

LongCodeZip: Compress Long Context for Code Language Models

Yuling Shi¹, Yichun Qian², Hongyu Zhang³, Beijun Shen¹, Xiaodong Gu^{1*}

¹Shanghai Jiao Tong University, Shanghai, China

²Stanford University, Stanford, CA, USA

³Chongqing University, Chongqing, China

{yuling.shi, bjshen, xiaodong.gu}@sjtu.edu.cn, ycqian@stanford.edu, hyzhang@cqu.edu.cn

Abstract—Code generation under long contexts is becoming increasingly critical as Large Language Models (LLMs) are required to reason over extensive information in the code-base. While recent advances enable code LLMs to process long inputs, high API costs and generation latency remain substantial bottlenecks. Existing context pruning techniques, such as LLMLingua, achieve promising results for general text but overlook code-specific structures and dependencies, leading to suboptimal performance in programming tasks. In this paper, we propose LongCodeZip, a novel plug-and-play code compression framework designed specifically for code LLMs. LongCodeZip employs a dual-stage strategy: (1) coarse-grained compression, which identifies and ranks function-level chunks using conditional perplexity with respect to the instruction, retaining only the most relevant functions; and (2) fine-grained compression, which segments retained functions into blocks based on perplexity and selects an optimal subset under an adaptive token budget to maximize relevance. Evaluations across multiple tasks, including code completion, summarization, and question answering, show that LongCodeZip consistently outperforms baseline methods, achieving up to a 5.6x compression ratio without degrading task performance. By effectively reducing context size while preserving essential information, LongCodeZip enables LLMs to better scale to real-world, large-scale code scenarios, advancing the efficiency and capability of code intelligence applications’.

I. INTRODUCTION

LLMs specialized for code have revolutionized software development by demonstrating remarkable capabilities in code completion [1], [2], translation [3], [4], program synthesis [5], [6], [7] and program repair [8], [9]. Models like DeepSeek-Coder [10], Qwen2.5-Coder [11], Seed-Coder [12] can reason over diverse programming languages and significantly enhance productivity. As code LLMs are increasingly deployed for real-world tasks like repository-level question answering [13] and long-context code completion [1], there is a growing demand for handling contexts that span tens of thousands of tokens. This need has motivated efforts to extend LLM context windows [14], [11], [15]. However, effective handling of long code contexts remains a central bottleneck. Three major challenges arise in such long code context scenarios. First, as the input context grows, the quadratic complexity of the transformer attention mechanism [16] leads to decreased

generation efficiency. At the same time, processing longer inputs with LLMs results in rapidly increasing API costs, especially when pricing models are expensive [17], [18]. Second, LLMs struggle to identify and utilize relevant content amid lengthy inputs [19], [20]. Third, even though recent LLMs support extended context windows to 128k tokens, these limits can still be reached when processing large files and long conversation histories, leading to context truncation and degraded outputs [21].

These issues are particularly pronounced in code LLMs. Unlike natural language text, source code is highly structured with complex dependencies spanning across functions, classes, and files. Dependencies between variable declarations, function definitions, and their uses often extend beyond what current context windows can accommodate. As a result, LLMs frequently produce code that fails to compile, violates existing patterns, or ignores critical constraints when the relevant context exceeds their window size [22]. Consequently, context compression has emerged as a key demand for enabling long-context code understanding.

Existing approaches to address long context limitations have notable shortcomings when applied to code. General text compression methods like LLMLingua [23] and Selective Context [24] fail to account for code-specific characteristics and often break code structure. Retrieval-augmented generation (RAG) [25] reduce context length by selecting relevant code snippets from the repository context, but it merely rely on text similarities, and may overlook implicit dependencies within the context. Traditional code compressors such as DietCode [26] and SlimCode [27] improve syntax and structure awareness but are generally limited to function-level pruning or short code examples, leaving compression of long context for code largely unaddressed.

To overcome these limitations, we introduce LongCodeZip, a training-free, model-agnostic, and plug-and-play context compression framework for code LLMs. Our approach leverages the inherent structure of code through a novel two-stage compression strategy that preserves code semantics while significantly reducing token consumption. First, we perform coarse-grained compression by identifying and ranking function-level chunks based on their relevance to the instruction. Then, within the selected functions, it applies perplexity-based block detection followed by fine-grained block-level

¹Our code and data are available at <https://github.com/YerbaPage/LongCodeZip>

* Xiaodong Gu is the corresponding author

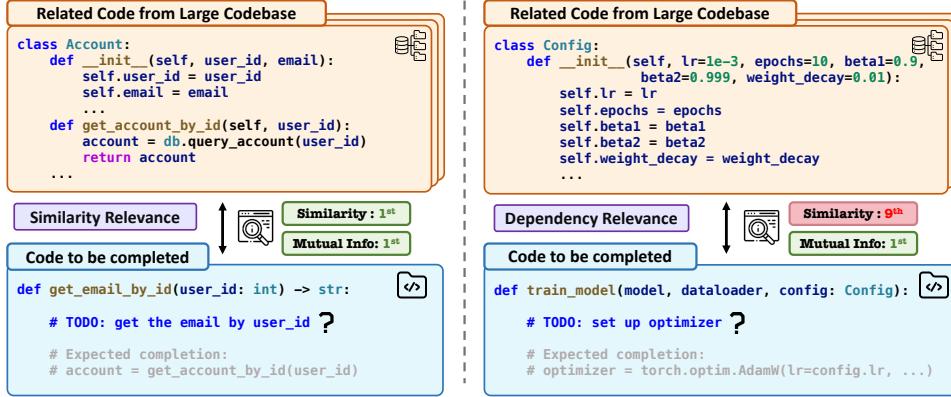


Fig. 1: Challenge for RAG, a similarity-based context compression method.

compression to maximize relevance under an adaptive token budget. To the best of our knowledge, LongCodeZip is the first framework specifically designed for long-context code compression and to introduce perplexity-based block detection, providing an efficient and general-purpose solution that preserves task-critical content within strict token limitations.

We evaluate LongCodeZip across multiple code benchmarks with long contexts, including Long Code Completion [1], Long Module Summarization [21], and RepoQA [13]. Results demonstrate that our approach achieves up to a 5.6 \times compression ratio without sacrificing performance, generalizes well across tasks and models (even with only 0.5B model as the compressor), and significantly reduces generation time and token costs.

Our main contributions include:

- 1) A novel long-context, code-specific hierarchical compression approach that performs function-level chunking and selection, followed by perplexity-based block detection and block-level pruning.
- 2) An adaptive budget allocation and 0/1 knapsack selection mechanism that prioritizes relevant blocks and maximizes critical detail within the token budget.
- 3) A comprehensive evaluation demonstrating that LongCodeZip outperforms baselines on code completion, summarization, and question answering tasks, achieving up to a 5.6 \times compression ratio without sacrificing performance.

II. MOTIVATION

Code generation under long context is becoming increasingly important in LLM-based software development. Such tasks often require referencing numerous related files across an entire project repository, resulting in input contexts that span tens of thousands of tokens. However, these long contexts typically contain scattered and redundant information, which can distract the model and degrade output quality. Moreover, the substantial computational cost of processing such large inputs further exacerbates latency and resource constraints, creating a significant bottleneck for practical deployment.

Retrieval-augmented generation (RAG) [28], [29] provides an efficient way to condense overly lengthy contexts. RAG

retrieves and appends relevant code snippets to the prompt, leveraging embedding models such as UniXcoder [30] or CodeBERT [31], and similarity measures such as cosine similarity. While RAG effectively reduces context length, it primarily relies on surface-level lexical similarity between snippets. Consequently, it often fails to capture code segments with deeper semantic or functional dependencies—particularly when such relationships are implicit, abstracted, or span multiple components.

Consider the examples in Figure 1. In the first scenario, the task is to complete an `get_email_by_id` function. Retrieving `Account` class and the `get_account_by_id` function proves effective, as they share similar function and parameter names. In this case, RAG works well due to strong lexical and structural overlap. In the second scenario, however, the task is to implement a `train_model` function that relies on configuration values defined in a separate `Config` class. Here, crucial context like `Config` is often missed, since RAG may not identify these implicit or non-lexical dependencies. This omission can lead to incomplete or incorrect code generation.

This example highlights the need for context selection criteria that extend beyond surface-level similarity. In both scenarios, an effective similarity measure should assign high relevance to `get_account_by_id` in the first case and, critically, to `Config` in the second—even when there is minimal lexical overlap between the configuration class and the training function.

III. METHODOLOGY

A. Problem Formulation

Given a long code context $c = \{c_1, \dots, c_n\}$ with n tokens and a task instruction $q = \{q_1, \dots, q_m\}$, the goal of context compression is to produce a compressed context $c' \subseteq c$ such that $|c'| \leq B$, where B is the computational budget in tokens. The objective is to maximize task performance while satisfying the budget constraint. For instance, in the code completion task, the instruction could be: “Complete the following function [code to be completed]”. The long context could consist of the unfinished code along with retrieved code snippets.

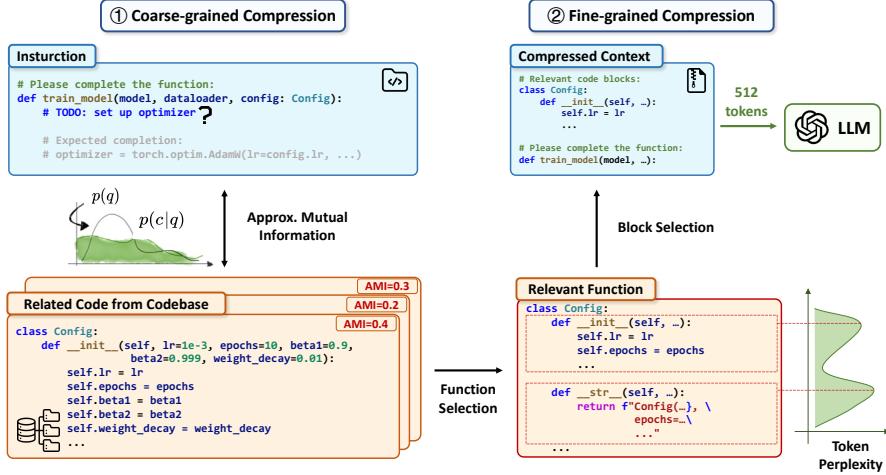


Fig. 2: Overview of the LongCodeZip framework.

Rather than relying solely on embedding similarity between q and c , we propose to select context snippets based on their mutual information, specifically, how much they reduce the perplexity (PPL) of generating q . Specifically, for each candidate context c , we define the approximated mutual information $\text{AMI}(c, q)$ as the reduction in perplexity when c is provided:

$$\text{AMI}(c, q) = \text{PPL}(q) - \text{PPL}(q | c) \quad (1)$$

where $\text{PPL}(q | c)$ is the conditional perplexity of q given c , lower values indicate higher likelihood of q [23]:

$$\text{PPL}(q | c) = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(q_i | q_{<i}, c) \right) \quad (2)$$

Similarly, $\text{PPL}(q)$ denotes the perplexity of q without the context:

$$\text{PPL}(q) = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(q_i | q_{<i}) \right) \quad (3)$$

Here, P denotes the model’s next-token prediction probability, and $q_{<i}$ is the sequence of preceding tokens before q_i . A higher AMI score indicates that c enables the model to better predict q , capturing both surface-level and dependency-based relevance. We therefore compress long contexts by retaining code snippets with the highest mutual information, ensuring that the most essential information for code generation is preserved.

B. Overview

The overview of LongCodeZip is illustrated in Figure 2. Given input of long source code, a task instruction, and a token budget, LongCodeZip follows a *coarse-to-fine* compression pipeline. In the coarse-grained compression stage (Section III-C), the source code is divided into function-level chunks, which are ranked by their relevance to the instruction using conditional perplexity. The top N functions are then selected under a coarse budget, effectively filtering out irrelevant code and avoiding unnecessary computation. In the fine-grained

compression stage (Section III-D), each retained function is further segmented into semantic blocks via perplexity-based chunking. An adaptive retention ratio is assigned to each function according to its estimated importance. Within each function, the most relevant blocks are selected by formulating the problem as a 0/1 knapsack optimization, ensuring that the retained content maximizes relevance while fitting within the allocated token budget.

By combining coarse-grained filtering with fine-grained pruning, LongCodeZip achieves a balance between aggressive compression and semantic preservation, thereby improving both efficiency and task performance.

C. Coarse-Grained Compression: Relevant Function Selection

The coarse-grained compression aims to select high-level code chunks that are most relevant to the task instruction. This process consists of three steps:

Function-Level Chunking. We first split the source code into chunks along function or class boundaries. Functions naturally encapsulate coherent logic and exhibit strong modularity [31]. Chunking at this level ensures that retained code segments are both syntactically valid and semantically self-contained, which is essential for preserving program integrity.

Instruction-aware Relevance Ranking. To measure the relevance of each chunk to the task instruction, we employ an instruction-aware ranking mechanism based on approximated mutual information (1). Chunks are scored and ranked in descending order, allowing us to prioritize those most informative for the given task.

Budget-Constrained Function Selection. Finally, we greedily select the top-ranked chunks under a coarse-grained token budget B_{coarse} , which is the division of the final token budget B by the configurable fine-grained compression ratio R_{fine} . This greedy selection balances efficiency and coverage: a larger budget allows more functions to pass into the fine-grained stage, potentially improving downstream quality but at

Algorithm 1: Pseudo code of Adaptive Fine-Grained Budget Allocation

Input: Large functions $\{f_1, \dots, f_N\}$ with min-max normalized AMI scores $\{\text{AMI}_1, \dots, \text{AMI}_N\}$ and token counts $\{T_1, \dots, T_N\}$; total token budget for large functions B_{large} ; baseline retention ratio R_{base} ; importance parameter β

Output: Function-wise adjusted retention rates $\{R_1, \dots, R_N\}$

```

 $\mathcal{R} \leftarrow \emptyset$  // Initialize retention rate map
for  $f_i \in \{f_1, \dots, f_N\}$  do
     $R_{\text{biased},i} \leftarrow R_{\text{base}} \cdot (1 + \beta \cdot (2 \times \text{AMI}_i - 1))$ ; //
    Compute biased rate
    Clamp  $R_{\text{biased},i}$  to  $[0, 1]$ ;
for  $f_i \in \{f_1, \dots, f_N\}$  do
     $R_i \leftarrow R_{\text{biased},i} \cdot \frac{B_{\text{large}}}{\sum_i R_{\text{biased},j} \cdot T_j}$ ; // Adjust rate
return  $R_1, \dots, R_N$ ;

```

higher computational cost, while a smaller budget accelerates processing at the risk of discarding useful code. Chunks not selected are replaced with placeholders (e.g., comment markers or ellipses), which preserve the global structure while reducing overall context length.

D. Fine-Grained Compression: Intra-Function Pruning

After selecting relevant function-level chunks in the first stage, we apply finer-grained compression to further reduce context length while preserving critical content. This process involves three steps:

Block-Level Chunking. The main challenge in intra-function compression is pruning code without breaking internal logic. To address this, each function is segmented into smaller, semantically coherent blocks. A naive idea is to split code by whitespace lines, but such line-based heuristics often misalign with semantic boundaries. Inspired by techniques in natural language processing [32], we employ a perplexity-based method to identify semantic block boundaries within code. While perplexity-based grouping has shown effectiveness in natural language segmentation, it remains under-explored in code. Consecutive lines in code often form strong semantic associations, making perplexity a useful signal. Within a semantically coherent region, perplexity tends to decrease as context accumulates [32]. We treat each line of code as the smallest atomic unit and group consecutive lines based on their perplexity scores, calculated as in (3). When a line's perplexity exhibits a sharp local increase, exceeding that of its neighbors by at least α times of the standard deviation over all lines, we mark it as a block boundary. Such high-perplexity lines typically mark the beginning of a new block, reflecting underlying semantic or structural changes. This perplexity-guided aggregation allows blocks to capture meaningful code segments while preserving the code structure.

Adaptive Budget Allocation. Functions selected in the coarse-grained stage vary in importance. Hence, applying a

uniform compression ratio across all of them is suboptimal. To address this, we introduce an adaptive budget allocation mechanism that distributes the fine-grained token budget proportionally to function importance. Functions with higher AMI scores receive more token budgets, preserving greater detail, while very small functions $\mathcal{F}_{\text{small}}$ (shorter than five lines) are kept in full. Algorithm 1 summarizes the procedure.

We first define the baseline retention ratio for large functions:

$$R_{\text{base}} = \frac{B - \sum_{j \in \mathcal{F}_{\text{small}}} T_j}{\sum_{k \in \mathcal{F}_{\text{large}}} T_k}, \quad (4)$$

where B is the final token budget, $\mathcal{F}_{\text{small}}$ and $\mathcal{F}_{\text{large}}$ represent the sets of small and large functions respectively, and T_j denotes the number of tokens in function j .

For functions f_1, \dots, f_N selected in the coarse-grained stage, we perform min-max normalization to all AMI scores to $\text{AMI}_{\text{norm},i}$.

For each large function f_i , and its normalized AMI score $\text{AMI}_{\text{norm},i} \in [0, 1]$, a biased retention ratio is then computed as

$$R_{\text{biased},i} = R_{\text{base}} \cdot (1 + \beta \cdot (2 \times \text{AMI}_{\text{norm},i} - 1)), \quad (5)$$

where R_{base} is the baseline retention ratio for large functions (Equation 4). The importance parameter β adjusts sensitivity to importance. When the importance parameter is set to 0, there is no bias, meaning all functions are treated equally. A more positive β increases the emphasis on important functions, allocating more tokens to them. All retention rates are clamped to $[0, 1]$ and globally rescaled so that the total number of retained tokens matches the target token budget for large functions B_{large} :

$$R_i = R_{\text{biased},i} \cdot \frac{B_{\text{large}}}{\sum_j R_{\text{biased},j} \cdot T_j}, \quad (6)$$

where T_j represents the number of tokens in the j -th function. This adjustment preserves the relative importance between functions while ensuring the global constraint is satisfied.

Dynamic Block Selection. For each function, LongCodeZip identifies a subset of blocks to retain, aiming to maximize the total relevance within the constraints of the allocated token budget. This strategy ensures that the compressed context achieves the highest possible information density. We formulate this selection as a classic 0/1 knapsack problem: each block is treated as an item, where the value corresponds to its normalized AMI score and the weight corresponds to its token length. The detailed procedure is outlined in Algorithm 2. We employ a dynamic programming approach to compute the optimal subset of blocks that satisfies the budget constraint while maximizing the cumulative value.

IV. EXPERIMENTAL SETUP

A. Research Questions (RQs)

RQ1: Can LongCodeZip effectively compress code context while preserving the downstream performance?

Algorithm 2: Knapsack Block Selection for Code Compression

Input: Blocks $\{b_1, \dots, b_N\}$ with min-max normalized AMI scores $\{\text{AMI}_1, \dots, \text{AMI}_N\}$ and token counts $\{T_1, \dots, T_N\}$; token budget of this function B_i ; user-defined preserved set \mathcal{P}

Output: Selected blocks $\mathcal{B}_{\text{selected}} \subseteq \{b_1, \dots, b_N\}$

$$B_{\text{remain}} \leftarrow \max(0, B_i - \sum_{j \in \mathcal{P}} T_j);$$

// Compute remaining budget

if $B_{\text{remain}} = 0$ **then**

- └ **return** \mathcal{P} ;

$\mathcal{K} \leftarrow \emptyset$;

for $i = 1$ **to** N **do**

- if** $i \notin \mathcal{P}$ **then**
- └ add (i, T_i, AMI_i) to \mathcal{K} ;

$\mathcal{B}_{\text{selected}} \leftarrow 0/1 \text{ Knapsack DP}(\mathcal{K}, B_{\text{remain}}) \cup \mathcal{P}$;

return $\mathcal{B}_{\text{selected}}$;

RQ2: How does different parts of LongCodeZip contribute to the performance?

RQ3: Does LongCodeZip exhibit cross-model generalization capabilities?

RQ4: What is the efficiency benefit of LongCodeZip in downstream tasks?

B. Datasets

We evaluate our method on long code context benchmarks across three common tasks: code completion, code summarization, and code question answering. These tasks reflect practical developer needs and assess whether compressed code retains sufficient information for downstream performance. For each task, we construct prompts following the benchmark papers [1], [21], [13]. Dataset statistics are shown in Table I.

The Long Code Completion dataset [1] targets the code completion task under long-context of relevant functions. To highlight long-context difficulties, we filtered the test set to 500 Python examples with input contexts longer than 5,000 tokens. The Long Module Summarization dataset [21] contains 216 examples from 43 Python repositories. To focus on the challenging long-context scenario, we also further filtered the original dataset to 139 examples that have more than 2,000 context tokens. RepoQA [13] is a multilingual benchmark that contains 600 long code question answering tests across 60 repositories and 6 programming languages. It requires the model to locate and return a function within the long context using a natural language instruction, similar to a retrieval task.

C. Baselines and Models

We evaluate LongCodeZip against a variety of competitive baselines:

- 1) **No Compression:** The full code context is used without any compression, representing the upper bound of performance.
- 2) **No Context:** The model is evaluated with only the task instruction, without any context, representing the lower bound.

3) **Random Baselines:** Random Token randomly removes individual tokens, while Random Line randomly removes the whole lines of code.

4) **Retrieval-based Methods:** RAG (Sliding Window) uses fixed-size overlapping chunks, whereas RAG (Function Chunking) splits code at function boundaries. Both methods use the state-of-the-art code embedding model UniXCoder-base [30], [29].

5) **Code Compression Methods:** We compare against the compression components from DietCode [26] and SlimCode [27]. DietCode was originally implemented for Python and Java, while SlimCode supports only Java. To enable direct comparison on other benchmarks, we reproduce the SlimCode for Python with tree-sitter².

6) **Text Compression Methods:** We also include several state-of-the-art prompt compression methods for natural languages, including LLMLingua [23], LongLLMLingua [33], and LLMLingua-2 [34].

All methods are evaluated on a diverse set of code LLMs, covering both earlier models like Deepseek-Coder-6.7B [10] and latest models like Qwen2.5-Coder-7B [11] and Seed-Coder-8B [12]. Specifically, we use the instruct version of these models from Huggingface³. To further demonstrate the generalizability of LongCodeZip, we also extend our evaluation to state-of-the-art closed-source models, including GPT-4o⁴ and Claude-3.7-Sonnet⁵.

D. Evaluation Metrics

The evaluation of LongCodeZip encompasses two primary dimensions: compression efficiency and downstream generation performance with compressed context. We report compression *Ratio* on all tasks:

$$\text{Ratio} = \frac{|C_{\text{original}}|}{|C_{\text{compressed}}|} \quad (7)$$

where $|C_{\text{compressed}}|$ and $|C_{\text{original}}|$ denote the number of tokens in the compressed and original contexts respectively.

For the code completion task, We follow LongCoder [1] to evaluate the performance of the models in terms of Exact Match (EM) and Edit Similarity (ES).

For the code summarization task, we follow [21] to use a third party model GPT-4O-MINI⁶ to evaluate which summary better explains the code between the ground truth and the generated one. This LLM-as-Judge evaluation strategy is widely adopted in both NLP and software engineering domains [35], [36], as it has been demonstrated to align well with human preferences and provides more nuanced evaluation compared to traditional metrics [37], [38]. The model chooses the better summary after reviewing both options alongside the code. To

²<https://tree-sitter.github.io/tree-sitter/>

³<https://huggingface.co/models>

⁴<https://openai.com/index/hello-gpt-4o/>

⁵<https://www.anthropic.com/news/clause-3-7-sonnet>

⁶<https://platform.openai.com/docs/models/gpt-4o-mini>

TABLE I: Datasets used for evaluating long-context code compression.

Dataset	# Examples	Avg. Context Len.	Avg. Ground Truth Len.	Languages
Long Code Completion	500	9,328.2	12.4	Python
Long Module Summarization	139	10,809.6	1,758.1	Python
Repo QA	600	11,524.6	156.0	Python, Java, JS, Rust, Go, C++

avoid bias, we prompt it twice with the order reversed. We then compute *CompScore* as:

$$\text{CompScore} = \frac{1}{2}[\mathcal{P}(s_o \succ \hat{s}) + (1 - \mathcal{P}(\hat{s} \succ s_o))] \quad (8)$$

where $\mathcal{P}(s_o \succ \hat{s})$ is the probability that the referee model prefers the generated summary s_o over the reference \hat{s} , and $\mathcal{P}(\hat{s} \succ s_o)$ is the probability for the reverse order. S_{comp} ranges from 0 to 100, with 50 indicating equal preference.

For the code QA task, we follow [13] to evaluate the retrieval accuracy of needle functions, reporting the percentage of models that retrieve a correct match above a BLEU similarity threshold of 0.8 between the generated function f_o and the target function \hat{f} : $\text{BLEU}(\hat{f}, f_o) > 0.8$.

E. Implementation Details

We tailored hyperparameters for each distinct task, consistently mirroring the generation model with our compression model. For **code completion**, which demands focused context, we set the token budget B to 2k, the fine-grained ratio (R_{fine}) to 0.8, and the importance adjustment parameter (β) to 0.5. Conversely, **code summarization** necessitates understanding broader context within large modules; consequently, we increased B to 5k, reduced R_{fine} to 0.3, and maintained β at 0.5. For the **RepoQA task**, where the objective is to precisely replicate entire functions, we set B to 2k and R_{fine} to 1.0 to ensure the structural integrity of functions. These values for B , β , and R_{fine} were determined through experiments on a small held-out set that did not overlap with the test data. All experiments were conducted on a system equipped with an Intel Xeon Gold 6254 CPU and an NVIDIA A100-80G GPU.

V. RESULTS

A. RQ1: Effectiveness on Code Compression

Tables II, III, and IV present the evaluation results of our approach on three downstream tasks, respectively. The best scores among compression methods are bolded. Across all three tasks and multiple backbone models, LongCodeZip consistently outperforms compression baselines by substantial and statistically significant margins ($p < 0.001$ via Wilcoxon signed-rank test on 10 repeated experiments), even when operating at comparable or stricter compression ratios.

Specifically, on the Long Code Completion task, RAG-based methods achieve higher ES and EM scores than other baseline methods, but still fall short of our approach. For instance, with Qwen2.5-Coder-7B, RAG (Function Chunking) achieves an ES score of 52.79 and an EM score of 26.00 at a $3.1\times$ compression ratio. In contrast, our approach achieves 57.55 ES and 32.40 EM at a stricter $4.3\times$ compression ratio, representing a 28% shorter compressed context than the RAG

TABLE II: Results on Long Code Completion

Model	Method	ES	EM	Ratio
DEEPEEK-CODER-6.7B	No Compression	57.14	34.40	1.0x
	No Context	41.29	13.20	-
	Random Token	44.86	13.40	4.4x
	Random Line	50.54	21.20	4.5x
	RAG (Sliding Window)	58.48	31.60	4.2x
	RAG (Function Chunking)	57.93	30.80	5.7x
	LLMLingua	43.61	14.00	5.6x
	LLMLingua-2	46.23	15.00	4.4x
	LongLLMLingua	54.09	26.40	4.8x
	DietCode	51.57	20.20	3.4x
QWEN2.5-CODER-7B	SlimCode	48.84	19.80	4.5x
	LongCodeZip	60.58	35.40	5.3x
	No Compression	56.36	31.80	1.0x
	No Context	38.14	9.60	-
	Random Token	39.10	8.40	4.4x
	Random Line	39.73	12.40	4.5x
	RAG (Sliding Window)	50.81	24.60	2.8x
	RAG (Function Chunking)	52.79	26.00	3.1x
	LLMLingua	21.56	5.40	3.4x
	LLMLingua-2	41.29	12.20	4.4x
SEED-CODER-8B	LongLLMLingua	23.88	9.00	3.2x
	DietCode	43.91	13.20	3.4x
	SlimCode	40.85	12.20	4.5x
	LongCodeZip	57.55	32.40	4.3x
	No Compression	64.04	40.20	1.0x
	No Context	41.88	13.60	-
	Random Token	45.35	13.40	4.4x
	Random Line	50.10	21.20	4.5x
	RAG (Sliding Window)	58.51	32.40	2.8x
	RAG (Function Chunking)	60.52	35.00	3.7x

method. This demonstrates that our method not only preserves more critical information for code completion but also does so with significantly greater compression efficiency.

In contrast to the code completion results, RAG-based methods do not show clear advantages over other baselines on the Long Module Summarization task. However, our approach remains the most competitive, achieving a *CompScore* of 28.01 with Deepseek-Coder-6.7B at a $2.5\times$ compression ratio—surpassing other compression baselines by a considerable margin. This highlights the effectiveness of our method in preserving relevant semantic content for summarization, even with shorter input contexts.

On the RepoQA task, LLMLingua and LLMLingua-2 exhibit poor performance because token-level compression corrupts code syntax and structure, while LongLLMLingua improves this dramatically by performing coarse-grained document-level to fine-grained token-level compression, using instruction-aware contrastive perplexity to preserve code

TABLE III: Results on Long Module Summarization

Model	Method	CompScore	Ratio
DEEPEEK-CODER-6.7B	No Compression	19.09	1.0x
	No Context	2.49	-
	Random Token	11.88	1.8x
	Random Line	17.62	1.8x
	RAG (Sliding Window)	22.95	2.1x
	RAG (Function Chunking)	18.47	2.1x
	LLMLingua	17.65	2.1x
	LongLLMLingua	21.62	1.7x
	LLMLingua-2	18.48	2.1x
	DietCode	17.35	2.1x
QWEN2.5-CODER-7B	SlimCode	20.24	2.2x
	LongCodeZip	28.01	2.5x
	No Compression	56.00	1.0x
	No Context	6.13	-
	Random Token	34.09	1.8x
SEED-CODER-8B	Random Line	46.19	1.8x
	RAG (Sliding Window)	53.50	1.7x
	RAG (Function Chunking)	40.84	2.1x
	LLMLingua	39.81	1.7x
	LongLLMLingua	46.72	1.5x
	LLMLingua-2	52.99	2.1x
	DietCode	35.67	2.1x
	SlimCode	44.13	2.2x
	LongCodeZip	56.47	1.7x
	No Compression	44.95	1.0x
Claude-3.7-Sonnet	No Context	17.42	-
	Random Token	34.16	1.8x
	Random Line	41.27	1.8x
	RAG (Sliding Window)	42.54	3.0x
	RAG (Function Chunking)	43.19	2.1x
	LLMLingua	32.00	3.1x
	LongLLMLingua	49.73	2.4x
	LLMLingua-2	53.88	3.2x
	DietCode	44.74	2.1x
	SlimCode	46.01	2.2x
LongCodeZip	55.07	3.5x	

segments highly relevant to the instruction. Nonetheless, our approach consistently achieves the best performance across all models. Notably, on Deepseek-Coder-6.7B, our approach surpasses LongLLMLingua by 16% in overall score while compressing the context to half the length. This underscores the superior effectiveness of our method in both information retention and aggressive compression for long code understanding.

Notably, LongCodeZip demonstrates strong generalizability across state-of-the-art closed-source models. As comprehensively shown in Table V, on GPT-4o, LongCodeZip achieves an ES score of 64.72 (vs. 65.13 no-compression baseline) on Long Code Completion at a 4.3x compression ratio, closely matching the performance of the uncompressed input while significantly reducing context length. For the RepoQA task, LongCodeZip even surpasses the no-compression baseline, achieving 88.9 average score on GPT-4o, demonstrating that removing irrelevant context can improve performance on complex reasoning tasks. On the more powerful Claude-3.7-Sonnet, LongCodeZip achieves 66.27 ES (vs. 66.24 baseline) with the same compression efficiency. For the RepoQA task, LongCodeZip also surpasses the no-compression baseline on Claude-3.7-Sonnet, achieving 90.7 average score, further demonstrating the effectiveness of our approach.

TABLE IV: Results on RepoQA

Method	Py	C++	Java	TS	Rust	Go	Avg.	Ratio
DEEPEEK-CODER-6.7B								
No Compression	21.0	30.0	44.0	49.0	27.0	59.0	38.3	1.0x
No Context	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-
Random Token	0.0	1.0	2.0	1.0	0.0	6.0	1.7	3.6x
Random Line	3.0	12.0	9.0	7.0	5.0	8.0	7.3	3.5x
RAG (Sliding Window)	49.0	55.0	53.0	67.0	47.0	62.0	55.5	3.5x
RAG (Function Chunking)	42.0	40.0	30.0	36.0	49.0	57.0	42.3	4.0x
LLMLingua	0.0	2.0	6.0	1.0	2.0	4.0	2.5	3.6x
LLMLingua-2	1.0	1.0	4.0	0.0	0.0	3.0	1.5	4.6x
LongLLMLingua	52.0	54.0	65.0	62.0	56.0	67.0	59.3	3.0x
DietCode	13.0	-	28.0	-	-	-	20.5	3.7x
SlimCode	15.0	-	35.0	-	-	-	25.0	4.3x
LongCodeZip	76.0	69.0	80.0	75.0	73.0	79.0	75.3	5.3x
QWEN2.5-CODER-7B								
No Compression	84.0	77.0	89.0	93.0	83.0	90.0	86.0	1.0x
No Context	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-
Random Token	1.0	3.0	4.0	2.0	4.0	7.0	3.5	3.6x
Random Line	6.0	11.0	22.0	10.0	9.0	13.0	11.8	3.5x
RAG (Sliding Window)	64.0	65.0	68.0	72.0	57.0	79.0	67.5	3.7x
RAG (Function Chunking)	54.0	47.0	59.0	39.0	58.0	69.0	54.3	4.3x
LLMLingua	5.0	7.0	9.0	11.0	4.0	16.0	8.7	4.1x
LLMLingua-2	1.0	2.0	8.0	1.0	1.0	4.0	2.8	4.6x
LongLLMLingua	70.0	63.0	71.0	68.0	78.0	78.0	71.3	4.3x
DietCode	17.0	-	35.0	-	-	-	26.0	3.7x
SlimCode	20.0	-	48.0	-	-	-	34.0	4.3x
LongCodeZip	92.0	78.0	87.0	85.0	86.0	95.0	87.2	4.5x
SEED-CODER-8B								
No Compression	73.0	52.0	70.0	81.0	57.0	81.0	69.0	1.0x
No Context	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-
Random Token	2.0	3.0	4.0	1.0	1.0	10.0	3.5	3.6x
Random Line	5.0	6.0	17.0	6.0	4.0	18.0	9.3	3.5x
RAG (Sliding Window)	58.0	51.0	66.0	64.0	57.0	74.0	61.7	3.9x
RAG (Function Chunking)	49.0	40.0	50.0	30.0	47.0	64.0	46.7	4.5x
LLMLingua	4.0	3.0	9.0	8.0	5.0	10.0	6.5	4.3x
LLMLingua-2	1.0	2.0	4.0	1.0	1.0	6.0	2.5	4.6x
LongLLMLingua	71.0	60.0	74.0	65.0	74.0	83.0	71.2	5.1x
DietCode	16.0	-	32.0	-	-	-	24.0	3.7x
SlimCode	25.0	-	50.0	-	-	-	37.5	4.3x
LongCodeZip	83.0	70.0	92.0	74.0	78.0	87.0	80.7	5.3x

We also conduct comprehensive comparisons with recent advanced approaches in code completion, including A³-CodeGen [39], cAST [40], RepoGenix [41], and RLCoder [42] across all evaluated models. As shown in Table VI, LongCodeZip consistently outperforms these advanced RAG methods across the most competitive open-source and closed-source models, SeedCoder and Claude-3.7-Sonnet. Our method can more efficiently retain essential information, achieving higher information density under the same token budget. This demonstrates the broad applicability and consistent effectiveness of our approach across diverse model architectures and capabilities. Notably, these RAG-based retrieval methods are complementary to our compression approach and could potentially be combined with our framework to further enhance performance by first retrieving relevant content and then applying our compression techniques.

Overall, our method achieves effectiveness on par with or better than the No Compression setting, and consistently outperforms all compression baselines across tasks and backbone models even under more aggressive compression.

TABLE V: Results with Closed-source Models

Method	Long Code Completion						Long Module Summarization						RepoQA					
	CLAUDE-3.7-SONNET			GPT-4O			CLAUDE-3.7-SONNET			GPT-4O			CLAUDE-3.7-SONNET			GPT-4O		
	ES	EM	Ratio	ES	EM	Ratio	CompScore	Ratio	CompScore	Ratio	Avg Acc	Ratio	Avg Acc	Ratio	Avg Acc	Ratio	Avg Acc	Ratio
No Compression	66.24	41.20	1.0x	65.13	40.80	1.0x	60.72	1.0x	58.42	1.0x	89.7	1.0x	87.8	1.0x				
No Context	43.97	14.20	-	42.92	14.00	-	6.58	-	6.41	-	0.0	-	0.0	-				
Random Token	47.61	14.00	4.4x	46.51	13.80	4.4x	37.45	1.8x	35.83	1.8x	3.8	3.6x	3.8	3.6x				
Random Line	52.61	22.20	4.5x	51.42	21.80	4.5x	50.12	1.8x	48.24	1.8x	12.2	3.5x	12.1	3.5x				
RAG (Sliding Window)	61.44	34.00	2.8x	60.03	33.20	2.8x	58.03	1.7x	55.85	1.7x	73.8	3.7x	73.0	3.7x				
RAG (Function Chunking)	63.55	36.80	3.1x	62.01	36.00	3.1x	44.56	2.1x	42.76	2.1x	55.0	4.3x	52.5	4.3x				
LLMLingua	46.58	15.20	3.4x	45.53	14.80	3.4x	43.21	1.7x	41.57	1.7x	2.8	4.1x	2.7	4.1x				
LLMLingua-2	49.02	16.20	4.4x	47.90	15.80	4.4x	57.85	2.1x	55.48	2.1x	3.0	4.6x	2.8	4.6x				
LongLLMLingua	57.58	27.80	3.2x	56.24	27.20	3.2x	50.86	1.5x	48.89	1.5x	74.5	4.8x	73.2	4.8x				
DietCode	54.00	19.80	3.4x	52.76	19.40	3.4x	38.82	2.1x	37.21	2.1x	26.7	3.7x	25.5	3.7x				
SlimCode	53.03	20.80	4.5x	51.78	20.40	4.5x	48.11	2.2x	46.13	2.2x	38.3	4.3x	37.0	4.3x				
LongCodeZip	66.27	40.20	4.3x	64.72	38.80	4.3x	61.47	1.7x	59.04	1.7x	88.9	5.1x	88.9	5.1x				

TABLE VI: Comparison with Advanced RAG Methods on Long Code Completion

Model	Method	ES	EM	Ratio
SEED-CODER-8B	No Compression	64.04	40.20	1.0x
	A ³ -CodeGen	58.70	33.10	3.8x
	cAST	57.35	30.90	4.1x
	RepoGenix	60.28	34.70	3.5x
	RLCoder	58.14	32.30	4.0x
CLAUDE-3.7-SONNET	LongCodeZip	63.11	37.40	5.6x
	No Compression	66.24	41.20	1.0x
	A ³ -CodeGen	60.15	35.80	3.8x
	cAST	58.92	33.60	4.1x
	RepoGenix	62.48	37.40	3.5x
	RLCoder	62.76	37.90	4.0x
	LongCodeZip	66.27	40.20	4.3x

TABLE VII: Ablation Study Results

Configuration	ES	EM	Ratio
LongCodeZip	57.55	32.40	4.3x
Coarse-grained Ablations:			
w/ Similarity-based Ranking			
w/ Similarity-based Ranking	49.66 (-7.89)	25.20 (-7.20)	4.3x
w/ Random Ranking	39.76 (-17.79)	11.50 (-20.90)	4.4x
Fine-grained Ablations:			
w/o Fine-grained Compression			
w/o Fine-grained Compression	56.10 (-1.45)	31.20 (-1.20)	4.2x
w/o Adaptive Budget Allocation	55.21 (-2.34)	29.40 (-3.00)	4.3x
w/ Line Chunking	55.98 (-1.57)	31.20 (-1.20)	4.3x
w/ Random Line Selection	55.07 (-2.48)	29.00 (-3.40)	4.3x

Finding 1

LongCodeZip is effective across various downstream tasks, with up to 5.6x compression ratio without sacrificing downstream performance.

B. RQ2: Ablation Study

To understand the contribution of each component in LongCodeZip, we conduct an ablation study on the Long Code Completion task using Qwen2.5-Coder-7B. For all ablations, the total token budget and other hyper-parameters are set the same as the full method. We systematically remove or modify key components to analyze their individual impact. For coarse-grained ablations, we replace our conditional perplexity-based

TABLE VIII: Cross-model Results

Compression Model	DS-6.7B	Seed-8B	Qwen-7B	Avg. ES
No Compression	57.14	64.04	56.36	59.18
No Context	41.29	41.88	38.14	40.44
DEEPEEK-CODER-6.7B	60.58	61.48	56.55	59.54
SEED-CODER-8B	60.86	63.11	55.95	59.97
QWEN2.5-CODER-0.5B	61.12	62.68	56.58	60.13
QWEN2.5-CODER-1.5B	60.89	62.79	56.18	59.95
QWEN2.5-CODER-3B	60.74	63.10	56.79	60.21
QWEN2.5-CODER-7B	61.34	62.62	57.55	60.58

ranking with similarity-based ranking, and compare against random function ranking to establish a lower bound. For fine-grained ablations, we test four variants: removing fine-grained compression entirely (coarse-grained selection only), removing adaptive budget allocation (uniform budget allocation), replacing meta-chunking with simple line-based chunking, and using random line selection within selected functions.

Different components contribute varying degrees to performance. The coarse-grained ranking mechanism is most critical - conditional perplexity-based ranking outperforms similarity-based approaches by 7.89% and random selection by 17.79% in ES score. This demonstrates that semantic relevance through conditional perplexity is superior to lexical similarity. For fine-grained components, adaptive budget allocation improves ES by 2.34%, enabling important functions to retain more detail. Perplexity-based chunking outperforms simple line chunking by 1.57% in ES while being more computationally efficient, as line-by-line compression ranking would incur higher overhead compared to block-based analysis. Knapsack-based selection outperforms random line selection by 2.48% in ES, confirming relevance-guided selection helps compression quality.

Finding 2

Coarse-grained conditional perplexity ranking has the most impact on the performance of LongCodeZip, while fine-grained optimizations further improve the compression information density.

TABLE IX: Efficiency Analysis of Different Methods

Method	Comp. Time (s)	Comp. GPU Mem (GB)	Gen. Time (s)	Gen. GPU Mem (GB)	Ratio	ES	EM
No Compression	0.0	0.0	15.70	Base + 3.48	1.0x	56.36	31.80
No Context	0.0	0.0	0.68	Base + 0.06	-	38.14	9.60
RAG (Function Chunking)	0.53	1.07	7.57	Base + 1.13	3.1x	52.79	26.00
LLMLingua-2	0.65	4.71	6.53	Base + 0.79	4.4x	41.29	12.20
DietCode	15.23	0.0	7.26	Base + 1.03	3.4x	43.91	13.20
SlimCode	0.35	0.0	6.48	Base + 0.78	4.5x	40.85	12.20
LongCodeZip	2.58	<i>Base + 0.69</i>	6.59	<i>Base + 0.81</i>	4.3x	57.55	32.40

Note: Comp.: Compression, Gen.: Generation, Mem: Memory. *Base* model parameters memory: 28.37 GB.

C. RQ3: Transferability

Table VIII presents the cross-model performance (ES) of our approach in the long code completion task. Each row denotes the model used for context compression, while each column specifies the model used for code generation given the compressed context as the input. The results show that our approach generalizes well across different model architectures and sizes, regardless of which compression or generation model is used for downstream tasks. Models released at different times, from DeepSeek-Coder in 2023 [10] to Qwen2.5-Coder in 2024 [11] and Seed-Coder in 2025 [12], achieve similarly strong performance with only minor differences in average ES scores. Notably, even small models (e.g., Qwen2.5-Coder-0.5B) are highly effective, highlighting the strong transferability of our method. Using such small models will significantly reduce compression time and memory overhead, making our approach particularly suitable for resource-constrained scenarios.

Finding 3

Our LongCodeZip generalizes well across different types and sizes of models in the cross-model setting, using a 0.5B model can also bring promising performance.

D. RQ4: Efficiency Analysis

To evaluate the practical efficiency of LongCodeZip, we analyze the Long Code Completion task using Qwen2.5-Coder-7B by measuring both compression overhead and downstream benefits. We select several representative baselines based on their downstream performance in Table IX. The GPU memory costs represent peak memory usage per stage, with generation memory cost referring to additional memory for forward propagation during generation beyond base model parameters (28.37GB). Due to the space limit, we only report the results with several representative baselines. Note that SlimCode and DietCode require no GPU memory for compression because they are not based on neural models.

Table IX demonstrates that LongCodeZip achieves superior compression efficiency while maintaining the best performance. While our method requires a slightly higher compression overhead of 2.58s and additional GPU memory compared to the baselines, it significantly reduces input token costs by 77% and decreases generation latency from 15.70s to 6.59s compared to no compression. This also translates to substantial

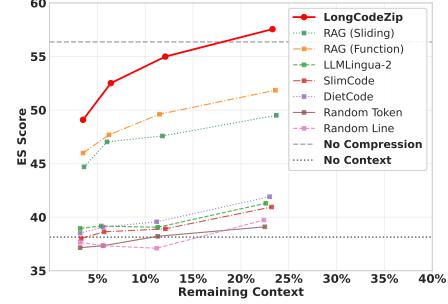


Fig. 3: Performance (ES) vs remaining context (%).

cost savings when using expensive commercial LLM APIs, where pricing is primarily based on input token count. More importantly, as demonstrated in RQ3, the compression overhead can be effectively mitigated by using a lightweight 0.5B model without sacrificing quality. And the efficiency gains can also be further enhanced through techniques like quantization [43], making our approach highly practical for real-world deployment scenarios where cost efficiency is paramount.

Finding 4

Our LongCodeZip achieves 4.3x compression ratio with only 2.6s overhead, reduces generation time from 15.7s to 6.6s, yet it still maintains high downstream performance.

VI. DISCUSSION

A. Compression vs Performance

Understanding the relationship between compression ratio and model performance is essential for evaluating the effectiveness of code compression methods in long-context scenarios. Figure 3 presents the ES score versus the percentage of remaining context, showing Qwen2.5-Code-7B results for representative methods on the Long Code Completion task. LongCodeZip consistently achieves the highest ES scores across all compression ratios, demonstrating its strong ability to identify and retain the most relevant context for code completion. Notably, LongCodeZip can effectively leverage additional context, resulting in substantial performance gains—especially at severe compression ratios (with remaining context less than 10%) where context is extremely limited. This gain becomes less pronounced at more relaxed compression ratios, which is reasonable since our method

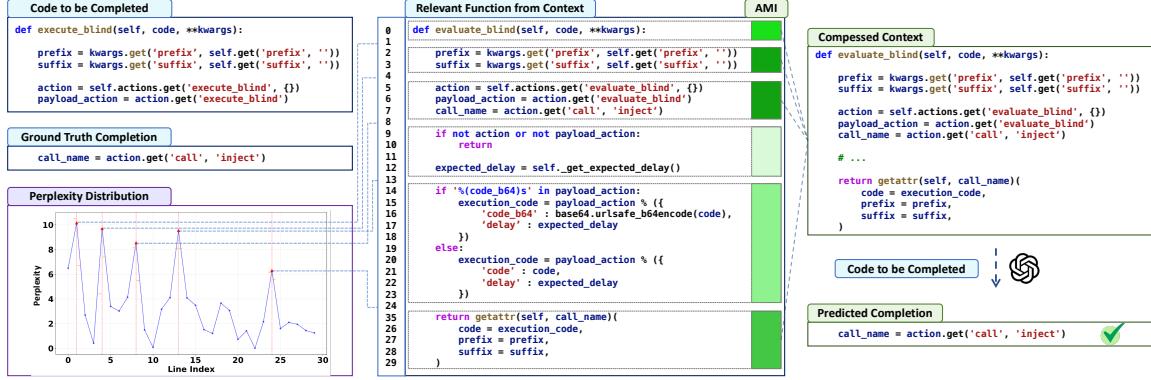


Fig. 4: Example of fine-grained compression process on long code completion.

ranks and selects the most relevant functions early on, so the marginal benefit of extra context diminishes. In contrast, most baselines perform close to random selection, and adding more context does not significantly improve their ES scores. Among the baselines, RAG-based methods do exhibit improvement as more context is retained, but their overall ES scores remain significantly lower than those of LongCodeZip.

B. Case Study

We illustrate the effectiveness of LongCodeZip through a case study in Figure 4, focusing on the fine-grained compression stage (coarse-grained design choice is discussed in Section II). Our method identifies semantic boundaries at positions where a line’s perplexity sharply increases relative to its neighbors. The method tends to separate out major, independent functional modules, and also naturally groups together smaller, closely related segments. Some of our detected boundaries align with double newlines, which is consistent with common programming practices in codes with good code smell. The resulting compressed blocks, highlighted by boxes in the right panel, preserve the key information need for completion. The preserved blocks closely matches the code to be completed. This shows our approach effectively compresses code while retaining relevant and useful content to the task.

In our experiments, we have also observed some common failure modes. In particular, when the context either lacks information relevant to the task instruction or when it is difficult to align an ambiguous instruction with any segment of the context, our method may struggle to identify and preserve useful blocks.

C. Necessity of Two-staged Compression

While the coarse-grained step provides the largest compression gains by removing entire irrelevant functions, the fine-grained compression step is crucial for balancing the trade-off between compression overhead and task model cost. Users can disable the fine-grained step for faster, cheaper compression when using less expensive models. However, for powerful but costly APIs like Claude-3.7-Sonnet, the precise pruning from the fine-grained step becomes critical, yielding substantial cost reductions that justify the additional computational overhead.

This adaptive design allows LongCodeZip to accommodate different deployment scenarios and cost constraints.

VII. THREATS TO VALIDITY

While our evaluation is comprehensive, several threats to validity should be acknowledged. 1) A primary threat concerns the accuracy of our evaluation for the summarization evaluation relies on LLM-generated scores, which may differ from human expert assessments and potentially suffer from ordering effects. To mitigate this, we followed the original paper [21] to average scores over different prompt orderings and employed GPT-4o-mini as an independent referee. These practices reduce bias and improve the objectivity of our results. 2) Our findings may be specific to the datasets, programming languages, or LLMs used. To improve generalizability, we evaluated our approach across diverse datasets, languages, model families, and in cross-model settings. This diversity provides convincing evidence on the generalizability of our findings. 3) There is a risk of data leakage if models are exposed to benchmark data during training. To exclude potential effects of data leakage, we used DeepSeek-Coder-6.7B [10], which was trained only on data available before March 2023, while all the benchmarks we evaluated were released after that date [1], [21], [13]. This step helps ensure the integrity and reliability of our results.

VIII. RELATED WORK

A. Large Language Models for Code

General-purpose LLMs such as GPT [44], Gemini [45], [46], Qwen [47], [48] and DeepSeek [49] demonstrate strong code capabilities through large-scale pretraining on diverse data. To better perform on code-related tasks, a series of code-specialized LLMs have been proposed. CodeX [50] adapts GPT architecture and is pretrained on a large corpus of GitHub code using next-token prediction. Similarly, CodeGen [51] adopts a decoder-only architecture, with a focus on multi-turn program synthesis and open-source availability. StarCoder [52], on the contrary, adopts a fill-in-the-middle objective for improved bidirectional context modeling. CodeLlama [14] extends LLaMA with code-specific tokenization and longer contexts. CodeT5+[53] employs span denoising in

an encoder-decoder framework. More recent models further incorporate instruction tuning and reinforcement learning (RL) to improve alignment and generalization. WizardCoder [54] fine-tunes StarCoder [52] with Evol-Instruct and ChatGPT [55] feedback. DeepSeek-Coder [10] combines instruction tuning, RL and compiler feedback to optimize for correctness and human preference. Qwen2.5-Coder [11] also undergoes instruction tuning and RL, with a focus on long-context fidelity through multi-stage alignment.

These models have demonstrated remarkable performance in downstream tasks such as code generation [51], [56], code summarization [57], [58], and code question answering [59], [60]. Despite increasingly longer context windows, LLMs exhibit significant limitations in long context scenarios, especially when relevant information appears in the middle of a prompt [19], [20] or when code completion requires cross-function or structural dependencies [21], [61]. To address these challenges, a number of long code benchmarks have been introduced, such as LongCodeBench [62], LongCodeU [63], YABLoCo [64], and LongCodeArena [21]. In parallel, recent research has tailored models to code tasks specifically. LongCoder [1] adopts a sliding window mechanism for self-attention to enhance long-context code completion. HiRoPE [65] leverages the hierarchical structure of source code to enable length extrapolation without additional training. aiXcoder-7B-v2 [66] introduces reinforcement learning-based fine-tuning to guide LLMs in utilizing long-range context for repository-level code completion. Complementing these architectural and training advancements, we propose an efficient code context compression technique that preserves essential semantic information while enabling LLMs to operate effectively within constrained input lengths.

B. Context Compression of Large Language Models

Context compression strategies can be categorized into hard prompt methods and soft prompt methods [67], [68], [69], [70]. Soft prompt methods [71], [72], [73] summarize input into dense vectors or prefix embeddings. While memory-efficient, they require fine-tuning on the target model, making them impractical in closed-source settings like using GPT-4. In contrast, hard prompt methods directly manipulate the input by removing or rephrasing less informative content. LLMLingua [23], LongLLMLingua [33], and Selective Context [24] use learned or statistical importance scores to prune uninformative tokens or sentences. LLMLingua-2 [34] advances this by employing data distillation from GPT-4 to train a token classification model, achieving efficient and faithful compression. AttentionRAG [74] prunes the context based on the attention between queries and retrieved contexts.

However, these methods are primarily designed for natural language, and often fail to capture the structural and semantic regularities of source code, leading to suboptimal performance in code-related tasks. This has led to a growing body of research focused on code-specific compression. ShortenDoc [75] targets docstring compression specifically, whereas our method targets source code, which typically dominates the

input in long-context scenarios. DietCode [26] combines static frequency-based filtering with CodeBERT attention heuristics to discard low-impact tokens, but its reliance on model-specific attention reduces adaptability across different architectures. SlimCode[27] applies rule-based token pruning using token types and program dependency graphs, which may not generalize well across languages or tasks. However, these existing methods mainly focus on compressing single functions for short context tasks.

Additionally, advanced RAG-based approaches have been developed for enhancing repository-level code completion by retrieving relevant context, including A³-CodeGen [39], which incorporates third-party library information; cAST [40], which leverages structural chunking via abstract syntax trees; RepoGenix [41], which combines analogous and relevant contexts; and RLCoder [42], which trains stronger retrievers for improved context selection. Unlike these approaches that are specifically designed for repository-level code completion, we propose a training-free code context compression technique that provides broader applicability across diverse long-context code tasks including summarization and question answering by preserving essential semantic information while enabling existing LLMs to operate effectively within constrained input lengths. Our contribution lies in the synergistic integration and significant code-aware approach to address the unique structural and semantic characteristics of programming languages.

To the best of our knowledge, our approach is the first to explicitly target long-context compression in code LLMs, providing a training-free and model-agnostic solution that efficiently preserves task-relevant content under tight token budgets.

IX. CONCLUSION

In this paper, we have introduced LongCodeZip, a training-free, model-agnostic and plug-and-play framework for long-context code compression. Our two-stage hierarchical approach combines function-level selection with block-level pruning. Comprehensive experiments across code completion, summarization, and question answering tasks demonstrate that LongCodeZip achieves up to a 5.6x compression ratio without sacrificing task performance, consistently outperforms existing baselines, and significantly reduces computational costs. The framework exhibits strong cross-model generalization and maintains competitive performance even with a lightweight 0.5B compression model. As the first framework specifically designed for long-context code compression, LongCodeZip enables code LLMs to scale more efficiently to real-world, large-scale software development scenarios.

ACKNOWLEDGMENT

We thank anonymous reviewers for their constructive comments. We also thank Maoquan Wang and Yanfu Yan for helpful discussions on improving this work. This research is funded by the National Key Research and Development Program of China (Grant No. 2023YFB4503802) and the Natural Science Foundation of Shanghai (Grant No. 25ZR1401175).

REFERENCES

- [1] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, “Longcoder: A long-range pre-trained language model for code completion,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.
- [2] T. Liu, C. Xu, and J. McAuley, “Repobench: Benchmarking repository-level code auto-completion systems,” *arXiv preprint arXiv:2306.03091*, 2023.
- [3] Y. Wang, Y. Wang, S. Wang, D. Guo, J. Chen, J. Grundy, X. Liu, Y. Ma, M. Mao, H. Zhang *et al.*, “Repotransbench: A real-world benchmark for repository-level code translation,” *arXiv preprint arXiv:2412.17744*, 2024.
- [4] C. Wang, T. Yu, J. Wang, D. Chen, W. Zhang, Y. Shi, X. Gu, and B. Shen, “Evoc2rust: A skeleton-guided framework for project-level c-to-rust translation,” *arXiv preprint arXiv:2508.04295*, 2025.
- [5] W. Zeng, Y. Wang, C. Hu, Y. Shi, C. Wan, H. Zhang, and X. Gu, “Pruning the unsurprising: Efficient code reasoning via first-token surprisal,” *arXiv preprint arXiv:2508.05988*, 2025.
- [6] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, “Algo: Synthesizing algorithmic programs with generated oracle verifiers,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 54 769–54 784, 2023.
- [7] Y. Shi, S. Wang, C. Wan, and X. Gu, “From code to correctness: Closing the last mile of code generation with hierarchical debugging,” *arXiv preprint arXiv:2410.01215*, 2024.
- [8] S. Chen, S. Lin, X. Gu, Y. Shi, H. Lian, L. Yun, D. Chen, W. Sun, L. Cao, and Q. Wang, “Swe-exp: Experience-driven software issue resolution,” *arXiv preprint arXiv:2507.23361*, 2025.
- [9] H. Li, Y. Shi, S. Lin, X. Gu, H. Lian, X. Wang, Y. Jia, T. Huang, and Q. Wang, “Swe-debate: Competitive multi-agent debate for software issue resolution,” *arXiv preprint arXiv:2507.23348*, 2025.
- [10] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, “Deepseek-coder: When the large language model meets programming—the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [11] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, “Qwen2. 5-coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.
- [12] Y. Zhang, J. Su, Y. Sun, C. Xi, X. Xiao, S. Zheng, A. Zhang, K. Liu, D. Zan, T. Sun *et al.*, “Seed-coder: Let the code model curate data for itself,” *arXiv preprint arXiv:2506.03524*, 2025.
- [13] J. Liu, J. Le Tian, V. Daita, Y. Wei, Y. Ding, Y. K. Wang, J. Yang, and L. Zhang, “Repoqa: Evaluating long context code understanding,” in *First Workshop on Long Context Foundation Models ICML*, 2024.
- [14] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [15] J. Zhang, Y. Fan, W. Lin, R. Chen, H. Jiang, W. Chai, J. Wang, and K. Wang, “Gam-agent: Game-theoretic and uncertainty-aware collaboration for complex visual reasoning,” *arXiv preprint arXiv:2505.23399*, 2025.
- [16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [17] J. Zhang, Y. Fan, K. Cai, Z. Huang, X. Sun, J. Wang, C. Tang, and K. Wang, “Drdiff: Dynamic routing diffusion with hierarchical attention for breaking the efficiency-quality trade-off,” 2025. [Online]. Available: <https://arxiv.org/abs/2509.02785>
- [18] J. Zhang, Z. Huang, Y. Fan, N. Liu, M. Li, Z. Yang, J. Yao, J. Wang, and K. Wang, “KABB: Knowledge-aware bayesian bandits for dynamic expert coordination in multi-agent systems,” in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: <https://openreview.net/forum?id=AKV9a4jho>
- [19] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *arXiv preprint arXiv:2307.03172*, 2023.
- [20] J. Li, M. Wang, Z. Zheng, and M. Zhang, “Loogle: Can long-context language models understand long contexts?” *arXiv preprint arXiv:2311.04939*, 2023.
- [21] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi *et al.*, “Long code arena: a set of benchmarks for long-context code models,” *arXiv preprint arXiv:2406.11612*, 2024.
- [22] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.
- [23] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, “LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 13 358–13 376.
- [24] Y. Li, B. Dong, F. Guerin, and C. Lin, “Compressing Context to Enhance Inference Efficiency of Large Language Models,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 6342–6353.
- [25] X. Cheng, X. Wang, X. Zhang, T. Ge, S.-Q. Chen, F. Wei, H. Zhang, and D. Zhao, “xRAG: Extreme Context Compression for Retrieval-augmented Generation with One Token,” May 2024.
- [26] Z. Zhang, H. Zhang, B. Shen, and X. Gu, “Diet code is healthy: Simplifying programs for pre-trained models of code,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1073–1084.
- [27] Y. Wang, X. Li, T. N. Nguyen, S. Wang, C. Ni, and L. Ding, “Natural is the best: Model-agnostic code simplification for pre-trained large language models,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 586–608, 2024.
- [28] Z. Li, C. Li, M. Zhang, Q. Mei, and M. Bendersky, “Retrieval augmented generation or long-context llms? a comprehensive study and hybrid approach,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024, pp. 881–893.
- [29] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, “Repocoder: Repository-level code completion through iterative retrieval and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 2471–2484.
- [30] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” *arXiv preprint arXiv:2203.03850*, 2022.
- [31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [32] J. Zhao, Z. Ji, Y. Feng, P. Qi, S. Niu, B. Tang, F. Xiong, and Z. Li, “Metachunking: Learning efficient text segmentation via logical perception,” *arXiv preprint arXiv:2410.12788*, 2024.
- [33] H. Jiang, Q. Wu, X. Luo, D. Li, C.-Y. Lin, Y. Yang, and L. Qiu, “LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1658–1677.
- [34] Z. Pan, Q. Wu, H. Jiang, M. Xia, X. Luo, J. Zhang, Q. Lin, V. Rühle, Y. Yang, C.-Y. Lin *et al.*, “Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression,” in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 963–981.
- [35] Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, and C. Zhu, “G-eval: NLG evaluation using gpt-4 with better human alignment,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 2511–2522.
- [36] H. Song, H. Su, I. Shalyminov, J. Cai, and S. Mansour, “Finesure: Fine-grained summarization evaluation using llms,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 906–922.
- [37] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia, “Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1955–1977, 2025.
- [38] J. He, J. Shi, T. Y. Zhuo, C. Treude, J. Sun, Z. Xing, X. Du, and D. Lo, “From code to courtroom: Llms as the new software judges,” *arXiv preprint arXiv:2503.02246*, 2025.

- [39] D. Liao, S. Pan, X. Sun, X. Ren, Q. Huang, Z. Xing, H. Jin, and Q. Li, “A 3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware,” *IEEE Transactions on Software Engineering*, 2024.
- [40] Y. Zhang, X. Zhao, Z. Wang, C. Yang, J. Wei, and T. Wu, “cCAST: Enhancing Code Retrieval-Augmented Generation with Structural Chunking via Abstract Syntax Tree,” Jun. 2025.
- [41] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, W. Jiang, H. Chen, C. Wang, and G. Fan, “RepoGenix: Dual Context-Aided Repository-Level Code Completion with Language Models,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE ’24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2466–2467.
- [42] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, “Rlcoder: Reinforcement learning for repository-level code completion,” *arXiv preprint arXiv:2407.19487*, 2024.
- [43] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” in *The Eleventh International Conference on Learning Representations*. OpenReview, 2023.
- [44] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Alten Schmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [45] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [46] G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang *et al.*, “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context,” *arXiv preprint arXiv:2403.05530*, 2024.
- [47] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, “Qwen technical report,” *arXiv preprint arXiv:2309.16609*, 2023.
- [48] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei *et al.*, “Qwen2. 5 technical report,” *arXiv preprint arXiv:2412.15115*, 2024.
- [49] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025.
- [50] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [51] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.
- [52] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chiu *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [53] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint arXiv:2305.07922*, 2023.
- [54] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, “Wizardcoder: Empowering code large language models with evol-instruct,” *arXiv preprint arXiv:2306.08568*, 2023.
- [55] OpenAI, “ChatGPT,” <https://openai.com/blog/chatgpt>, 2022, accessed: 2025-05-11.
- [56] Y. Shi, H. Zhang, C. Wan, and X. Gu, “Between lines of code: Unraveling the distinct patterns of machine and human programmers,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 51–62.
- [57] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” *arXiv preprint arXiv:2005.00653*, 2020.
- [58] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, Y. Wang, D. Guo, S. Han, H. Zhang, and D. Zhang, “Context-aware code summarization with multi-relational graph neural network,” *Automated Software Engineering*, vol. 32, no. 1, pp. 1–26, 2025.
- [59] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 933–944.
- [60] W. Peng, Y. Shi, Y. Wang, X. Zhang, B. Shen, and X. Gu, “Swe-qa: Can language models answer repository-level code questions?” *arXiv preprint arXiv:2509.14635*, 2025.
- [61] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, “Codereval: A benchmark of pragmatic code generation with generative pre-trained models,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [62] S. Rando, L. Romani, A. Sampieri, Y. Kyuragi, L. Franco, F. Galasso, T. Hashimoto, and J. Yang, “Longcodebench: Evaluating coding llms at 1m context windows,” *arXiv preprint arXiv:2505.07897*, 2025.
- [63] J. Li, X. Guo, L. Li, K. Zhang, G. Li, Z. Tao, F. Liu, C. Tao, Y. Zhu, and Z. Jin, “Longcode: Benchmarking long-context language models on long code understanding,” *arXiv preprint arXiv:2503.04359*, 2025.
- [64] A. Valeev, R. Garaev, V. Lomshakov, I. Piontkovskaya, V. Ivanov, and I. Adewuyi, “Yabloco: Yet another benchmark for long context code generation,” *arXiv preprint arXiv:2505.04406*, 2025.
- [65] K. Zhang, G. Li, H. Zhang, and Z. Jin, “Hirope: Length extrapolation for code models using hierarchical position,” *arXiv preprint arXiv:2403.19115*, 2024.
- [66] J. Li, H. Zhu, H. Liu, X. Shi, H. Zong, Y. Dong, K. Zhang, S. Jiang, Z. Jin, and G. Li, “aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion,” *arXiv preprint arXiv:2503.15301*, 2025.
- [67] Z. Li, Y. Liu, Y. Su, and N. Collier, “Prompt compression for large language models: A survey,” *arXiv preprint arXiv:2410.12388*, 2024.
- [68] X. Liu, Y. Wang, J. Ma, and L. Zhang, “Video compression commander: Plug-and-play inference acceleration for video large language models,” *arXiv preprint arXiv:2505.14454*, 2025.
- [69] X. Liu, Z. Wang, Y. Han, Y. Wang, J. Yuan, J. Song, B. Zheng, L. Zhang, S. Huang, and H. Chen, “Global compression commander: Plug-and-play inference acceleration for high-resolution large vision-language models,” *arXiv preprint arXiv:2501.05179*, 2025.
- [70] X. Liu, Z. Wen, S. Wang, J. Chen, Z. Tao, Y. Wang, X. Jin, C. Zou, Y. Wang, C. Liao *et al.*, “Shifting ai efficiency from model-centric to data-centric compression,” *arXiv preprint arXiv:2505.19147*, 2025.
- [71] J. Mu, X. Li, and N. Goodman, “Learning to compress prompts with gist tokens,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 19 327–19 352, 2023.
- [72] Z. Li, Y. Su, and N. Collier, “500xCompressor: Generalized Prompt Compression for Large Language Models,” Aug. 2024.
- [73] C. Wang, Y. Yang, R. Li, D. Sun, R. Cai, Y. Zhang, and C. Fu, “Adapting llms for efficient context processing through soft prompt compression,” in *Proceedings of the International Conference on Modeling, Natural Language Processing and Machine Learning*, 2024, pp. 91–97.
- [74] Y. Fang, T. Sun, Y. Shi, and X. Gu, “Attentionrag: Attention-guided context pruning in retrieval-augmented generation,” *arXiv preprint arXiv:2503.10720*, 2025.
- [75] G. Yang, Y. Zhou, W. Cheng, X. Zhang, X. Chen, T. Y. Zhuo, K. Liu, X. Zhou, D. Lo, and T. Chen, “Less is More: DocString Compression in Code Generation,” Oct. 2024.