code units* refers to code unit semantic analysis and inter-code unit relation understanding tasks. Code unit semantic analysis only returns one unit. We use **CodeBLEU** as its metric, where it is a popular metric to indicate the consistency of two code sequences. For inter-code unit relation understanding, it returns multiple code units. We first determine whether the name of each generated unit is in ground truth. Then, we calculate CodeBLEU of each unit if it exists in the ground truth. For the code unit whose name is not in the ground truth, its CodeBLEU value is set to 0. Based on CodeBLEU value of each unit, we finally calculate CodeBLEU-based recall and precision, dubbed **CB-R** and **CB-P**. ❹ *Output with descriptions* is related to the long documentation understanding task. Considering that documentation contains many text descriptions, we employ **BLEU** for evaluation, which is commonly used to measure the consistency between two sequences in the natural language process field.

These automatic evaluations can effectively measure LCLMs' long code understanding ability since the outputs of each task are deterministic and task' features are incorporated into these metrics. Figure 5 shows the automatic evaluation correlates well with human annotation, which further demonstrates the reliability of our automatic metrics.

4 Experiment Setups

In this section, we aim to answer the following research questions through a series of experiments.
RQ1. How is the long code understanding ability of LCLMs? We evaluate LCLMs' long code understanding ability on LongCodeU in §5.1.
RQ2. What is performance of LCLMs across long code lengths? We explore the performance of LCLMs on long code with different lengths. §5.2 demonstrates LCLMs' performance comparison across long code lengths.
RQ3. How do developers select models in real-world application scenarios? Based on the experimental results, we summarize the empirical lessons we learned, aiming to help developers select suitable LCLMs (§5.3).

4.1 Base LCLMs

We evaluate 9 advanced LCLMs on LongCodeU, which contain 6 general models (*i.e.,* GPT-4o (GPT, 2024), Claude-3.5-Sonnet (cla, 2024), Gemini-1.5-Flash (Team et al., 2024), DeepSeek-V2.5 (Bai et al., 2023), Mistral-v0.3 (Jiang et al., 2023), and Phi-3.5 (Abdin et al., 2024)) and 3 code models (*i.e.,* DeepSeek-Coder-V2 (Bai et al., 2023), Qwen2.5-Coder (Hui et al., 2024), CodeLlama (Roziere et al., 2023)). DeepSeek-R1 and GPT-o3-mini are newly released LCLMs, but their availability through invoking API is unstable or limited to usage frequency. We can not present their performance now and will evaluate them in the future.

4.2 Experimental Setup

We use greedy search for all experiments. We evaluate LCLMs within their maximum context window length. In the semantic relation extraction task, we use an advanced embedding model stella_en_400M_v5 to encode code units and natural language requirements with the 1,024 dimension version, and select the top five similar code units by calculating the cosine similarities of their embeddings. Constrained by computing resources, we evaluate DeepSeek-V2.5 by invoking API[3] provided by DeepSeek. Although the context length of DeepSeek-V2.5 achieves 128K tokens, the API provided by DeepSeek only supports up to 60K tokens. Thus, we analyze DeepSeek-V2.5 on long codes with 0∼64K tokens.

5 Results and Analysis

5.1 Long Code Understanding Capability

Table 3 presents the performance of LCLMs on LongCodeU. For LCLMs with a context length of less than 128K tokens, we only evaluate them within their supporting maximum lengths.

Comparison across LCLMs. We observe significant performance gaps among LCLMs. For similar-scale models, the performance of code LCLMs is better than the counterpart of general models on most tasks. For example, Qwen2.5-Coder outperforms Phi-3.5 24.31% on intra-code unit understanding (the second-aspect task) in terms of CodeBLEU, and DeepSeek-Coder-V2 achieves 11.75% average improvements on

[3]https://api-docs.deepseek.com/api/deepseek-api

Table 3: The performance of LCLMs on LONGCODEU. We only report recall-based results (EM-R, LCS-R, and CB-R) due to page limitation. The precision-based results (EM-P, LCS-P, and CB-P) can be found in Appendix A.

	#Param	Context Size	Task								
			CU_P	CU_SA	CU_DFA	DRA_T1	DRA_T2	SRE_T1	SRE_T2	LDU	#Avg
Code Models			*Open-Source LCLMs*								
Qwen2.5-Coder	7.6B	128K	43.47	71.06	74.01	30.38	9.59	24.34	21.81	21.83	37.06
DeepSeek-Coder-V2	15.7B	128K	38.67	65.21	48.42	47.26	22.92	24.61	26.21	50.69	40.49
CodeLlama	33.7B	16K	68.57	62.41	79.87	68.82	34.94	44.48	36.34	46.92	55.29
General Models			*Open-Source LCLMs*								
Phi-3.5	3.8B	128K	39.92	46.75	49.52	30.76	9.66	18.99	14.48	34.14	30.53
Mistral-v0.3	7.3B	32K	57.42	63.90	58.00	46.66	18.92	33.91	32.50	58.64	46.24
DeepSeek-V2.5	236B	128K	**70.58**	82.11	77.47	72.25	**56.80**	**49.08**	**47.42**	85.85	**67.70**
			Proprietary Source LCLMs								
Claude-3.5-Sonnet	–	200K	43.82	40.60	45.65	29.37	28.70	26.55	27.77	41.81	35.53
Gemini-1.5-Flash	–	1000K	58.45	83.46	80.37	**72.51**	46.42	39.84	38.69	81.43	61.39
GPT-4o	–	128K	56.42	**86.76**	**87.87**	71.58	48.88	44.45	43.14	**87.54**	65.83

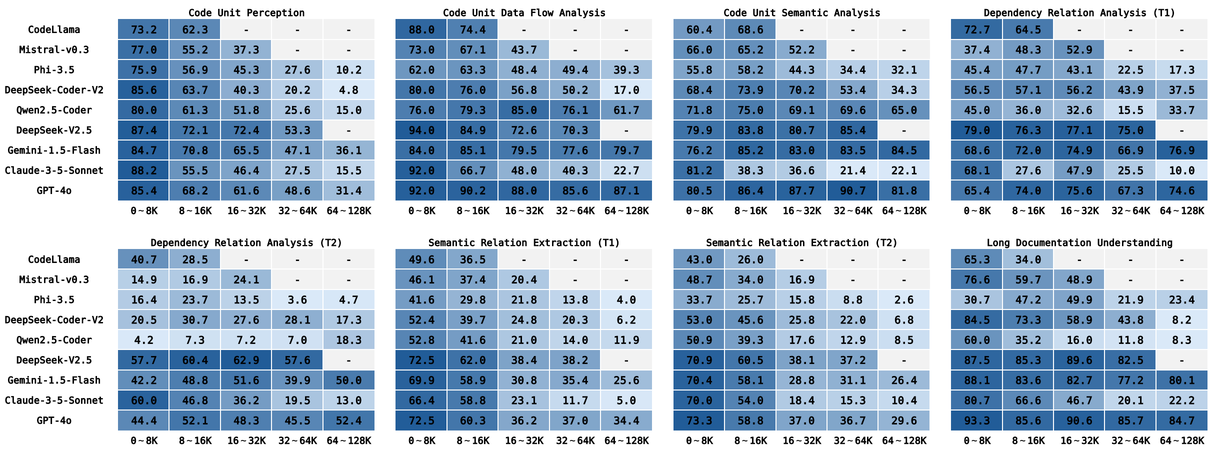


Figure 3: Performance comparison across tasks and long code lengths on LONGCODEU (grey blocks indicate unavailable configurations). The rate of performance degradation exhibits task-specific and model-specific patterns.

inter-code unit relation understanding (the third-aspect task) in CB-R. Among open-source models, DeepSeek-V2.5 performs the best, which is related to its large number of parameters. In proprietary source models, GPT-4o achieves the best performance, while Claude-3.5-Sonnet is not satisfactory.

Comparison across Tasks. In the four aspects, LCLMs perform the best in code unit perception and long documentation understanding, and achieve moderate results in intra-unit code understanding. Inter-code unit relation understanding is the most challenging. Their performances are reasonable. Because it's a fundamental ability for LCLMs to understand code documentation and perceive code units. Based on this, LCLMs understand intra-unit code and then analyze inter-code unit relations, thereby comprehending long code. When observing the third-aspect task (*i.e.,* inter-code unit relation understanding) closely, it can be seen that the performance of dependency relation analysis is lower than that of semantic relation extraction. We also find that no LCLM outperforms others on all tasks. For instance, in terms of scale-similar code models and general models, the former outperforms the latter in code unit perception, while code models perform worse than general models in long documentation understanding.

5.2 Performance across Long Code Lengths

To address RQ2, we classify examples into five buckets according to the long code length, including 0∼8K, 8∼16K, 16∼32K, 32∼64K, and 64∼128K on each task. We conduct experiments on these classes to investigate the true context ability supported by LCLMs in long code understanding. Figure 3, Appendix A, and B show the results of LCLMs across long code lengths.

Our experimental results reveal key limitations in current LCLMs' capabilities for long code understanding. We can also observe a negative correlation between the long code length and the performance of LCLMs. LCLMs' performance drops dramatically when the long code length is greater than 32K tokens, falling well short of their claimed 128K∼1M context windows. Besides, when the long code contains 64∼128K tokens, the performance of LCLMs is near to 10 or even close to 0 on some tasks such as dependency relation analysis and semantic relation extraction. In addition, the degradation slopes of performances vary by task. For example, there is a large slope in long documentation understanding, which means that LCLMs are suitable for processing relatively short documentation. The slope on code unit understanding including code unit data flow analysis and code unit semantic analysis tasks is relatively small. LCLMs fail to model code context effectively in their claimed context windows.

5.3 Empirical Lessons.

Based on the above experiments, we summarize the empirical lessons we learned as: ❶ The small-size LCLMs such as Qwen2.5-Coder can satisfy developers' need for code unit perception and understanding if long code length is less than 16K. Otherwise, we suggest choosing larger-size models. ❷ For understanding long code documentation with more than 32K tokens, it is recommended to use GPT-4o and Gemini-1.5-Flash. Otherwise, Mistral-v0.3 and DeepSeek-Coder-V2 can achieve satisfying performances. ❸ For tasks with high requirements for understanding inter-code unit relations, we suggest selecting the most powerful LCLMs that developers can access. Because when long code length exceeds 64K which is common in repository-level tasks, strong LCLMs also achieve weak perfrmances. These poor performances explain why powerful LCLMs perform unsatisfactorily in repository-level downstream tasks. For example, GPT-4o only achieves 4.00% Success@1 on repository-level code translation benchmark-RepoTransBench (Wang et al., 2024).

6 Discussion

6.1 Case Study

We analyze the outputs of LCLMs, particularly GPT-4o, on the inter-code unit relation understanding task. Appendix D presents two notable examples. GPT-4o often extracts code units that are structurally similar or share overlapping tokens but have distinct or even opposite functionalities. This highlights the need to enhance models' ability to distinguish confusing code units.

6.2 Code Understanding or Memorization?

We collect a small number of early-released code repositories from GitHub that LCLMs have potentially encountered during LCLMs training, aiming to analyze the dependency degree of LCLMs on

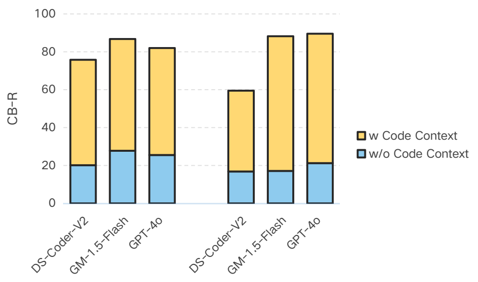


Figure 4: Assessing long code understanding vs. memorization on CU_SA (left) and DRA_T2 (right) tasks.

memorization and long code understanding ability. Concretely, we enter only an instruction and anchor input to models, withholding the long code context, and assess their performance. We select tasks where models can work normally even without long code context, such as code unit semantic analysis and dependency relation analysis (T2). As shown in Figure 4, the memorization performance (w/o context) is much lower than the results with long code context, even though models might have met these contexts when training. The δ score (the results with context minus the performance without context in yellow) relieves the memory phenomenon (Yu et al., 2023) and also reveals the significant importance for measuring LCLMs' long code understanding ability.

6.3 Reliable Evaluation Metrics?

Reliable evaluation metrics are essential for assessing long code understanding. We measure the consistency between our metrics and human evaluation by selecting 20 GPT-4o outputs on each task and inviting two advanced developers to manually evaluate them. Using Kendall-Tau τ (Kendall, 1938), we found that the average τ value is at least 0.75 across all tasks, with the minimum value exceeding 0.7 (Figure 5). This demonstrates a strong correlation between our metrics and human evaluation, confirming the reliability of our metrics.

6.4 Further Evaluation of LCLMs on DCC and CCI Tasks

We conduct a code change impact (CCI) analysis task to further evaluate LCLMs' long code understanding ability. Meanwhile, we also evaluate LCLMs on understanding dynamic code characteristics (DCC) like runtime behavior tracing. The details and results are shown in Appendix C.

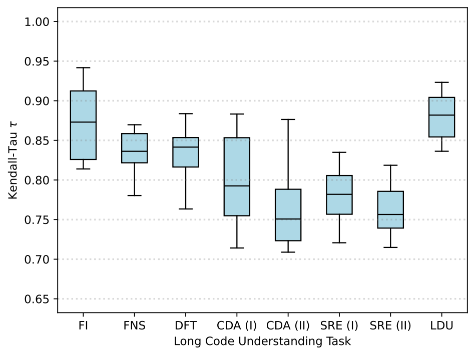


Figure 5: The value of Kendall-Tau τ between our automatic metrics and human evaluation.

7 Conclusion

In this paper, we propose a comprehensive long code understanding benchmark LONGCODEU. It introduces four aspects (8 tasks) to evaluate LCLMs' long code understanding ability required for practical applications, including code unit perception, intra-code unit understanding, inter-code unit relation understanding, and long code documentation understanding. Our experimental results reveal key limitations in current LCLMs' capabilities for long code understanding. When the long code length contains more than 32K tokens, the performance of LCLMs drops dramatically, falling far short of their claimed 128K∼1M context windows. We hope our findings can provide valuable insights for optimizing LCLMs and driving advancements in software engineering.

8 Acknowledgement

This research is supported by the National Key R&D Program under Grant No. 2023YFB4503801, the National Natural Science Foundation of China under Grant Nos. 62192731, 62192730, 62192733, 62072007, and the Major Program (JD) of Hubei Province (No.2023BAA024).

Limitations

This paper proposes a benchmark - LONGCODEU to evaluate the long code understanding ability of long context language models from four aspects which are essential capabilities required for LCLMs to complete real-world downstream tasks. Based on LONGCODEU, we evaluate 9 popular LCLMs and analyze their strengths and shortcomings. We think that DevEval has three limitations.

❶ LONGCODEU is a monolingual benchmark (*i.e.,* requirements in English and code in Python) and ignores other languages. In practice, LLMs require understanding requirements in different natural languages (*e.g.,* Chinese, Spanish) and generating programs in various programming languages (*e.g.,* Java, C). Thus, we plan to build a multilingual LONGCODEU in future work.

❷ Most recently, there are a few newly released LCLMs such as DeepSeek-R1 and OpenAI o3-mini-high. Constrained by the availability or stabilization of API, we do not provide the performance of these newly released models. In the future, we will evaluate their long code understanding abilities on LONGCODEU.

❸ In our experiments, we only consider the long code with 0∼128K tokens, although some LCLMs have supported longer context windows, *e.g.,* Claude-3-5-Sonnet with 200K context size and even Gemini-1.5-Flash with 1000K context size. We will continue to update and evolve our benchmark, in order to support LONGCODEU to measure LCLMs on longer codes.

References

2024. Claude 3.5 haiku. *https://www.anthropic.com/claude/haiku.*

2024. Gpt-4o. *https://openai.com/index/hello-gpt-4o/.*

2024. Tiobe-index. *https://www.tiobe.com/tiobe- index/.*

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219.*

Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *arXiv preprint arXiv:2307.11088.*

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508.*

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. *arXiv preprint arXiv:2403.16792.*

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria

Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, et al. 2024. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*.

Guanzheng Chen, Xin Li, Zaiqiao Meng, Shangsong Liang, and Lidong Bing. 2023a. Clex: Continuous length extrapolation for large language models. *arXiv preprint arXiv:2310.16450*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023b. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Yuhan Chen, Ang Lv, Ting-En Lin, Changyu Chen, Yuchuan Wu, Fei Huang, Yongbin Li, and Rui Yan. 2023c. Fortify the shortest stave in attention: Enhancing context awareness of large language models for effective tool use. *arXiv preprint arXiv:2312.04455*.

Nilesh Dhulshette, Sapan Shah, and Vinay Kulkarni. 2025. Hierarchical repository-level code summarization for business applications using local llms. *arXiv preprint arXiv:2501.07857*.

Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. Longnet: Scaling transformers to 1,000,000,000 tokens. *arXiv preprint arXiv:2307.02486*.

Chi Han, Qifan Wang, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2023. Lm-infinite: Simple on-the-fly length generalization for large language models. *arXiv preprint arXiv:2308.16137*.

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

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. 2024. Llm maybe longlm: Self-extend llm context window without tuning. *arXiv preprint arXiv:2401.01325*.

Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024a. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024b. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3603–3614. Association for Computational Linguistics.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*.

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*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Yanlin Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John C. Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2024. Repotransbench: A real-world benchmark for repository-level code translation. *CoRR*, abs/2412.17744.

Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. 2023. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*.

Xi Ye, Fangcong Yin, Yinghui He, Joie Zhang, Howard Yen, Tianyu Gao, Greg Durrett, and Danqi Chen. 2025. Longproc: Benchmarking long-context language models on long procedural generation. *arXiv preprint arXiv:2501.05414*.

Jifan Yu, Xiaozhi Wang, Shangqing Tu, Shulin Cao, Daniel Zhang-Li, Xin Lv, Hao Peng, Zijun Yao, Xiaohan Zhang, Hanming Li, et al. 2023. Kola: Carefully benchmarking world knowledge of large language models. *arXiv preprint arXiv:2306.09296*.

Kechi Zhang, Ge Li, Huangzhao Zhang, and Zhi Jin. 2024a. Hirope: Length extrapolation for code models using hierarchical position. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13615–13627.

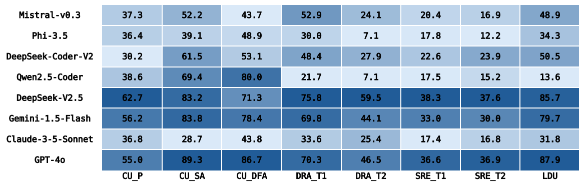
Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024b. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

A Rrecision-based results in RQ2

We present the performance of LCLMs on several tasks which can be measured by precision-based metrics in Figure 7.

B Recall-based performance of LCLMs in moderate-length (16∼64K)

We also conduct experiments to evaluate LCLMs on moderate-length (16∼64K) codes. As shown in Figure 6, the performances of all LCLMs on the moderate-length scenario need to be improved.



	CU_P	CU_SA	CU_DFA	DRA_T1	DRA_T2	SRE_T1	SRE_T2	LDU
Mistral-v0.3	37.3	52.2	43.7	52.9	24.1	20.4	16.9	48.9
Phi-3.5	36.4	39.1	48.9	30.0	7.1	17.8	12.2	34.3
DeepSeek-Coder-V2	30.2	61.5	53.1	48.4	27.9	22.6	23.9	50.5
Qwen2.5-Coder	38.6	69.4	80.0	21.7	7.1	17.5	15.2	13.6
DeepSeek-V2.5	62.7	83.2	71.3	75.8	59.5	38.3	37.6	85.7
Gemini-1.5-Flash	56.2	83.8	78.4	69.8	44.1	33.0	30.0	79.7
Claude-3-5-Sonnet	36.8	28.7	43.8	33.6	25.4	17.4	16.8	31.8
GPT-4o	55.0	89.3	86.7	70.3	46.5	36.6	36.9	87.9

Figure 6: Performance of LCLMs on all tasks within 16∼64K code length.

C The performance of LCLMs on DCC and CCI tasks

Given a modified code unit and its original unit, CCI requires LCLMs to identify other units that are affected by the given code unit changing. We use tree-sitter to analyze dependency relations and identify affected units. We invite two developers to check the affected units. For the DCC task, given a code unit name and its specific inputs, it requires LCLMs to output lines that are executed based on the specific inputs. We use BLEU as the evaluation metric. From Table 4, we can find that the dynamic code characteristics task and the code change impact analysis task are difficult for LCLMs.

D Case Study

Figure 8 shows a representative error case in the dependency relation analysis task. We can find that the output of GPT-4o extracts an error code unit “stream_async" that is confusing to correct invoked function “stream" since the two code snippets have similar structures.

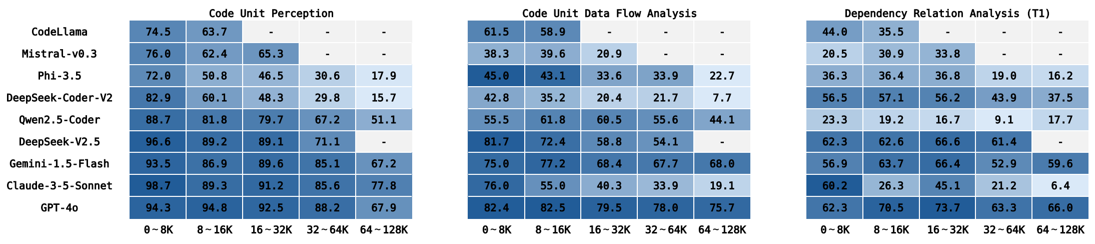
Figure 9 demonstrates a generated result of GPT-4o in the semantic relation extraction task. We can observe that the output contains an error “delete" function which has opposite functionalities to the anchor input, *i.e.,* the given natural language description.

E Instructions in Our Benchmark

We present the instructions employed in diverse tasks. Each instruction has undergone iterative refinement to ensure that different models can not only achieve relatively high metrics but also produce outputs that appear satisfactory and are applicable to real-world development scenarios when using these instructions. These instructions standardize the output format of the models (even though some models may not output strictly in accordance with these specifications), facilitating the parsing of the streaming output of the models using regular expressions during the evaluation process.

Table 4: The performance of LCLMs on DCC and CCI tasks.

	Qwen2.5-Coder	DeepSeek-Coder-V2	CodeLlama	Phi-3.5	Mistral-v0.3	DeepSeek-V2.5	Claude-3.5-Sonnet	Gemini-1.5-Flash	GPT-4o
CCI	24.62	40.81	47.91	27.25	55.92	68.07	42.39	70.34	67.31
DCC	53.13	57.58	67.88	57.68	38.46	72.51	52.30	83.68	57.40



Code Unit Perception

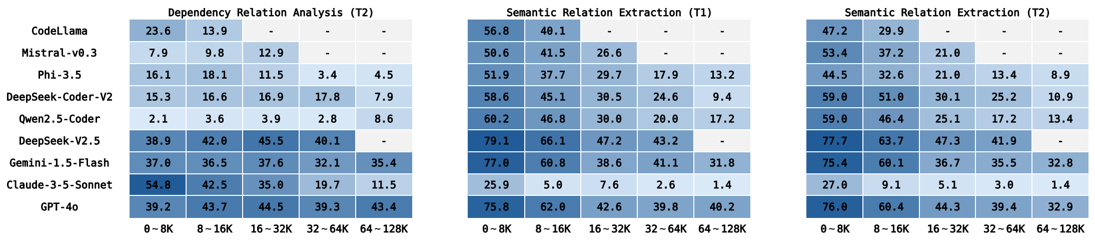
	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	74.5	63.7	-	-	-
Mistral-v0.3	76.0	62.4	65.3	-	-
Phi-3.5	72.0	50.8	46.5	30.6	17.9
DeepSeek-Coder-V2	82.9	60.1	48.3	29.8	15.7
Qwen2.5-Coder	88.7	81.8	79.7	67.2	51.1
DeepSeek-V2.5	96.6	89.2	89.1	71.1	-
Gemini-1.5-Flash	93.5	86.9	89.6	85.1	67.2
Claude-3-5-Sonnet	98.7	89.3	91.2	85.6	77.8
GPT-4o	94.3	94.8	92.5	88.2	67.9

Code Unit Data Flow Analysis

	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	61.5	58.9	-	-	-
Mistral-v0.3	38.3	39.6	20.9	-	-
Phi-3.5	45.0	43.1	33.6	33.9	22.7
DeepSeek-Coder-V2	42.8	35.2	20.4	21.7	7.7
Qwen2.5-Coder	55.5	61.8	60.5	55.6	44.1
DeepSeek-V2.5	81.7	72.4	58.8	54.1	-
Gemini-1.5-Flash	75.0	77.2	68.4	67.7	68.0
Claude-3-5-Sonnet	76.0	55.0	40.3	33.9	19.1
GPT-4o	82.4	82.5	79.5	78.0	75.7

Dependency Relation Analysis (T1)

	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	44.0	35.5	-	-	-
Mistral-v0.3	20.5	30.9	33.8	-	-
Phi-3.5	36.3	36.4	36.8	19.0	16.2
DeepSeek-Coder-V2	56.5	57.1	56.2	43.9	37.5
Qwen2.5-Coder	23.3	19.2	16.7	9.1	17.7
DeepSeek-V2.5	62.3	62.6	66.6	61.4	-
Gemini-1.5-Flash	56.9	63.7	66.4	52.9	59.6
Claude-3-5-Sonnet	60.2	26.3	45.1	21.2	6.4
GPT-4o	62.3	70.5	73.7	63.3	66.0



Dependency Relation Analysis (T2)

	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	23.6	13.9	-	-	-
Mistral-v0.3	7.9	9.8	12.9	-	-
Phi-3.5	16.1	18.1	11.5	3.4	4.5
DeepSeek-Coder-V2	15.3	16.6	16.9	17.8	7.9
Qwen2.5-Coder	2.1	3.6	3.9	2.8	8.6
DeepSeek-V2.5	38.9	42.0	45.5	40.1	-
Gemini-1.5-Flash	37.0	36.5	37.6	32.1	35.4
Claude-3-5-Sonnet	54.8	42.5	35.0	19.7	11.5
GPT-4o	39.2	43.7	44.5	39.3	43.4

Semantic Relation Extraction (T1)

	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	56.8	40.1	-	-	-
Mistral-v0.3	50.6	41.5	26.6	-	-
Phi-3.5	51.9	37.7	29.7	17.9	13.2
DeepSeek-Coder-V2	58.6	45.1	30.5	24.6	9.4
Qwen2.5-Coder	60.2	46.8	30.0	20.0	17.2
DeepSeek-V2.5	79.1	66.1	47.2	43.2	-
Gemini-1.5-Flash	77.0	60.8	38.6	41.1	31.8
Claude-3-5-Sonnet	25.9	5.0	7.6	2.6	1.4
GPT-4o	75.8	62.0	42.6	39.8	40.2

Semantic Relation Extraction (T2)

	0~8K	8~16K	16~32K	32~64K	64~128K
CodeLlama	47.2	29.9	-	-	-
Mistral-v0.3	53.4	37.2	21.0	-	-
Phi-3.5	44.5	32.6	21.0	13.4	8.9
DeepSeek-Coder-V2	59.0	51.0	30.1	25.2	10.9
Qwen2.5-Coder	59.0	46.4	25.1	17.2	13.4
DeepSeek-V2.5	77.7	63.7	47.3	41.9	-
Gemini-1.5-Flash	75.4	60.1	36.7	35.5	32.8
Claude-3-5-Sonnet	27.0	9.1	5.1	3.0	1.4
GPT-4o	76.0	60.4	44.3	39.4	32.9

Figure 7: Performance comparison across long code lengths on tasks which can be measured by precision-based metrics. (grey blocks indicate unavailable configurations). The rate of performance degradation exhibits task-specific and model-specific patterns.

```
You're an expert Python programmer...Extract all functions and methods that the
list function calls from the long Python source codes.
...
Do not include any explanations or additional text.
```

...

~~~
The long Python source codes are as follows:
```
...
def stream(
        ...
    ) -> Iterator[ExecutionInstance]:
        ...
        page = self.page(
            date_created_from=date_created_from,
            date_created_to=date_created_to,
            page_size=limits["page_size"],
        )

        return self._version.stream(page, limits["limit"])
...
async def stream_async(
        ...
    ) -> AsyncIterator[ExecutionInstance]:
            ...
        page = await self.page_async(
            date_created_from=date_created_from,
            date_created_to=date_created_to,
            page_size=limits["page_size"],
        )

        return self._version.stream_async(page, limits["limit"])
...
```
~~~

...

~~~
The programming requirement of the list function is:
```
def list(
        ...
    ) -> List[ExecutionInstance]:
        """
        Lists ExecutionInstance records from the API as a list.
        ...
        :param datetime date_created_from:
        ...
        :returns: list that will contain up to limit results
```
~~~

(a) Input

~~~
```
###### function_1 ######
def stream_async(
        ...
    ) -> AsyncIterator[ExecutionInstance]:
        ...
        page = await self.page_async(
            date_created_from=date_created_from,
            date_created_to=date_created_to,
            page_size=limits["page_size"],
        )

        return self._version.stream_async(page, limits["limit"])
...
~~~

(b) Output

Figure 8: For the dependency relation analysis task, the output of GPT-4o extracts a error code unit “stream_async" that is confusing to correct invoked function “stream".

```
You're an expert Python programmer...Identify and extract the five functions or
classes that are most semantically similar to the __call__ function from the long
Python source codes.
...
Do not include any explanations or additional text.
```

...

~~~
The long Python source codes are as follows:
```
...
    def create(self) -> AssistantsKnowledgeInstance:
        """
        Create the AssistantsKnowledgeInstance
            ...
        """
            ...
        return AssistantsKnowledgeInstance(
            self._version, payload, assistant_id=self._solution["assistant_id"]
        )
...
    def delete(self) -> bool:
        """
        Deletes the AssistantsKnowledgeInstance
            ...
        """
        return self._version.delete(
            method="DELETE",
            uri=self._uri,
        )
...
def forward(
        ...
    ) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Applies the poison noise and compute the loss with respect to the target
gradient.
    ...
```
~~~

...

~~~
The programming requirement of the list function is:
```
def __call__(self, id: str) -> AssistantsKnowledgeContext:
        """
        Constructs a AssistantsKnowledgeContext

        :param id: The knowledge ID.
```
~~~

(a) Input

~~~
```
...
###### Top-3 ######
def delete(self) -> bool:
    """
    Deletes the AssistantsKnowledgeInstance
    ...
...
###### Top-5 ######
def create(self) -> AssistantsKnowledgeInstance:
    """
    Create the AssistantsKnowledgeInstance
    ...
...
```
~~~

(b) Output

Figure 9: For the semantic relation extraction task, the output contains an error “delete" function which has opposite functionalities to the anchor input, *i.e.,* the given natural language description.

(a) Code Unit Perception

You're an expert Python programmer. Identify all defined Python functions and extract their function names from the long Python source codes.

Only extract the Python function names that are defined in the long Python source codes.
Do not extract the Python function names that are mentioned but not defined in the long Python source codes.
Do not extract the Python function names that are defined in the classes of the long Python source codes.
All extracted Python function names are stored in a list Function_names according to the order these function names appear in the long Python source codes.

Make sure the output is strictly in this format:
“
Function_names = [Function_name_1, Function_name_2, ..., Function_name_k]
”

Do not include any explanations or additional text.

(b) Code Unit Data Flow Analysis

You are an expert Python programmer. Identify the {Target_function_name} function from the long Python source code and extract the lines where the value of the {Target_variable_name} variable changes within the {Target_function_name} function.

Only extract lines where the value of the {Target_variable_name} variable changes.
Do not extract lines where the {Target_variable_name} variable is referenced but its value does not change.

Return the extracted lines according to the order they appear in the {Target_function_name} function.

Make sure the output is in the following format:

“
###### The extracted code line ######
{extracted_code_line}
###### Line number ######
{line_number}

###### The extracted code line ######
{extracted_code_line}
###### Line number ######
{line_number}
...
###### The extracted code line ######
{extracted_code_line}
###### Line number ######
{line_number}

“

Do not include any explanations or additional text in the output.

(c) Code Unit Semantic Analysis

You're an expert Python programmer. Understand the long Python source codes. Identify and extract the function that satisfies the given programming requirement.

Extract the entire source code of the function that satisfies the given programming requirement.
Do not only extract the name of the function that satisfies the given programming requirement.

Extracted the entire source code of the function:
“
def function_name(parameters):
# function body
”

Do not include any explanations or additional text.

(d) Dependency Relation Analysis (T1)

You're an expert Python programmer. Extract all functions and methods that the {Target_function_name} function calls from the long Python source codes.

Only extract functions and methods from the given long Python context.
Do not extract functions and methods from the Python standard library or third-party libraries.
Only extract functions and methods that the {Target_function_name} function directly calls.
Do not extract functions and methods that the {Target_function_name} function indirectly calls.
Extract the entire source code of functions and methods that the {Target_function_name} calls.
Do not only extract the name of functions and methods that the {Target_function_name} function calls.
Ensure indentation is preserved.

Please follow the format to complete the skeleton below:
“
###### function_1 ######
def function_name_1(parameters):
# function body

###### function_2 ######
def function_name_2(parameters):
# function body

###### method_1 ######
def method_name_1(self, parameters):
# method body

###### method_2 ######
def method_name_2(self, parameters):
# method body
”

Do not include any explanations or additional text.

(e) Dependency Relation Analysis (T2)

You're an expert Python programmer. Understand the Python programming requirement of the {Target_function_name} function. Extract all functions and methods that the {Target_function_name} function calls from the long Python source codes.

Only extract functions and methods from the given long Python context.
Do not extract functions and methods from the Python standard library or third-party libraries.
Only extract functions and methods that the {Target_function_name} function directly calls.
Do not extract functions and methods that the {Target_function_name} function indirectly calls.
Ensure indentation is preserved.

Do not include any explanations or additional text.

(f) Semantic Relation Extraction (T1)

You're an expert Python programmer. Identify and extract the functions or classes that are most semantically similar to the {Target_function_name} function from the long Python source codes.

Extract the entire source code of functions and classes that are the top-5 semantically similar to the {Target_function_name} function.
Do not only extract the name of functions and classes that are the top-5 semantically similar to the {Target_function_name} function.
The order of extracted five functions or classes is in order of decreasing similarity to the {Target_function_name} function.
Ensure indentation is preserved.

Do not extract the target function {Target_function_name} itself.

Please follow the format to complete the skeleton below:

```
“
###### Top-1 ######
def name_1(parameters):
# function body

###### Top-2 ######
def name_2(parameters):
# body

###### Top-3 ######
def name_3(parameters):
# body

###### Top-4 ######
def name_4(parameters):
# body

###### Top-5 ######
def name_5(parameters):
# body
”
```

Do not include any explanations or additional text.

(g) Semantic Relation Extraction (T2)

You're an expert Python programmer. Understand the Python programming requirement of the {Target_function_name} function. Identify and extract the functions or classes that are most semantically similar to the {Target_function_name} function from the long Python source codes.

Extract the entire source code of functions and classes that are the top-5 semantically similar to the {Target_function_name} function.
Do not only extract the name of functions and classes that are the top-5 semantically similar to the {Target_function_name} function.
The order of extracted five functions or classes is in order of decreasing similarity to the {Target_function_name} function.
Ensure indentation is preserved.

Do not extract the target function {Target_function_name} itself.

Please follow the format to complete the skeleton below:

```
“
###### Top-1 ######
def name_1(parameters):
# function body

###### Top-2 ######
def name_2(parameters):
# body

###### Top-3 ######
def name_3(parameters):
# body

###### Top-4 ######
def name_4(parameters):
# body

###### Top-5 ######
def name_5(parameters):
# body
”
```

Do not include any explanations or additional text.

(h) Long Documentation Understanding

You are an expert Python programmer. Understand the multiple natural language documentations and identify the one that describes the {Target_API_name} API.

Each documentation is labeled with a sequence number and enclosed between special markers: <BEGIN >indicates the start of a documentation, and <END >indicates its conclusion.

Extract and return only the complete documentation corresponding to the {Target_API_name} API, exactly as it appears.

Please follow the format to complete the skeleton below:

```
“
# The sequence number
{sequence number}

# The complete documentation
{complete documentation}
”
```

Do not include any explanations or additional text.