RepoDistill: Distilling Repository Knowledge through Compression-Aware Budget Allocation and Policy Optimization

Xin Yin[1†] **Zixiang Ding**[3†] **Yiang Zhang**[1]
Qiang Wang[3] **Rui Wang**[4] **Chao Ni**[2,1*] **Zhe Cui**[3*]
[1] The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
[2] Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University, China
[3] Hithink Research, China [4] Shanghai Jiao Tong University, China
{xyin,zhangyiang,chaoni}@zju.edu.cn
{dingzixiang,wangqiang3,cuizhe}@myhexin.com wangrui12@sjtu.edu.cn

Abstract

Large Language Models (LLMs) have achieved strong performance on many code-related tasks, yet they still struggle with repository-level scenarios where reasoning depends on long, noisy, and structurally complex contexts. While existing retrieval methods, including both similarity-based and graph-based approaches, can identify relevant code snippets, they often retrieve excessive contexts that intensify the "lost-in-the-middle" phenomenon and dilute model attention with redundant contexts. To address this, we present RepoDistill, a novel framework that integrates retrieval with learned budget allocation for fine-grained context compression. RepoDistill first employs a plug-and-play lightweight GraphRAG to retrieve context that follows logical flows. It then applies Compression-Aware Budget Allocation guided by Compression-Aware Policy Optimization, which formulates context management as a multi-step decision problem and learns allocation policies for contexts. Experiments show that RepoDistill outperforms baselines, achieving gains of up to +7.00 on SWE-QA, +24.4% on CoderEval, and +0.25 on LongCodeU. Furthermore, a compact 4B-parameter model trained with RepoDistill can serve as an effective context compressor for closed-source LLMs, reducing input tokens by up to 66% while maintaining comparable performance. We release our code at https://anonymous.4open.science/r/RepoDistill-6CE3/.

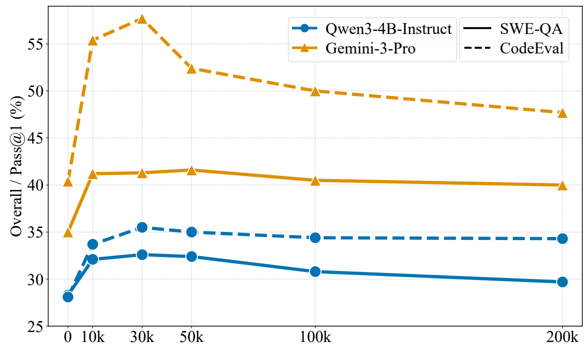

Figure 1: Performance trends on SWE-QA and CoderEval under varying context lengths

1 Introduction

In recent years, Large Language Models (LLMs) have emerged as powerful tools for software engineering, demonstrating strong performance in code translation (Yang et al., 2024; Yin et al., 2024), code summarization (Sun et al., 2025; Ahmed et al., 2024), and code understanding (Nam et al., 2024; Xu et al., 2025). However, real-world software development requires reasoning that goes beyond individual files and instead operates at the repository-level, where context is extensive and interdependent. Tasks such as repository-level code generation and question answering (QA) inherently demand that LLMs navigate long contexts and resolve intricate cross-file dependencies (Peng et al., 2025b,a; Li et al., 2025; Yin et al.). Naively feeding massive amounts of retrieved code into the prompt, however, often triggers the "lost-in-the-middle" phenomenon (Liu et al., 2024) and dilutes the model's attention. To empirically examine this issue, we conducted a preliminary study on repository-level QA and code generation using the standard *Function Chunking RAG* (Wang et al., 2025b). We varied the retrieved context length from 10k to 200k tokens. As illustrated in Figure 1, performance initially improves when the context length increases from 0 to 30k tokens due to the inclusion of more relevant information, but it then degrades sharply as the context extends to 100k to 200k tokens. Crucially, even SOTA models like Gemini-3-Pro (DeepMind, 2025) fail to sustain robust reasoning as the context length increases.

These empirical findings highlight fundamental

[†]Equal contribution.
[*]Corresponding authors. Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou, China.

limitations of existing methods on repository-level tasks. Specifically, we identify two key obstacles that hinder effective long-context code understanding: (1) When processing long contexts, LLMs suffer from the "lost-in-the-middle" phenomenon, and the model's attention is diluted by redundant classes or functions, making it difficult to reason over critical information buried within the long input. (2) Even within retrieved classes or functions that are deemed relevant, substantial noise remains, since only a few lines may be truly pertinent to the query while the entire function body occupies valuable context budget.

To address these limitations, we propose RepoDistill, a novel framework that combines Graph-Based Retrieval-Augmented Generation (GraphRAG), Compression-Aware Budget Allocation (CABA), and Compression-Aware Policy Optimization (CAPO). First, RepoDistill employs a plug-and-play lightweight GraphRAG to retrieve contexts that follow logical flows. Instead of ingesting raw retrieved context, RepoDistill then applies CABA to allocate token budgets for contexts. This budget allocation is guided by CAPO, which formulates context management as a dynamic decision-making process. Through optimization policy, the model learns to autonomously calibrate the budget allocation (0% to 100%) for each snippet, effectively distilling answer-critical logic while discarding redundant information.

We evaluate RepoDistill on three benchmarks covering repository-level question answering (SWE-QA), code generation (CoderEval), and long-context code understanding (LongCodeU). Experimental results demonstrate that RepoDistill consistently outperforms baselines across all settings, achieving overall gains of up to +7.00 points with open-source models and up to +7.61 points with closed-source models on SWE-QA. For code generation, RepoDistill improves Pass@10 by up to +24.1% on Python and +24.4% on Java; for long-context understanding, it yields improvements of up to +0.25 on LDU. Furthermore, a compact 4B-parameter model trained with RepoDistill can serve as an effective context compressor for closed-source LLMs, reducing input tokens by up to 66% while maintaining comparable performance.

In summary, our contributions are as follows:

- We introduce RepoDistill, a novel framework that mitigates the inherent limitations of LLMs in handling long-context code tasks.
- We integrate GraphRAG, CABA, and CAPO to retrieve structurally relevant code, allocate compression budgets, and learn adaptive budget allocation policies.
- We conduct extensive experiments on SWE-QA, CoderEval, and LongCodeU benchmarks. Results show that RepoDistill significantly outperforms baselines and can serve as an effective pre-processor for closed-source LLMs.

2 Approach

As shown in Figure 2, RepoDistill integrates three components: Graph-Based Retrieval-Augmented Generation (GraphRAG), Compression-Aware Budget Allocation (CABA), and Compression-Aware Policy Optimization (CAPO):

- **GraphRAG** constructs and queries a code dependency graph to retrieve task relevant contexts.
- **CABA** equips LLMs with the capability to handle long contexts by allocating budgets to retrieved code snippets.
- **CAPO** formulates long-context code understanding as a multi-step decision problem and guides the LLM to learn budget allocation policies for contexts, retaining answer-critical information.

2.1 Plug-and-Play Lightweight GraphRAG

Repository-level code tasks require capturing explicit structural information, such as inheritance hierarchies and function-call relationships. Given the inherent complexity of repositories and the overhead of constructing comprehensive dependency graphs, we design a lightweight method that can be hot-swapped with more sophisticated graph-aware methods (e.g., RepoScope and GRACE). By explicitly modeling rich dependency relationships among code elements, our GraphRAG retrieves task-relevant code contexts and reasoning paths, providing precise and comprehensive contextual support. The GraphRAG module operates through two core components: Repository Graph Construction and Repository Graph Retrieval.

2.1.1 Repository Graph Construction

This module constructs a structured dependency graph from a repository, capturing containment and invocation relationships among functions and classes. Formally, given a repository with files $\mathcal{F} = \{f_1, f_2, ..., f_n\}$, the process outputs a directed

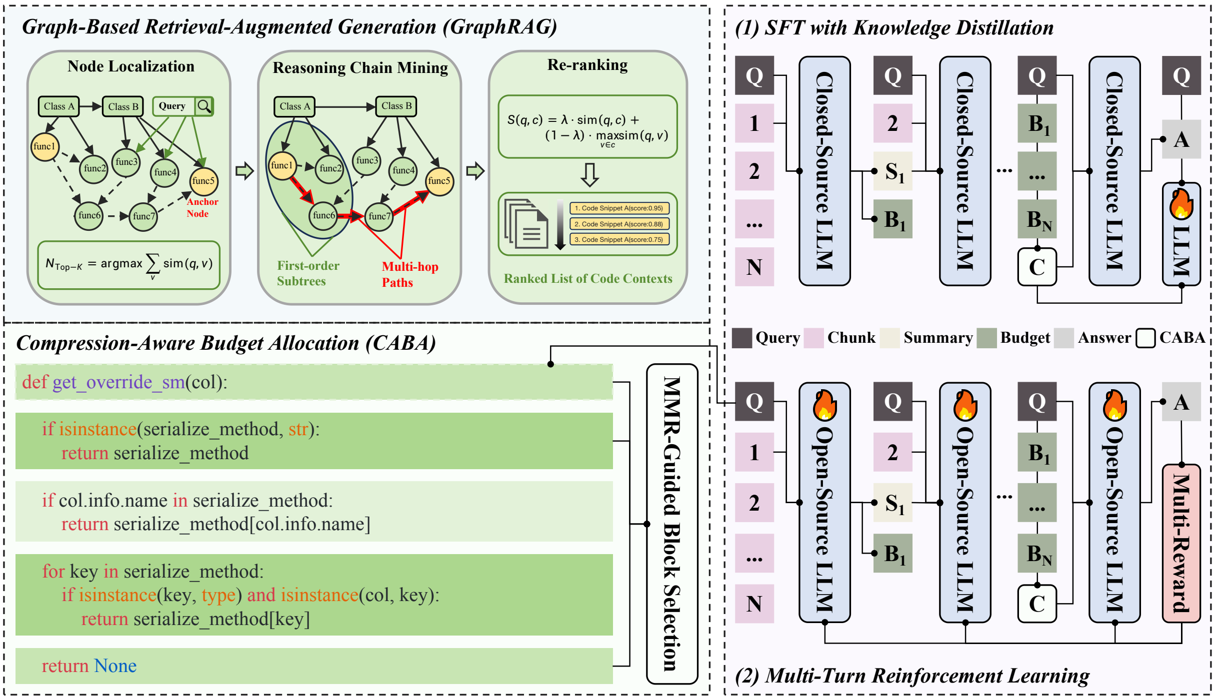

Figure 2: Overview of the RepoDistill. RepoDistill uses GraphRAG to retrieve structurally relevant contexts, and applies CABA with CAPO to dynamically allocate retention budgets while preserving answer-critical information.

attributed graph $G = (V, E)$. Here, V denotes nodes corresponding to code structural units, and $E \subseteq V \times R \times V$ represents edges with relation types $R = \{\text{CONTAIN}, \text{INVOKE}\}$. The construction procedure consists of three sequential steps.

Code Parsing. Each source file $f_i \in \mathcal{F}$ is parsed into a Concrete Syntax Tree (CST) using tree-sitter (Zhu et al., 2024; Ouyang et al., 2025b; Ni et al., 2023). Candidate units are extracted via CST traversal, and we record metadata including file path, line number, name, and code snippet.

Dependency Relation Extraction. We extract two relation types: (1) **CONTAIN** edges link classes to their contained functions or inner classes, reflecting hierarchical structure; (2) **INVOKE** edges connect a caller unit to its callee by resolving function calls within CST subtrees, producing a candidate set of relations.

Graph Building. The final graph G is assembled by instantiating nodes with unit metadata and edges with identified relations. This structured representation encapsulates the code's logical dependencies, providing the foundation for subsequent retrieval.

2.1.2 Repository Graph Retrieval

Given a code graph $G = (V, E)$ and a query q, this module retrieves the most relevant code snippets together with their structured contexts. The retrieval process consists of three steps.

Node Localization. This step identifies the Top-K nodes from V that are most semantically relevant to q. Semantic similarity $\text{sim}(q, v_i)$ is computed using Qwen3-Embedding-0.6B (Zhang et al., 2025) with cosine similarity. The initial anchor set $N_{\text{Top-K}}$ is selected as:

$$N_{\text{Top-K}} = \arg \max_{V' \subset V, |V'|=K} \sum_{v \in V'} \text{sim}(q, v) \quad (1)$$

Reasoning Chain Mining. To capture code logic and dependencies, we expand the context from $N_{\text{Top-K}}$ in two ways:

- First-Order Subtrees: For each anchor $v \in N_{\text{Top-K}}$, we retrieve all nodes directly connected via **INVOKE** edges.
- Multi-Hop Paths: We mine the shortest paths of anchor pairs $v_i, v_j \in N_{\text{Top-K}}$ using **INVOKE** and **CONTAIN** edges, forming a path set $\mathcal{P}$.

This expansion yields a candidate set $\mathcal{C}$ containing the anchors, their first-order subtrees, and the mined multi-hop paths.

Re-Ranking. For each candidate $c \in \mathcal{C}$, we compute a unified relevance score $S(q, c)$:

$$S(q, c) = \lambda \cdot \text{sim}(q, c) + (1 - \lambda) \cdot \max_{v \in c} \text{sim}(q, v) \quad (2)$$

where λ is a hyperparameter, set to 0.5. Candidates are sorted by $S(q, c)$ to produce the final ranked list, which provides structured contexts for tasks.

2.2 Compression-Aware Budget Allocation

Although GraphRAG retrieves relevant functions, they may still contain within-function redundancy (e.g., non-essential branches), which can dilute model attention. Inspired by LongCodeZip (Shi et al., 2025), we design a fine-grained budget allocation mechanism. Unlike LongCodeZip, which employs a coarse-to-fine strategy that first filters functions and then prunes within functions using fixed budgets, RepoDistill allocates adaptive budgets to each function, with the specific allocation policy determined by CAPO (§ 2.3). Given a function f, query q, and token budget t (where t is computed as the function's token length multiplied by the budget rate assigned by CAPO), RepoDistill executes a two-stage compression pipeline:

Perplexity-Guided Block Segmentation. Following LongCodeZip (Shi et al., 2025), we segment functions into semantically coherent blocks using perplexity as boundary signals. The intuition is that abrupt increases in perplexity indicate transitions to new semantic units. Specifically, we compute each line's perplexity and designate a line as the start of a block if its perplexity exceeds neighboring lines by at α (set to 0.2) times the standard deviation.

MMR-Guided Block Selection. RepoDistill next determines which blocks to retain to maximize task relevance while ensuring logical diversity and respecting the token budget t. In LongCodeZip, this is formulated as a 0/1 knapsack problem, which assumes the value of each block is independent. However, code blocks often exhibit strong contextual dependencies and semantic redundancy. Simply accumulating the AMI may lead to a collection of fragments that are individually relevant but collectively redundant. Therefore, we employ the *Maximal Marginal Relevance* (MMR) to iteratively select blocks. The goal is to balance the relevance of a block to the query with its novelty relative to the blocks already selected. For a candidate block b in the set of blocks R, its MMR is defined as:

$$\mathrm{MMR}(b) = \lambda \cdot \mathrm{AMI}(b, q) - (1 - \lambda) \cdot \max_{b_j \in S} \mathrm{Sim}(b, b_j) \quad (3)$$

where q is the query, S is the set of selected blocks, and λ (set to 0.1) is a trade-off parameter that controls the balance between relevance and diversity. The term $\mathrm{AMI}(b, q)$ measures the information gain of the query q given block b, calculated as:

$$\mathrm{AMI}(b, q) = \mathrm{PPL}(q) - \mathrm{PPL}(q \mid b) \quad (4)$$

We adopt a greedy strategy: in each iteration, we select the block b^* that yields the highest MMR score and fits within the remaining token budget. This process continues until the budget is exhausted or no further blocks can be added. Compared to the 0/1 knapsack formulation in LongCodeZip, the MMR-based approach ensures that the compressed context covers more diverse logical facets of the context. Following LongCodeZip, we use Qwen2.5-Coder-0.5B (Hui et al., 2024) to compute PPL and $\mathrm{Sim}(b, b_j)$.

2.3 Compression-Aware Policy Optimization

In this section, we describe the details of compression-aware policy optimization, including the overall workflow (§ 2.3.1), supervised fine-tuning with knowledge distillation (§ 2.3.2), and multi-turn reinforcement learning (§ 2.3.3).

2.3.1 Overall Workflow

As illustrated in Figure 2, RepoDistill reformulates repository-level code understanding as a multi-turn decision process. At each turn, the LLM receives three inputs: (1) the task query, (2) a chunk of code snippets, and (3) a memory summary from the previous turn. Upon processing each chunk, the LLM assigns a retention budget to every code snippet (e.g., 0%/25%/50%/75%/100%, where 0% indicates complete filtering and 100% indicates full retention) and updates the memory summary to capture salient information and maintain coherence across turns. After all chunks have been processed, RepoDistill applies the CABA (§ 2.2) to compress contexts according to its assigned budget. Finally, the LLM synthesizes the final answer based on the task query and the compressed contexts. Detailed prompt templates, including the semantic definitions of each retention budget, memory summary specifications, and the output format for multi-dimensional budget assignments per turn, are provided in Appendix § A.1.

We develop the model's decision-making capabilities via a two-stage training paradigm: first, performing supervised fine-tuning (SFT) on high-quality demonstrations (§ 2.3.2), followed by reinforcement learning to refine the budget allocation strategy (§ 2.3.3)."

2.3.2 SFT with Knowledge Distillation

To bootstrap the LLM with effective budget allocation behavior, we first instantiate RepoDistill (No Training) using Qwen3-Max (Qwen, 2025b) to generate a large-scale training corpus. For each instance in the training set, it assigns a retention budget to every code snippet and produces the corresponding final answer. We sample 10 candidate trajectories per instance, which are then evaluated by human to select the highest-quality sample (detailed in Appendix § A.1.1). This curated dataset is then used for SFT, effectively distilling the budget allocation strategy from the stronger closed-source LLM into the smaller LLM. This human-in-the-loop SFT stage provides a quality-controlled initialization, reducing the model's dependence on LLM-as-judge metrics during training.

2.3.3 Multi-Turn Reinforcement Learning

To optimize the budget allocation policy, we adopt the RLVR recipe (Guo et al., 2025; Seed et al., 2025) with Group Relative Policy Optimization (Shao et al., 2024a) as the base algorithm for its simplicity and effectiveness. As illustrated in Figure 2, RepoDistill generates multiple independent conversations for a single query, each consisting of a sequence of budget allocation decisions and memory updates. We treat each conversation as an independent optimization target, computing an outcome reward based on the final answer and distributing group-normalized advantages across all decision steps within that conversation.

For the reward function, we design three components: (1) a formatting reward R_{format} that encourages well-formed outputs; (2) a task-accuracy reward R_{metric} that evaluates correctness using the metrics from § 3; and (3) an efficiency reward $R_{\text{efficiency}}$ that incentivizes aggressive compression while preserving answer-critical information. Specifically, let L_{input} denote the total input length and L_{used} denote the length after compression. We define the efficiency reward as:

$$R_{\text{efficiency}} = R_{\text{metric}} \cdot \text{Sigmoid}\left(\ln \frac{L_{\text{input}}}{L_{\text{used}}}\right) \tag{5}$$

3 Experimental Setup

Datasets. We evaluate RepoDistill on three benchmarks covering repository-level QA (i.e., SWE-QA (Peng et al., 2025a)), code generation (i.e., CoderEval (Yu et al., 2024)), and long-context understanding (i.e., LongCodeU (Li et al., 2025)). See detailed descriptions in Appendix § A.1.5.

Metrics. We adopt the metrics from the original benchmark papers (Peng et al., 2025a; Yu et al., 2024; Li et al., 2025). For SWE-QA, we employ an LLM-as-judge framework using DeepSeek-V3.2 (Liu et al., 2025a) and Kimi-K2 (Team et al., 2025) to rate answer quality across five dimensions: *correctness*, *completeness*, *relevance*, *clarity*, and *reasoning quality*, each scored from 0.0 to 10.0 (see detailed prompt in Appendix § A.1.4). To ensure reliability, we conduct five independent evaluations per instance with anonymized systems and shuffled answer orders to prevent position bias, and average the scores from both judge LLMs. For CoderEval, we report Pass@k ($k \in \{1, 5, 10\}$) to evaluate code generation. For LongCodeU, we focus on three representative tasks: Code Unit Semantic Analysis (CUSA) using Exact Match, Dependency Relation Analysis (DRA) using Recall, and Long Documentation Understanding (LDU) using BLEU score.

Backbone LLMs and Baselines. We evaluate six widely recognized LLMs, comprising three open-source models (i.e., Qwen3-4B-Instruct, Qwen3-30B-A3B-Instruct (Yang et al., 2025), and Qwen3-Coder-30B-A3B-Instruct (Qwen, 2025a)) and three closed-source models (i.e., GPT-5.2 (OpenAI, 2025), Claude-Sonnet-4.5 (Anthropic, 2025), and Gemini-3-Pro (DeepMind, 2025)).

For SWE-QA and CoderEval, we adopt five baselines for context retrieval: one *Direct* LLM method, two semantic similarity-based retrieval methods (i.e., *Function Chunking RAG* (Wang et al., 2025b) and *Sliding Window RAG* (Zhang et al.)), and two graph-based methods (i.e., *RepoScope* (Liu et al., 2025b) and *GRACE* (Wang et al., 2025a)). For **LongCodeU**, since the relevant context is already extracted, we directly compare LLM performance on contexts compressed by RepoDistill against the original uncompressed context.

In addition to context retrieval methods, we also compare against LongCodeZip (Shi et al., 2025) for context compression and Context-Picker (Zhu et al., 2025) for context selection. See detailed descriptions in Appendix § A.1.6.

Implementation. We partition the SWE-QA dataset (12 projects) into training (8 projects), validation (2 projects), and test (2 projects) sets to avoid cross-repo leakage. For open-source LLMs, we train them on the training split to develop their capability for long code compression and critical information retaining, then evaluate on SWE-QA and other downstream tasks (e.g., code generation in CoderEval and code understanding in LongCodeU) to assess the generalization and robustness of RepoDistill. For closed-source LLMs, we conduct two experiments: (1) directly applying closed-source LLMs without training, denoted as Re-

Table 1: Comparative performance of different methods on SWE-QA (see Appendix § A.2.1 for additional LLMs)

Method	Evaluation Metrics					Overall
	Correctness	Completeness	Relevance	Clarity	Reasoning	
Qwen3-4B-Instruct (Direct)	5.12	3.33	7.06	7.54	5.20	28.25
+ Function Chunking RAG	5.48 (+0.36)	4.30 (+0.97)	6.94 (-0.12)	7.39 (-0.15)	5.62 (+0.42)	29.73 (+1.48)
+ Sliding Window RAG	5.60 (+0.48)	4.40 (+1.07)	6.93 (-0.13)	7.36 (-0.18)	5.77 (+0.57)	30.06 (+1.81)
+ RepoScope	5.70 (+0.58)	4.65 (+1.32)	7.14 (+0.08)	7.67 (+0.13)	5.83 (+0.63)	30.99 (+2.74)
+ GRACE	5.79 (+0.67)	4.63 (+1.30)	7.38 (+0.32)	7.58 (+0.04)	6.15 (+0.95)	31.53 (+3.28)
+ RepoDistill	**6.43** (+1.31)	**5.85** (+2.52)	**8.03** (+0.97)	**8.09** (+0.55)	**6.79** (+1.59)	**35.19** (+6.94)
GPT-5.2 (Direct)	6.98	5.51	8.71	8.52	7.16	36.88
+ Function Chunking RAG	8.20 (+1.22)	8.39 (+2.88)	9.23 (+0.52)	8.36 (-0.16)	8.27 (+1.11)	42.45 (+5.57)
+ Sliding Window RAG	7.98 (+1.00)	8.26 (+2.75)	9.14 (+0.43)	8.24 (-0.28)	8.32 (+1.16)	41.94 (+5.06)
+ RepoScope	8.09 (+1.11)	8.42 (+2.91)	9.29 (+0.58)	8.30 (-0.22)	8.36 (+1.20)	42.46 (+5.58)
+ GRACE	8.16 (+1.18)	8.46 (+2.95)	9.18 (+0.47)	8.36 (-0.16)	8.33 (+1.17)	42.49 (+5.61)
+ RepoDistill (No Training)	**8.31** (+1.33)	**8.56** (+3.05)	**9.35** (+0.64)	**8.64** (+0.12)	**8.51** (+1.35)	**43.37** (+6.49)
+ RepoDistill (Qwen3-Trained)	8.25 (+1.27)	8.46 (+2.95)	9.32 (+0.61)	8.56 (+0.04)	8.44 (+1.28)	43.03 (+6.15)

poDistill (No Training); (2) integrating the trained Qwen3-4B-Instruct as a pre-processing module, denoted as RepoDistill (Qwen3-Trained). For context retrieval methods in main results (§ 4.1), we retrieve relevant functions (200k tokens) from the repository as context. For training, we set the epochs to 5 for SFT and 2 for GRPO, using the AdamW optimizer with a learning rate of 1e-5 and linear warm-up scheduling. We set the chunk size of CAPO to 10,000 tokens, while adopting default settings for LLM, including temperature, top-k, and top-p. All experiments were conducted on a system equipped with 32 $\times$ NVIDIA A100-80GB GPUs.

4 Evaluation

4.1 Main Results

Table 1 and Table 2 present the comparative results of RepoDistill against baselines across SWE-QA, CoderEval, and LongCodeU benchmarks.

Comparison on Open-Source LLM. On SWE-QA, all retrieval-augmented methods consistently outperform the *Direct* baseline across all evaluation dimensions. Semantic similarity-based methods (i.e., *Function Chunking RAG* and *Sliding Window RAG*) achieve moderate improvements of +1.48 and +1.81 points, respectively, while graph-based methods (i.e., *RepoScope* and *GRACE*) yield higher gains of +2.74 and +3.28 points by leveraging structural dependencies. RepoDistill achieves the best performance with an overall improvement of +6.94 points for Qwen3-4B-Instruct, substantially outperforming the best baseline *GRACE* by +3.66 points.

On CoderEval, RepoDistill consistently improves Pass@k metrics on both Python and Java. Specifically, RepoDistill improves Pass@10 by +10.5% on Python (i.e., 32.1% to 42.6%) and +21.4% on Java (i.e., 34.8% to 56.2%), outperforming all retrieval baselines. On LongCodeU, RepoDistill achieves consistent improvements across all tasks, with LDU improving by +0.25. These results indicate that the budget allocation strategies learned on SWE-QA transfer effectively to downstream code generation, preserving critical code snippets for generating correct solutions.

Comparison on Closed-Source LLM. On SWE-QA, RepoDistill (No Training) outperforms all baselines. Specifically, GPT-5.2 with RepoDistill (No Training) achieves an overall score of 43.37, surpassing the best baseline *GRACE* by +0.88 points. Notably, when integrating the trained Qwen3-4B-Instruct as a pre-processing module, RepoDistill (Qwen3-Trained) achieves a score of 43.03, only 0.34 points lower than RepoDistill (No Training), while reducing the cost of GPT-5.2.

For CoderEval and LongCodeU, both variants improve performance. For example, on CoderEval with GPT-5.2, RepoDistill (No Training) improves Pass@10 by +19.8% (Python) and +19.0% (Java), while RepoDistill (Qwen3-Trained) achieves +17.0% (Python) and +15.2% (Java).

These results demonstrate that a lightweight open-source LLM can serve as an effective context compressor for closed-source LLMs, reducing costs while maintaining competitive performance.

4.2 Ablation Studies

To investigate the role of each component in RepoDistill, we conduct ablation studies on Qwen3-

Table 2: Comparative performance of different methods on CoderEval and LongCodeU (see Appendix § A.2.1 for additional LLMs). We exclude RAG-level comparisons as LongCodeU supplies fixed pre-retrieved contexts.

Method	CoderEval-Python			CoderEval-Java			LongCodeU		
	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10	CUSA	DRA	LDU
Qwen3-4B-Instruct	28.1%	31.5%	32.1%	30.5%	33.2%	34.8%	0.58	0.59	0.46
+ Function Chunking RAG	34.3%	38.6%	39.1%	46.2%	50.5%	51.8%	-	-	-
+ Sliding Window RAG	30.4%	35.3%	36.8%	45.1%	49.2%	50.5%	-	-	-
+ RepoScope	35.6%	38.1%	38.2%	47.5%	51.8%	53.2%	-	-	-
+ GRACE	35.2%	38.4%	39.3%	47.1%	51.4%	52.9%	-	-	-
+ RepoDistill	**37.8%**	**40.0%**	**42.6%**	**50.8%**	**54.5%**	**56.2%**	**0.65**	**0.69**	**0.71**
GPT-5.2	42.5%	49.3%	51.9%	47.5%	56.6%	59.6%	0.76	0.70	0.82
+ Function Chunking RAG	49.9%	64.4%	65.9%	54.6%	68.2%	71.7%	-	-	-
+ Sliding Window RAG	50.2%	63.1%	66.3%	57.0%	65.2%	68.0%	-	-	-
+ RepoScope	50.9%	65.2%	67.5%	58.1%	67.1%	69.2%	-	-	-
+ GRACE	50.1%	64.4%	66.7%	57.1%	66.1%	68.1%	-	-	-
+ RepoDistill (No Training)	**64.6%**	**68.2%**	**71.7%**	**67.4%**	**75.5%**	**78.6%**	**0.82**	**0.73**	**0.89**
+ RepoDistill (Qwen3-Trained)	60.1%	65.8%	68.9%	62.2%	71.2%	74.8%	0.81	0.72	0.87

4B-Instruct across SWE-QA, CoderEval, and LongCodeU benchmarks. Table 3 presents the results.

Table 3: Ablation study on Qwen3-4B-Instruct. CoderEval results are reported as Pass@10. "Vanilla RAG" refers to *Function Chunking RAG*.

Method	SWE-QA	CoderEval		CUSA
		Python	Java	
Vanilla RAG	29.73	39.1%	51.8%	-
GRACE	31.53	39.3%	51.4%	-
RepoDistill	**35.19**	**42.6%**	**56.2%**	**0.65**
+ w/ LongCodeZip	34.77	40.7%	54.6%	0.63
+ w/ Context-Picker	33.96	41.2%	53.9%	0.63
+ w/ Vanilla RAG	32.98	40.7%	53.6%	-
+ w/ GRACE	**35.27**	**43.0%**	**56.9%**	-
- w/o CABA	34.02	41.6%	54.1%	0.63
- w/o CAPO	32.58	41.2%	53.7%	0.61

Plug-and-Play GraphRAG. We first compare RepoDistill with two retrieval baselines: *Function Chunking RAG* (*Vanilla RAG*) and *GRACE*. RepoDistill consistently outperforms these baselines across all benchmarks. We further investigate whether the *GraphRAG* can be replaced with other methods. When substituting *GraphRAG* with *Vanilla RAG* or *GRACE*, we observe performance improvements over the original RAG baselines, demonstrating that our CABA and CAPO modules provide consistent gains regardless of the underlying retrieval method.

Notably, while RepoDistill with *GRACE* achieves the best overall performance, the gap compared to RepoDistill with lightweight *GraphRAG* is marginal, suggesting that our plug-and-play design achieves competitive results without requiring complex graph construction.

Impact of CABA. Removing CABA (w/o CABA) means making binary decisions to either filter or retain entire functions. This degrades performance (35.19 to 34.02 on SWE-QA, 42.6% to 41.6% on CoderEval-Python, and 56.2% to 54.1% on CoderEval-Java), demonstrating that CABA effectively mitigates noise within functions by selectively retaining task-relevant segments. We further replace CABA with LongCodeZip, the results show that RepoDistill with LongCodeZip (34.77 on SWE-QA) underperforms RepoDistill with CABA (35.19), indicating that MMR-based method better preserves diverse and complementary code semantics compared to the knapsack-based method.

Impact of CAPO. Removing CAPO (w/o CAPO) causes performance drops from 35.19 to 32.58 on SWE-QA, from 42.6% to 41.2% on CoderEval-Python, and from 56.2% to 53.7% on CoderEval-Java. This demonstrates that the learned budget allocation policies effectively identify and preserve answer-critical information while aggressively compressing less relevant content. We further replace CAPO with Context-Picker, which performs binary context selection (filter or retain) rather than fine-grained budget allocation. The results show that RepoDistill with Context-Picker (33.96 on SWE-QA) underperforms RepoDistill with CAPO (35.19), indicating that our budget allocation strategy, which simultaneously enables both selection and compression, preserves more task-critical information rather than binary selection.

4.3 Discussion

Performance Across Varying Context Lengths. We evaluate LLM performance across five context-length buckets. As shown in Figure 3, the performance gains from RepoDistill become more pronounced as context length increases. For shorter contexts (e.g., 10k/30k tokens), baselines achieve reasonable performance. However, as context length grows beyond 50k tokens, baseline performance degrades substantially due to increased redundant information, which hampers the LLM's ability to focus on key information. In contrast, RepoDistill maintains stable performance by effectively filtering noise and retaining only task-critical information. Notably, our lightweight *GraphRAG* achieves performance comparable to more sophisticated *GRACE*, indicating that lightweight graph construction is sufficient when combined with effective compression strategies.

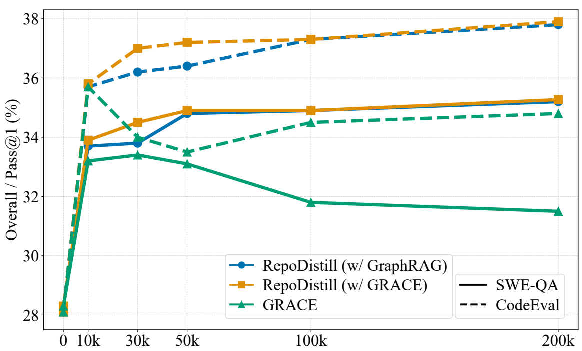

Figure 3: Performance of different methods under varying context lengths on SWE-QA and CoderEval. All methods use Qwen3-4B-Instruct as the base LLM.

Token Reduction Across Varying Context Lengths. RepoDistill can reduce input tokens for closed-source LLMs, lowering API costs while improving performance. As shown in Figure 4, contexts of 10k to 200k tokens are compressed to approximately 4.5k to 67.4k tokens on SWE-QA, and to 5.3k to 74.5k tokens on CoderEval, respectively. Despite this substantial reduction (up to 66% for 200k contexts), compressed contexts consistently yield better downstream performance than uncompressed ones (see Table 1).

4.4 Empirical Lessons

Based on the above experiments, we summarize the following empirical lessons: ❶ **More context is not always better than less context.** Our results show that simply increasing context length

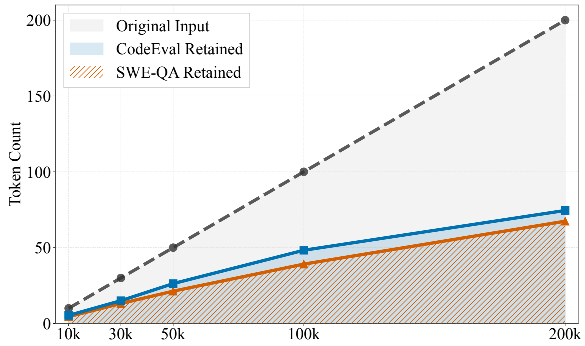

Figure 4: Token reduction of RepoDistill under varying context lengths on the SWE-QA and CoderEval. All results use Qwen3-4B-Instruct as the base LLM.

can even hurt performance when the additional tokens may introduce noise. By explicitly compressing and restructuring the retrieved context, RepoDistill helps LLMs focus on salient information. ❷ **Compression benefits scale with context length.** Performance gains from compression are most significant for longer contexts (e.g., >50K tokens), where models are easily distracted by irrelevant details. For repository-level tasks, intelligent compression is essential for both efficiency and accuracy. ❸ **Lightweight compressors can enhance expensive LLMs.** Qwen3-4B-Instruct trained with RepoDistill serves as an effective pre-processor for closed-source LLMs. RepoDistill (Qwen3-Trained) achieves 43.03 on SWE-QA with GPT-5.2, only marginally lower than RepoDistill (No Training), while significantly reducing computational costs. For cost-sensitive deployments, combining a lightweight compressor with powerful closed-source LLMs offers an attractive performance-efficiency trade-off.

5 Related Works

Long Context LLMs. Recent studies have explored diverse methods to extend LLMs' context windows. Fine-tuning on long sequences (Wu et al., 2021) is a direct method but often costly. Some methods involve additional fine-tuning with downscaled position indices to match the original context window (Xiong et al., 2024; Chen et al., 2023, 2024; Peng et al.). Training-free methods use window attention to handle long sequences (Han et al., 2023; Ding et al., 2023; Xiao et al.), while others modify relative distances to extend extrapolation length (Zhang et al., 2024; Jin et al., 2024).

Reinforcement Learning for LLMs. The evolution of RL-based LLM training has progressed from human preference alignment (Ouyang et al., 2022) toward rule-based verification (Bai et al., 2022), yielding significant improvements in reasoning tasks (OpenAI, 2024; Guo et al., 2025; DeepMind, 2024). Foundational algorithms such as PPO (Schulman et al., 2017) and its variants (e.g., GRPO (Shao et al., 2024b)) have been widely adopted, with subsequent researches (Hu, 2025; Yu et al., 2025; Liu et al., 2025c) addressing their training stability and efficiency. Beyond single-turn RL, multi-turn RL methods have emerged for training tool-augmented agents (Jin et al., 2025; Ouyang et al., 2025a; Wang et al., 2025c), typically through interleaved tool-response sequences. GiGPO (Feng et al., 2025) extends this paradigm to handle multiple parallel contexts with trajectory windowing.

6 Conclusion

In this paper, we introduced RepoDistill, a framework for repository-level code understanding. RepoDistill employs GraphRAG to retrieve structurally relevant contexts, and applies Compression-Aware Budget Allocation with Compression-Aware Policy Optimization to dynamically allocate retention budgets while preserving answer-critical information. Experiments on SWE-QA, CoderEval, and LongCodeU demonstrate consistent improvements, with gains of up to +7.00 (SWE-QA), +24.4% Pass@10 (CoderEval), and +0.25 LDU (LongCodeU). Furthermore, a compact 4B-parameter model trained with RepoDistill can serve as an effective context compressor for closed-source LLMs, reducing input tokens by up to 66% while maintaining comparable performance. These results show that RepoDistill offers a scalable and cost-effective solution for long-context code tasks.

Limitations

Despite the promising results, our work has several limitations:

- **Dependency on retrieval quality.** RepoDistill relies on the initial quality of GraphRAG retrieval. If the retrieval module fails to capture relevant code segments due to complex implicit dependencies or ambiguous queries, the compression module cannot recover missing information.
- **Training overhead.** Training the compression policy with GRPO incurs higher computational costs compared to standard supervised fine-tuning. Although inference remains efficient, the training phase requires significant GPU resources to explore the action space of budget allocation.
- **Language coverage.** Our evaluation focuses on Python and Java repositories. While the principles of RepoDistill are language-agnostic, applying it to languages with different syntax or dependency structures (e.g., C++ or Rust) may require adapting the graph construction and chunking strategies.

Future work will focus on improving training efficiency and extending the framework to support more programming languages and tasks.

Ethical Statement

Our research employs publicly available models and datasets with proper citations, which helps ensure transparency and reproducibility. This design minimizes the risk of generating toxic content by leveraging widely used benchmarks and carefully curated prompts.

Acknowledgements

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198 (SD2)), the National Natural Science Foundation of China (No.62572429), Ningbo Global in Innovation Center, Zhejiang University, the Fundamental Research Funds for the Central Universities (No.226-202200064), Zhejiang Provincial Natural Science Foundation of China (No.LY24F020008), the Key Research and Development Program of Zhejiang Province (No.2021C01105), the State Street Zhejiang University Technology Center.

References

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13.

Anthropic. 2025. Introducing claude sonnet 4.5.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Guanzheng Chen, Xin Li, Zaiqiao Meng, Shangsong Liang, and Lidong Bing. Clex: Continuous length extrapolation for large language models. In *The Twelfth International Conference on Learning Representations*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. 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. 2024. Fortify the shortest stave in attention: Enhancing context awareness of large language models for effective tool use. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11160–11174.

Google DeepMind. 2024. Gemini 2.0 flash thinking.

Google DeepMind. 2025. Gemini 3 pro: Best for complex tasks and bringing creative concepts to life.

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*.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025. Group-in-group policy optimization for llm agent training. *arXiv preprint arXiv:2505.10978*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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*.

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. 2024. Llm maybe longlm: Selfextend llm context window without tuning. In *Proceedings of the 41st International Conference on Machine Learning*, pages 22099–22114.

Jia Li, Xuyuan Guo, Lei Li, Kechi Zhang, Ge Li, Zhengwei Tao, Fang Liu, Chongyang Tao, Yuqi Zhu, and Zhi Jin. 2025. Benchmarking long-context language models on long code understanding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27309–27327.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Yang Liu, Li Zhang, Fang Liu, Zhuohang Wang, Donglin Wei, Zhishuo Yang, Kechi Zhang, Jia Li, and Lin Shi. 2025b. Reposcope: Leveraging call chain-aware multi-view context for repository-level code generation. *arXiv preprint arXiv:2507.14791*.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025c. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1611–1622.

OpenAI. 2024. Learning to reason with llms.

OpenAI. 2025. Introducing gpt-5.2.

Jie Ouyang, Ruiran Yan, Yucong Luo, Mingyue Cheng, Qi Liu, Zirui Liu, Shuo Yu, and Daoyu Wang. 2025a. Training powerful llm agents with end-to-end reinforcement learning.

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

Processing Systems, volume 35, pages 27730–27744. Curran Associates, Inc.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025b. Repograph: Enhancing ai software engineering with repository-level code graph. In *13th International Conference on Learning Representations, ICLR 2025*, pages 30361–30384. International Conference on Learning Representations, ICLR.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. 2025a. Swe-qa: Can language models answer repository-level code questions? *arXiv preprint arXiv:2509.14635*.

Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. 2025b. Soleval: Benchmarking large language models for repository-level solidity smart contract generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 4388–4411.

Qwen. 2025a. Qwen3-coder: Agentic coding in the world.

Qwen. 2025b. Qwen3-max: Just scale it.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, and 1 others. 2025. Seed1. 5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024a. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. 2024b. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2025. Longcodezip: Compress long context for code language models. *arXiv preprint arXiv:2510.00446*.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894. IEEE.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Xingliang Wang, Baoyi Wang, Chen Zhi, Junxiao Han, Xinkui Zhao, Jianwei Yin, and Shuiguang Deng. 2025a. Grace: Graph-guided repository-aware code completion through hierarchical code fusion. *arXiv preprint arXiv:2509.05980*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025b. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1140–1152. IEEE.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. 2025c. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *Preprint*, arXiv:2504.20073.

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. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*.

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, and 1 others. 2024. Effective long-context scaling of foundation models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4643–4663.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2025. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23762–23779.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others.

2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

Xin Yin, Chao Ni, Xinrui Li, Liushan Chen, Guojun Ma, and Xiaohu Yang. Enhancing llm's ability to generate more repository-aware unit tests through precise context injection.

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Kechi Zhang, Ge Li, Huangzhao Zhang, and Zhi Jin. 2024. 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.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, and 1 others. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Siyuan Zhu, Chengdong Xu, Kaiqiang Ke, and Chao Yu. 2025. Context-picker: Dynamic context selection using multi-stage reinforcement learning. *arXiv preprint arXiv:2512.14465*.

A Appendix

A.1 Experimental Details

A.1.1 Human-in-the-loop SFT

To ensure high-quality training data, we employ two annotators with software engineering backgrounds (3+ years of experience) to curate the SFT dataset. For each training instance, we sample 10 candidate trajectories from Qwen3-Max. The annotators independently evaluate all candidates and select the highest-quality trajectory based on: (1) answer correctness, (2) budget allocation reasonableness, and (3) memory summary quality. Disagreements are resolved through discussion. Inter-annotator agreement on instances achieves Cohen's $\kappa = 0.78$, indicating substantial agreement.

A.1.2 Prompt for Compression and Generation Phases

Table 4 presents the prompt templates for the two-phase pipeline of RepoDistill: context compression and final answer generation.

Context-Compression Prompt. This prompt guides the LLM to analyze retrieved code chunks and determine appropriate compression levels for each document. Given a problem statement, a chunk of code context, and a memory summary from previous chunks, the model assigns one of five compression levels (0%, 25%, 50%, 75%, or 100%) to each document based on its relevance to the problem. As for memory summary specifications, the model needs to update the memory summary to capture salient information across chunks and maintain coherence across turns. The output format requires a valid JSON object containing: (1) compression levels for each document, and (2) an updated memory summary (max 200 words) that synthesizes key findings from the current chunk while preserving cross-turn context continuity.

Answer-Generation Prompt. This prompt instructs the LLM to synthesize the final answer based on the compressed repository memory and the original problem statement. The prompt emphasizes direct and concise output without extraneous code snippets or explanations, ensuring that the model focuses on generating precise answers to the repository-level questions.

A.1.3 Example Outputs of Compression and Generation Phases

Table 5 illustrates the model's outputs during the compression and generation phases. In the compression phase, the model outputs a JSON object containing: (1) `compressed_documents`, which maps each document to its assigned retention budget (e.g., 0.75 indicates 75% retention), and (2) `updated_summary`, which captures salient information from the processed chunks for subsequent reasoning. In the generation phase, the model generates answers based on the compressed contexts.

A.1.4 Prompt for LLM-as-judge

Table 6 presents the evaluation prompt for the LLM-as-judge framework used to assess answer quality on SWE-QA.

Evaluation Criteria. The prompt instructs the judge model to evaluate candidate answers against reference answers across five dimensions: correctness, completeness, relevance, clarity, and reasoning quality. Each dimension is scored on a scale from 0.0 to 10.0, with detailed scoring guidelines provided for each score range (8.0-10.0, 6.0-7.9, 4.0-5.9, 2.0-3.9, 0.0-1.9). The guidelines specify the characteristics of answers at each performance level, ensuring consistent and reliable evaluation.

Output Format. The judge model is required to output a valid JSON object containing five floating-point fields corresponding to the evaluation scores for each dimension. The prompt explicitly requires no additional text, explanations, or formatting beyond the valid JSON output, ensuring clean and parseable results for automated scoring.

A.1.5 Detail of Datasets

We evaluate RepoDistill on three code-related benchmarks covering repository-level QA, code generation, and long-context understanding:

- **SWE-QA** (Peng et al., 2025a): A repository-level QA benchmark containing 576 high-quality question-answer pairs that span intention understanding, cross-file reasoning, and multi-hop dependency analysis.

- **CoderEval** (Yu et al., 2024): A pragmatic code generation benchmark comprising 230 Python and 230 Java tasks from open-source projects, covering six levels of context dependency.

- **LongCodeU** (Li et al., 2025): A long code understanding benchmark with contexts up to 128K tokens, evaluating LLMs across code unit perception, intra- and inter-code unit understanding, and documentation analysis.

A.1.6 Detail of Baselines

We adopt seven baselines for context retrieval and compression, comprising one direct LLM method, two semantic similarity-based retrieval methods, two graph-based methods, one context compression method, and one context selection method:

- **Direct**: The model receives only the task instruction without retrieved context, serving as a baseline to quantify the gains from retrieval-augmented generation.

- **Function Chunking RAG** (Wang et al., 2025b): This method parses the repository into function-level chunks and retrieves the relevant chunks based on semantic similarity.

- **Sliding Window RAG** (Zhang et al.): This method divides files into overlapping segments using a sliding window and retrieves the relevant segments based on semantic similarity.

- **RepoScope** (Liu et al., 2025b): This method constructs a Repository Structural Semantic Graph (RSSG) and retrieves a comprehensive four-view context, integrating both structural and similarity-based information.

- **GRACE** (Wang et al., 2025a): This method constructs a multi-level code graph unifying file structures, ASTs, call graphs, and data flow graphs, and employs a hybrid graph retriever combining GNN-based structural similarity with textual retrieval.

- **LongCodeZip** (Shi et al., 2025): This method employs a dual-stage compression strategy: coarse-grained filtering to retain relevant functions, followed by fine-grained block selection under a fixed token budget.

- **Context-Picker** (Zhu et al., 2025): This method formulates context selection as a decision-making process optimized via two-stage reinforcement learning, balancing recall-oriented coverage and precision-oriented pruning to distill minimal sufficient context sets.

A.2 Additional Experiments

A.2.1 Main Results (Additional LLMs)

Comparative performance of different models on SWE-QA. Table 7 presents comprehensive performance comparisons across six LLMs, including three open-source LLMs (i.e., Qwen3-4B-Instruct, Qwen3-30B-A3B-Instruct, and Qwen3-Coder-30B-A3B-Instruct) and three closed-source LLMs (i.e., GPT-5.2, Claude-Sonnet-4.5, and Gemini-3-Pro).

For open-source LLMs, RepoDistill consistently outperforms both retrieval-augmented baselines across all evaluation dimensions. Notably, Qwen3-30B-A3B-Instruct achieves the largest overall improvement of +7.00 points over the vanilla model, with substantial gains in completeness (+2.70) and reasoning (+1.60). The results demonstrate that RepoDistill effectively learns to identify and preserve answer-critical information through reinforcement-guided budget allocation.

For closed-source LLMs, we observe two interesting phenomena. First, RepoDistill (No Training) consistently achieves the best performance, surpassing *Function Chunking RAG* and *Sliding Window RAG* across all models. For instance, Gemini-3-Pro with RepoDistill (No Training) achieves an overall score of 42.61, outperforming *Function Chunking RAG* by 2.57 points. Second, the lightweight Qwen3-4B-Instruct trained with RepoDistill can serve as an effective pre-processor for powerful closed-source LLMs. RepoDistill (Qwen3-Trained) achieves comparable performance to RepoDistill (No Training), with performance gaps within 1 points across all three closed-source LLMs. This finding suggests that a small LLM can effectively serve as a cost-efficient context compressor for expensive closed-source LLMs.

Comparative performance of different models on CoderEval and LongCodeU. Table 8 extends the evaluation to additional LLMs including Qwen3-30B-A3B-Instruct, Qwen3-Coder-30B-A3B-Instruct, Claude-Sonnet-4.5, and Gemini-3-Pro, providing a comprehensive comparison across diverse LLM architectures.

For open-source LLMs, RepoDistill outperforms both *Function Chunking RAG* and *Sliding Window RAG* across all benchmarks. On Qwen3-Coder-30B-A3B-Instruct, RepoDistill achieves Pass@10 improvements of +25.6% on CoderEval-Python and +23.6% on CoderEval-Java compared to the baseline, demonstrating the transferability of budget allocation strategies learned on SWE-QA to code generation tasks. On LongCodeU, RepoDistill improves CUSA by +0.09, DRA by +0.09, and LDU by +0.09, highlighting its effectiveness in identifying and preserving task-critical information across different evaluation dimensions.

For closed-source LLMs, both RepoDistill variants enhance performance. On Claude-Sonnet-4.5, RepoDistill (No Training) improves Pass@10 by +20.6% (Python) and +19.6% (Java), while RepoDistill (Qwen3-Trained) achieves +17.8% (Python) and +15.9% (Java). On Gemini-3-Pro, RepoDistill (No Training) achieves Pass@10 improvements of +24.1% (Python) and +24.4% (Java), with LongCodeU scores improving by up to +0.12 on CUSA. Notably, RepoDistill (Qwen3-Trained) achieves performance comparable to RepoDistill (No Training) across most metrics.

A.3 Data Containing Personally Identifying Information or Offensive Content

To ensure the ethical integrity of our research, we carefully examined the data used for RepoDistill to verify that it does not contain any personally identifying information or offensive content. The data used in our experiments is derived from publicly available open-source code repositories, with no inclusion of private or sensitive personal information. We specifically focused on the code and its associated documentation, ensuring that any metadata related to individual contributors or personal identifiers was excluded. Additionally, we employed a manual review process to identify and filter any potentially offensive content within the code, comments, or documentation. This process helps maintain the privacy and safety of individuals and ensures the ethical use of the data in our research. Any identified offensive or sensitive content was removed prior to inclusion in the experiments.

A.4 Human Annotations

We employ two annotators with software engineering backgrounds (3+ years of experience) to curate the SFT dataset. The participants are compensated at a rate of approximately $20 per hour, which is considered fair given their expertise and the local cost of living. This compensation fairly acknowledges the time and effort required for manual annotation tasks while ensuring that the work meets the standards expected in academic research.

A.5 Data Consent

In this study, all data used for RepoDistill was collected from publicly available open-source datasets. These datasets are openly accessible, and the data extracted for the purpose of this research does not involve any private or proprietary information.

Table 4: Prompt of RepoDistill for context compression (top part) and final answer generation (bottom part)

Context-Compression Prompt

You are provided with a **problem**, a **chunk** of code context and a **previous memory** for previous chunks. Your task is to analyze each document in the provided chunk and determine the appropriate compression level needed to preserve the essential information for solving the given problem, while removing redundant content. For each `document_id` in the chunk, assign a compression level from the following options:

- 0% : Fully filtered (empty)
- 25% : Aggressive compression (essential information only)
- 50% : Balanced compression (core content retained)
- 75% : Light compression (key context preserved)
- 100% : Original text (no compression)

`<problem> {problem} </problem>`

`<memory> {memory} </memory>`

`<chunk> {chunk} </chunk>`

OUTPUT FORMAT REQUIREMENT:

You must output a valid JSON object with the following structure:

{"compressed_documents": {"Document_0": x, "Document_1": y, "Document_2": z, ...}, "updated_summary": "A concise summary (max 200 words) that: (1) captures salient information from the current chunk, and (2) maintains coherence with previously processed chunks."}

Answer-Generation Prompt

You are presented with a **problem**, and a **repository memory**. Your task is to directly answer the problem based on the provided repository memory and problem statement.

`<problem> {problem} </problem>`

`<repository memory> {repository memory} </repository memory>`

OUTPUT FORMAT REQUIREMENT:

Provide the final answer concisely and directly, without code snippets, extra explanations or commentary.

Table 5: Example output of context compression (middle part) and final answer generation (bottom part)

Example of Question

What is the structure of Scikit-learn's preprocessing module?

Example Output of Context-Compression

{"compressed_documents": {"Document_0": 0.75, "Document_1": 0.25, "Document_2": 0, "Document_3": 1, ...}, "updated_summary": "The provided memory contains source code and documentation from the scikit-learn library, focusing on data preprocessing, utility functions, and recent release updates..."}

Example Output of Answer-Generation

The structure of Scikit-learn's preprocessing module can be described as follows: The module contains a variety of classes and functions for scaling, centering, normalization, binarization, and other preprocessing tasks. It includes classes like `StandardScaler`, `MinMaxScaler`, `MaxAbsScaler`, `RobustScaler`, and `Binarizer`, each designed for specific preprocessing needs Functions such as `scale`, `minmax_scale`, `maxabs_scale`, and `binarize` are also provided for convenience.The module supports both dense and sparse data structures, ensuring flexibility for different types of input. It includes utilities for handling missing values and feature names. The `FunctionTransformer` class allows for custom transformations, making the module highly extensible. The preprocessing module is organized into submodules, such as `_data.py`, `_encoders.py`, and `_function_transformer.py`, each containing related functionality. This structure makes the preprocessing module a versatile and comprehensive tool for preparing data for machine learning models in scikit-learn.

Table 6: Prompt of LLM-as-Judge

LLM-as-Judge Prompt

You are a professional evaluator. Please rate the candidate answer against the reference answer based on five criteria.

Evaluation Criteria and Scoring Guidelines (each scored 0.0 to 10.0):

1. Correctness:

8.0-10.0 - Completely correct; core points and details are accurate with no ambiguity.
6.0-7.9 - Mostly correct; only minor details are slightly inaccurate or loosely expressed.
4.0-5.9 - Partially correct; some errors or omissions, but main points are generally accurate.
2.0-3.9 - Several errors or ambiguities that affect understanding of the core information.
0.0-1.9 - Serious errors; misleading or fails to convey key information.

2. Completeness:

8.0-10.0 - Covers all key points from the reference answer without omission.
6.0-7.9 - Covers most key points; only minor non-critical information missing.
4.0-5.9 - Missing several key points; content is somewhat incomplete.
2.0-3.9 - Important information largely missing; content is one-sided.
0.0-1.9 - Covers very little or irrelevant information; seriously incomplete.

3. Relevance:

8.0-10.0 - Content fully focused on the question topic; no irrelevant information.
6.0-7.9 - Mostly focused; only minor irrelevant or peripheral information.
4.0-5.9 - Topic not sufficiently focused; contains considerable off-topic content.
2.0-3.9 - Content deviates from topic; includes excessive irrelevant information.
0.0-1.9 - Majority of content irrelevant to the question.

4. Clarity:

8.0-10.0 - Fluent language; clear and precise expression; easy to understand.
6.0-7.9 - Mostly fluent; some expressions slightly unclear or not concise.
4.0-5.9 - Expression somewhat awkward; some ambiguity or lack of fluency.
2.0-3.9 - Language obscure; sentences are not smooth; hinders understanding.
0.0-1.9 - Expression confusing; very difficult to understand.

5. Reasoning:

8.0-10.0 - Reasoning is clear, logical, and well-structured; argumentation is solid.
6.0-7.9 - Reasoning generally reasonable; mostly clear logic; minor jumps.
4.0-5.9 - Reasoning is average; some logical jumps or organization issues.
2.0-3.9 - Reasoning unclear; lacks logical order; difficult to follow.
0.0-1.9 - No clear reasoning; logic is chaotic.

INPUT:

Problem: `{problem}`
Reference Answer: `{reference}`
Candidate Answer: `{candidate}`

OUTPUT:

Please output ONLY a JSON object with 5 floating-point fields in the range [0.0, 10.0], corresponding to the evaluation scores: {"correctness": <0.0-10.0>, "completeness": <0.0-10.0>, "relevance": <0.0-10.0>, "clarity": <0.0-10.0>, "reasoning": <0.0-10.0>}

REQUIREMENT:

No explanation, no extra text, no formatting other than valid JSON.

Table 7: Comparative performance of different models on SWE-QA

Model	Evaluation Metrics					Overall
	Correctness	**Completeness**	**Relevance**	**Clarity**	**Reasoning**	
Open-Source LLM						
Qwen3-4B-Instruct	5.12	3.33	7.06	7.54	5.20	28.25
+ Function Chunking RAG	5.48 (+0.36)	4.30 (+0.97)	6.94 (-0.12)	7.39 (-0.15)	5.62 (+0.42)	29.73 (+1.48)
+ Sliding Window RAG	5.60 (+0.48)	4.40 (+1.07)	6.93 (-0.13)	7.36 (-0.18)	5.77 (+0.57)	30.06 (+1.81)
+ RepoScope	5.70 (+0.58)	4.65 (+1.32)	7.14 (+0.08)	7.67 (+0.13)	5.83 (+0.63)	30.99 (+2.74)
+ GRACE	5.79 (+0.67)	4.63 (+1.30)	7.38 (+0.32)	7.58 (+0.04)	6.15 (+0.95)	31.53 (+3.28)
+ RepoDistill	**6.43** (+1.31)	**5.85** (+2.52)	**8.03** (+0.97)	**8.09** (+0.55)	**6.79** (+1.59)	**35.19** (+6.94)
Qwen3-30B-A3B-Instruct	6.40	4.01	8.15	8.40	6.38	33.34
+ Function Chunking RAG	7.20 (+0.80)	5.93 (+1.92)	8.63 (+0.48)	8.40 (+0.00)	7.35 (+0.97)	37.51 (+4.17)
+ Sliding Window RAG	7.44 (+1.04)	5.72 (+1.71)	8.28 (+0.13)	8.54 (+0.14)	7.32 (+0.94)	37.30 (+3.96)
+ RepoScope	7.50 (+1.10)	5.96 (+1.95)	8.75 (+0.60)	8.82 (+0.42)	7.62 (+1.24)	38.65 (+5.31)
+ GRACE	7.56 (+1.16)	6.47 (+2.46)	8.69 (+0.54)	8.52 (+0.12)	7.48 (+1.10)	38.72 (+5.38)
+ RepoDistill	**7.58** (+1.18)	**6.71** (+2.70)	**9.12** (+0.97)	**8.95** (+0.55)	**7.98** (+1.60)	**40.34** (+7.00)
Qwen3-Coder-30B-A3B-Instruct	6.31	5.05	8.18	8.25	6.59	34.38
+ Function Chunking RAG	6.94 (+0.63)	6.53 (+1.48)	8.30 (+0.12)	8.12 (-0.13)	7.04 (+0.45)	36.93 (+2.55)
+ Sliding Window RAG	6.80 (+0.49)	6.94 (+1.89)	8.49 (+0.31)	8.05 (-0.20)	6.84 (+0.25)	37.12 (+2.74)
+ RepoScope	7.03 (+0.72)	7.47 (+2.42)	8.85 (+0.67)	8.45 (+0.20)	6.77 (+0.18)	38.57 (+4.19)
+ GRACE	6.95 (+0.64)	7.54 (+2.49)	8.69 (+0.51)	8.17 (-0.08)	6.71 (+0.12)	38.06 (+3.68)
+ RepoDistill	**7.39** (+1.08)	**7.56** (+2.51)	**9.17** (+0.99)	**8.55** (+0.30)	**7.63** (+1.04)	**40.30** (+5.92)
Closed-Source LLM						
GPT-5.2	6.98	5.51	8.71	8.52	7.16	36.88
+ Function Chunking RAG	8.20 (+1.22)	8.39 (+2.88)	9.23 (+0.52)	8.36 (-0.16)	8.27 (+1.11)	42.45 (+5.57)
+ Sliding Window RAG	7.98 (+1.00)	8.26 (+2.75)	9.14 (+0.43)	8.24 (-0.28)	8.32 (+1.16)	41.94 (+5.06)
+ RepoScope	8.09 (+1.11)	8.42 (+2.91)	9.29 (+0.58)	8.30 (-0.22)	8.36 (+1.20)	42.46 (+5.58)
+ GRACE	8.16 (+1.18)	8.46 (+2.95)	9.18 (+0.47)	8.36 (-0.16)	8.33 (+1.17)	42.49 (+5.61)
+ RepoDistill (No Training)	**8.31** (+1.33)	**8.56** (+3.05)	**9.35** (+0.64)	**8.64** (+0.12)	**8.51** (+1.35)	**43.37** (+6.49)
+ RepoDistill (Qwen3-Trained)	8.25 (+1.27)	8.46 (+2.95)	9.32 (+0.61)	8.56 (+0.04)	8.44 (+1.28)	43.03 (+6.15)
Claude-Sonnet-4.5	6.81	5.14	8.42	8.54	6.98	35.89
+ Function Chunking RAG	7.71 (+0.90)	7.43 (+2.29)	8.98 (+0.56)	8.72 (+0.18)	7.16 (+0.18)	40.00 (+4.11)
+ Sliding Window RAG	7.66 (+0.85)	7.25 (+2.11)	8.86 (+0.44)	8.61 (+0.07)	7.08 (+0.10)	39.46 (+3.57)
+ RepoScope	7.95 (+1.14)	7.72 (+2.58)	8.99 (+0.57)	8.80 (+0.26)	7.42 (+0.44)	40.88 (+4.99)
+ GRACE	8.02 (+1.21)	7.52 (+2.38)	8.84 (+0.42)	8.62 (+0.08)	7.35 (+0.37)	40.35 (+4.46)
+ RepoDistill (No Training)	**8.05** (+1.24)	**7.85** (+2.71)	**9.25** (+0.83)	**8.91** (+0.37)	**7.79** (+0.81)	**41.85** (+5.96)
+ RepoDistill (Qwen3-Trained)	7.92 (+1.11)	8.12 (+2.98)	9.10 (+0.68)	8.82 (+0.28)	7.64 (+0.66)	41.60 (+5.71)
Gemini-3-Pro	7.07	4.01	8.73	8.71	6.48	35.00
+ Function Chunking RAG	7.62 (+0.55)	7.10 (+3.09)	8.90 (+0.17)	8.54 (-0.17)	7.88 (+1.40)	40.04 (+5.04)
+ Sliding Window RAG	7.78 (+0.71)	7.16 (+3.15)	9.01 (+0.28)	8.48 (-0.23)	7.82 (+1.34)	40.25 (+5.25)
+ RepoScope	7.83 (+0.76)	7.30 (+3.29)	9.19 (+0.46)	8.57 (-0.14)	8.17 (+1.69)	41.06 (+6.06)
+ GRACE	8.06 (+0.99)	7.32 (+3.31)	9.17 (+0.44)	8.69 (-0.02)	7.91 (+1.43)	41.15 (+6.15)
+ RepoDistill (No Training)	**8.14** (+1.07)	**7.91** (+3.90)	**9.32** (+0.59)	**8.89** (+0.18)	**8.35** (+1.87)	**42.61** (+7.61)
+ RepoDistill (Qwen3-Trained)	7.95 (+0.88)	7.64 (+3.63)	9.18 (+0.45)	8.75 (+0.04)	8.12 (+1.64)	41.64 (+6.64)

Table 8: Comparative performance of different models on CoderEval and LongCodeU

Model	CoderEval-Python			CoderEval-Java			LongCodeU		
	Pass@1	**Pass@5**	**Pass@10**	**Pass@1**	**Pass@5**	**Pass@10**	**CUSA**	**DRA**	**LDU**
Open-Source LLM									
Qwen3-4B-Instruct	28.1%	31.5%	32.1%	30.5%	33.2%	34.8%	0.58	0.59	0.46
+ Function Chunking RAG	34.3%	38.6%	39.1%	46.2%	50.5%	51.8%	-	-	-
+ Sliding Window RAG	30.4%	35.3%	36.8%	45.1%	49.2%	50.5%	-	-	-
+ RepoScope	35.6%	38.1%	38.2%	47.5%	51.8%	53.2%	-	-	-
+ GRACE	35.2%	38.4%	39.3%	47.1%	51.4%	52.9%	-	-	-
+ RepoDistill	**37.8%**	**40.0%**	**42.6%**	**50.8%**	**54.5%**	**56.2%**	**0.65**	**0.69**	**0.71**
Qwen3-30B-A3B-Instruct	31.6%	35.8%	37.2%	36.5%	39.2%	40.8%	0.70	0.71	0.76
+ Function Chunking RAG	32.1%	38.7%	40.2%	55.4%	59.5%	60.8%	-	-	-
+ Sliding Window RAG	33.5%	38.2%	40.6%	54.8%	58.1%	59.5%	-	-	-
+ RepoScope	36.1%	40.0%	42.9%	56.2%	60.2%	61.5%	-	-	-
+ GRACE	35.6%	39.1%	42.8%	55.9%	59.8%	61.2%	-	-	-
+ RepoDistill	**38.7%**	**45.2%**	**48.3%**	**59.8%**	**62.5%**	**63.8%**	**0.78**	**0.75**	**0.83**
Qwen3-Coder-30B-A3B-Instruct	31.7%	34.4%	34.8%	37.0%	39.6%	41.3%	0.65	0.70	0.74
+ Function Chunking RAG	37.8%	45.2%	47.0%	57.8%	61.7%	62.2%	-	-	-
+ Sliding Window RAG	36.2%	43.6%	46.2%	57.1%	60.1%	62.0%	-	-	-
+ RepoScope	38.7%	45.2%	46.9%	58.3%	61.9%	62.6%	-	-	-
+ GRACE	38.1%	45.5%	47.4%	58.0%	61.7%	61.8%	-	-	-
+ RepoDistill	**41.7%**	**48.7%**	**49.1%**	**62.6%**	**63.9%**	**64.9%**	**0.74**	**0.79**	**0.83**
Closed-Source LLM									
GPT-5.2	42.5%	49.3%	51.9%	47.5%	56.6%	59.6%	0.76	0.70	0.82
+ Function Chunking RAG	49.9%	64.4%	65.9%	54.6%	68.2%	71.7%	-	-	-
+ Sliding Window RAG	50.2%	63.1%	66.3%	57.0%	65.2%	68.0%	-	-	-
+ RepoScope	50.9%	65.2%	67.5%	58.1%	67.1%	69.2%	-	-	-
+ GRACE	50.1%	64.4%	66.7%	57.1%	66.1%	68.1%	-	-	-
+ RepoDistill (No Training)	**64.6%**	**68.2%**	**71.7%**	**67.4%**	**75.5%**	**78.6%**	**0.82**	**0.73**	**0.89**
+ RepoDistill (Qwen3-Trained)	60.1%	65.8%	68.9%	62.2%	71.2%	74.8%	0.81	0.72	0.87
Claude-Sonnet-4.5	39.1%	45.4%	47.7%	43.7%	52.0%	54.6%	0.70	0.73	0.85
+ Function Chunking RAG	46.4%	58.5%	61.2%	50.8%	62.7%	65.9%	-	-	-
+ Sliding Window RAG	46.7%	57.9%	61.5%	53.2%	59.9%	62.4%	-	-	-
+ RepoScope	47.4%	59.3%	62.7%	54.3%	61.8%	63.7%	-	-	-
+ GRACE	46.6%	58.5%	61.9%	53.3%	60.8%	62.6%	-	-	-
+ RepoDistill (No Training)	**61.2%**	**64.9%**	**68.3%**	**63.6%**	**71.3%**	**74.2%**	**0.80**	**0.76**	**0.88**
+ RepoDistill (Qwen3-Trained)	56.8%	62.5%	65.5%	58.4%	67.1%	70.5%	0.78	0.75	0.87
Gemini-3-Pro	40.4%	46.9%	49.3%	45.1%	53.7%	56.4%	0.72	0.74	0.83
+ Function Chunking RAG	47.7%	60.2%	62.8%	52.2%	64.5%	67.8%	-	-	-
+ Sliding Window RAG	48.0%	59.5%	63.1%	54.6%	61.6%	64.2%	-	-	-
+ RepoScope	48.7%	61.0%	64.4%	55.7%	63.5%	65.5%	-	-	-
+ GRACE	47.9%	60.2%	63.6%	54.7%	62.5%	64.4%	-	-	-
+ RepoDistill (No Training)	**65.8%**	**69.7%**	**73.4%**	**68.3%**	**76.6%**	**80.8%**	**0.84**	**0.79**	**0.89**
+ RepoDistill (Qwen3-Trained)	60.2%	65.3%	68.6%	63.1%	71.2%	74.9%	0.80	0.76	**0.89**