# SWE-Pruner: Self-Adaptive Context Pruning for Coding Agents

Anonymous ACL submission

## Abstract

LLM agents have demonstrated remarkable capabilities in software development, but their performance is hampered by long interaction contexts, which incur high API costs and latency. While various context compression approaches such as LongLLMLingua have emerged to tackle this challenge, they typically rely on fixed metrics such as PPL, ignoring the task-specific nature of code understanding. As a result, they frequently disrupt syntactic and logical structure and fail to retain critical implementation details. In this paper, we propose SWE-Pruner, a self-adaptive context pruning framework tailored for coding agents. Drawing inspiration from how human programmers "selectively skim" source code during development and debugging, SWE-Pruner performs task-aware adaptive pruning for long contexts. Given the current task, the agent formulates an explicit goal (e.g., "focus on error handling") as a hint to guide the pruning targets. A lightweight *neural skimmer* (0.6B parameters) is trained to dynamically select relevant lines from the surrounding context given the goal. Evaluations across four benchmarks and multiple models validate SWE-Pruner's effectiveness in various scenarios, achieving 23-54% token reduction on agent tasks like SWE-Bench Verified and up to 14.84× compression on single-turn tasks like LongCodeQA with minimal performance impact[1].

## 1 Introduction

Large Language Models (LLMs) have rapidly progressed in software engineering tasks, from simple code understanding (Chen et al., 2021; Li et al., 2023b; Yang et al., 2025; Hu et al., 2025) to interactive agents capable of navigating repositories, running tests, and submitting patches end-to-end (Chen et al., 2025; Li et al., 2025; Xiao et al., 2025). Recent coding agents like Claude Code (Anthropic, 2025) and Gemini CLI (Google, 2025) have augmented LLMs with sophisticated toolchains including terminals, editors, and file search, enabling multi-step reasoning over complex codebases.

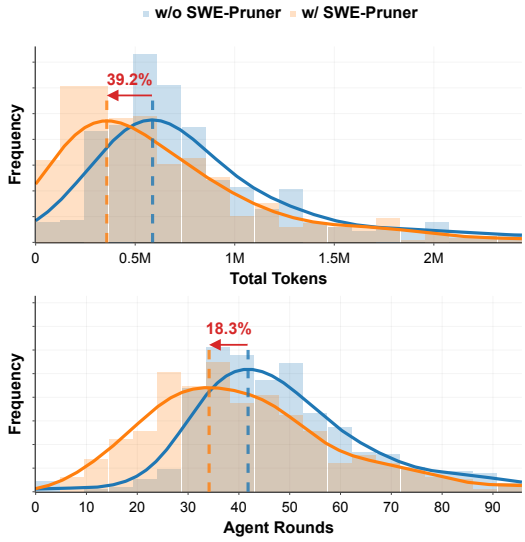



Figure 1: **SWE-Pruner effectiveness on SWE-Bench Verified with Claude Sonnet 4.5.** Our SWE-Pruner (orange) saves 39.2% in total tokens and 18.3% in agent rounds for the original Mini SWE Agent (blue).

Despite this progress, a critical bottleneck remains: the accumulation of context length within the constrained window of LLMs. Navigating real-world software repositories confronts agents with a massive "Context Wall." Although long-context models have emerged (Achiam et al., 2023; Team et al., 2024), blindly ingesting large volumes of code incurs prohibitive inference costs (Jiang et al., 2023) and introduces severe noise, leading to attention dilution and hallucinations (Liu et al., 2023a; Li et al., 2023a; Wang et al., 2025b).

Although context compression techniques offer a potential remedy, existing approaches face critical limitations when applied to coding agents.

[1]Code and model are available at https://anonymous.4open.science/r/swe-pruner-7503.

Prior work on context compression primarily targets natural language (Jiang et al., 2023; Li et al., 2023c; Jiang et al., 2024; Mu et al., 2023), leading to severe trade-offs when applied to code (Shi et al., 2025): token pruning methods compromise syntactic validity, while abstractive techniques discard character-level information critical for debugging (Jiang et al., 2023; Li et al., 2023c). Beyond structural concerns, these methods are fundamentally misaligned with coding agent requirements-they operate with static compression ratios and task-agnostic criteria (e.g., perplexity), unable to dynamically prioritize context based on the evolving goals of multi-turn agent interactions (Zhang et al., 2024; Yang et al., 2024a).

To address these limitations, we introduce SWE-Pruner, a self-adaptive pruning framework tailored for coding agents. Our key insight stems from how both programmers and agents navigate large codebases: rather than reading code line by line, they employ *goal-driven selective attention* that programmers quickly scan to locate relevant sections based on their goals (e.g., "find error handling logic"), while agents use tools like `grep` or file search to coarsely retrieve targeted code regions including excessive surrounding context. SWE-Pruner refines this process by enabling agents to articulate explicit natural language goals alongside retrieval actions. To capture the dynamic, task-specific objectives of agent workflows, we train a lightweight 0.6B pruning model on 61K synthetic data, enabling adaptive selection conditioned on given goals. By operating at line-level granularity, our approach preserves syntactic and structural integrity while achieving precise compression that distills coarse retrieval results into focused, task-relevant context.

We evaluate SWE-Pruner across multi-turn agent tasks (SWE-Bench Verified (Jimenez et al., 2024) and SWE-QA (Peng et al., 2025)) and single-turn long-context understanding tasks (Long Code Completion (Guo et al., 2023) and Long Code QA (Rando et al., 2025)). Across models and benchmarks, SWE-Pruner consistently delivers substantial efficiency gains with minimal performance degradation. Beyond token savings (e.g., 39% reduction on SWE-Bench Verified with Claude Sonnet 4.5, as illustrated in Figure 1), the focused context also improves agent decision quality, reducing interaction rounds by up to 26% through more decisive reasoning and fewer redundant exploratory actions.

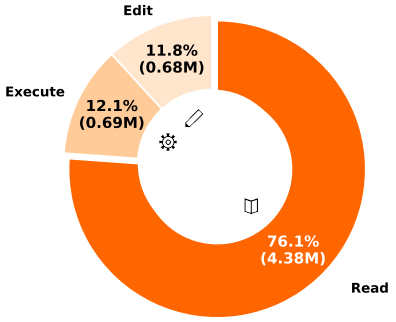



Figure 2: Token cost distribution over different tool calls for Mini-SWE-Agent on SWE-Bench Verified. Read operations dominate token consumption at 76.1%, motivating the need for context pruning mechanisms.

In summary, we make the following contributions:

- We propose a novel pruning framework for coding agents that performs task-aware, line-level pruning to alleviate the context wall problem.
- We design a novel architecture with a lightweight backbone of only 0.6B parameters for efficient inference with adaptive, task-dependent pruning.
- Evaluations across various benchmarks demonstrate 23–54% token reduction on agent tasks and up to 14.8× compression on single-turn tasks with minimal performance impact.

## 2 Motivation

In practice, coding agents spend an excessive amount of their token budget on repeatedly exploring the codebase (Majgaonkar et al., 2025; Xiao et al., 2025). To quantify this, we perform a preliminary analysis of the trajectories by a Mini SWE Agent (Yang et al., 2024b). The agent is instantiated with Claude Sonnet 4.5 on SWE-Bench Verified. We extract action types from the trajectories and categorize them into three types: 1) *read*, which denotes file and directory inspection operations such as `cat`, `grep`, and `head`; 2) *execute*, which means running programs or scripts for testing, and 3) *edit*, representing in-place modifications. We count the total tokens consumed in each agent round. As shown in Figure 2, *read*-type operations consume an overwhelming 76.1% of total tokens, significantly exceeding *execute* (12.1%) and *edit* (11.8%) operations combined. This issue

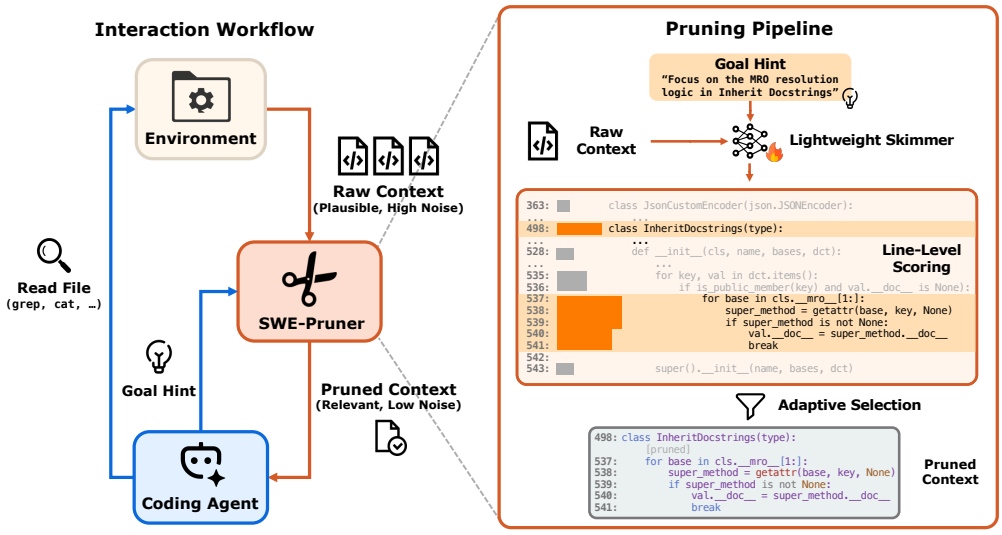



Figure 3: Overview of SWE-Pruner. Left: The **Interaction Workflow** demonstrates how SWE-Pruner functions as a middleware between the Coding Agent and the Environment. It intercepts the Raw Context from file operations and delivers a Pruned Context to the agent. Right: The **Pruning Pipeline** details the internal mechanism. Based on a specific goal hint from the coding agent, the *neural skimmer* processes the raw context through line-level scoring and adaptive selection to deliver the pruned context.

is further exacerbated in multi-round agent interactions, where code retrieved in earlier rounds persists in the context and accumulates as the interaction progresses. Similar patterns are observed with other backbone models like GLM-4.6 (Appendix A).

This empirical pattern reveals a critical opportunity for context optimization. When navigating unfamiliar codebases, agents must extensively explore through coarse-grained file operations that stream entire files or blocks into the context. While this exploratory strategy is necessary for understanding the codebase structure, it causes substantial token costs and introduces redundant content that accumulates across rounds. Addressing this challenge requires a context pruning mechanism that is context-aware (capable of identifying relevant information based on the agent's current goal) and lightweight enough to avoid introducing additional latency. These observations strongly motivate SWE-Pruner, an lightweight pruning framework that enables more efficient token allocation throughout the problem-solving trajectory.

# 3 Approach

## 3.1 Overview

We introduce SWE-Pruner, a framework that prunes long code contexts for agents through task-aware, adaptive filtering. As illustrated in Figure 3, SWE-Pruner operates as middleware between coding agents and their environment. When an agent issues file-reading commands (e.g., `grep`, `cat`), the raw retrieved context—often thousands of lines—is captured and filtered before reaching the agent. To guide this pruning, the agent generates a *Goal Hint* describing its current information need (e.g., "Focus on MRO resolution logic"). A lightweight neural skimmer is trained to adaptively selects relevant lines from the raw context, while preserving code structure. Only this pruned context (relevant and low-noise) is returned to the agent.

The framework comprises three core components as introduced in the following sections, respectively. First, the target agent is instructed to provide a goal hint indicating the current information need (Section 3.2). Next, a lightweight neural skimmer is designed to score context tokens and aggregate relevant lines (Section 3.3). Finally, we integrate the skimmer into real agentic systems with minimal modifications (Section 3.4).

## 3.2 Goal Hint Generation

Central to our framework is the *Goal Hint*—a natural language description of the agent's current information need. Rather than relying on keyword-based filtering, we instruct the agent to generate goal hints as complete, self-contained questions (e.g., *"How is authentication handled?"*) that

capture the semantic intent of its current reasoning step (detailed prompt in Appendix J). To enable agents to communicate these goal hints to our pruning system, we augment standard file manipulation tools (e.g., `cat`, `grep`) with an optional `context_focus_question` parameter. When provided, this parameter passes the goal hint to the skimmer, which then filters large outputs for query-relevant content. When omitted, the skimmer is bypassed and full outputs are returned, preserving backward compatibility. This lightweight wrapper design (illustrated below) requires minimal modifications to existing agent infrastructures, enabling seamless integration without disrupting established workflows.

```
# Original tool (unchanged)
def grep(file_path, pattern):
    # ... original grep implementation ...
    return matches

# New tool with pruner
def grep_with_pruner(file_path, pattern,
      context_focus_question=None):
    # Call original tool
    raw_output = grep(file_path, pattern)
    if context_focus_question:
        # Prune if hint provided
        return prune(raw_output, context_focus_question)
    return raw_output  # Bypass pruner otherwise
```

### 3.3 Lightweight Neural Skimmer

**Model Architecture.** We formulate context pruning as a reranking problem: Given the agent context $C = \{x_1, x_2, ..., x_n\}$ where each $x_i$ represents a token, and a query $q$ representing the agent's current goal, the model computes a relevance score $s_i$ for each token through a neural scoring function:

$$s_i = \mathcal{F}(q, x_i | C; \theta) \tag{1}$$

where $\mathcal{F}$ represents the scoring function parameterized by $\theta$ that evaluates token $x_i$ in the context of both the query $q$ and the full code context $C$.

To enable line-level pruning decisions, we aggregate token scores to the line level. Let $L = \{l_1, ..., l_m\}$ denote the set of lines obtained by splitting $C$, and let $T_j$ denote the set of tokens in line $l_j$. The line-level relevance score $\bar{s}_j$ is computed as the average of its constituent token scores:

$$\bar{s}_j = \frac{1}{|T_j|} \sum_{t \in T_j} s_t \tag{2}$$

This averaging operation ensures that lines are evaluated based on their overall relevance rather than being dominated by a few high-scoring tokens, maintaining the semantic coherence necessary for code comprehension.

We adopt the Qwen3-Reranker-0.6B (Zhang et al., 2025) as our backbone due to its efficiency and pre-trained knowledge of code structures. During inference, a line $l_j$ is retained if its aggregated score $\bar{s}_j$ exceeds a predefined threshold $\tau$. The model processes retrieved chunks in parallel to minimize latency. Given its lightweight architecture of 0.6B parameters, pruning overhead is negligible compared to the token savings for downstream agent LLMs. See Algorithm 1 for detailed inference steps.

**Training Objective.** We train the pruning model by minimizing the conditional random field negative log likelihood (CRF-NLL) (Zheng et al., 2015). Unlike mere binary cross entropy, CRF explicitly models transition probabilities between retain/prune states, enabling the model to learn line-level retention decisions while capturing sequential dependencies.

Given the context representation $\mathbf{x} = \mathbf{x}_1, \ldots, \mathbf{x}_n$ and silver labels $\mathbf{y}$, the pruning head minimizes the following loss function:

$$\mathcal{L}_{\text{compress}} = \frac{1}{B} \sum_{i=1}^{B} \frac{\mathcal{L}_{\text{CRF-NLL}}(\mathbf{x}_i, \mathbf{y}_i)}{L_i} \tag{3}$$

where $\mathbf{x}_i$ denotes the learned feature representation for each token; $L_i$ is the sequence-length normalization, which prevents bias toward aggressive pruning in long contexts. We keep the original reranking head in Qwen Reranker (Bai et al., 2023) to preserve document-level relevance scoring capability. The reranking head minimizes $\mathcal{L}_{\text{rerank}} = \text{MSE}(s_{\text{pred}}, s_{\text{ref}})$ between predicted and reference document-level relevance scores in $[0, 1]$. The final objective combines both heads with a balancing weight $\lambda$:

$$\mathcal{L}_{\text{total}} = (1 - \lambda) \cdot \mathcal{L}_{\text{compress}} + \lambda \cdot \mathcal{L}_{\text{rerank}} \tag{4}$$

More details about the architecture, CRF formulations, and training configurations are provided in Appendix B.

**Constructing Training Data.** Training the neural skimmer requires data with line-level supervision that preserves both relevance and code structure. Since such dataset is unavailable, we construct a polyglot training corpus following a teacher-student paradigm. We sample code snippets from GitHub[2], a curated collection of high-quality repositories, and utilize a teacher LLM

[2] https://huggingface.co/datasets/nick007x/github-code-2025

(Qwen3-Coder-30B-A3B-Instruct (Team, 2025)) to synthesize task-oriented queries that target specific functional subsets of each snippet. This process produces training quadruplets $(q, C, M, S)$, where $M$ is a binary line-level mask indicating which lines to retain and $S$ is a document-level relevance score.

To ensure generalization across diverse coding scenarios, we design queries based on a taxonomy of nine distinct agentic tasks, such as *Code Debugging*, *Feature Addition*, and *Code Refactoring*, covering common information needs in real-world agentic workflows. We employ an LLM-as-a-Judge filtering mechanism (Wang et al., 2025a; He et al., 2025) with Qwen3-Next-80B-A3B-Thinking (Team, 2025) to ensure annotation quality. This rigorous filtering yields a final training corpus of 61,184 high-quality samples with verified line-level annotations. Complete details on task taxonomy, data statistics, and generation configurations are provided in Appendix C.

### 3.4 Integration with Agentic Workflows

The trained skimmer is deployed in real coding agents to accomplish specific tasks. Our framework flexibly adapts to different task scenarios. For multi-turn agent tasks (e.g., SWE-Bench), agents dynamically generate Goal Hints at each round based on their evolving reasoning trace, enabling them to shift focus from high-level navigation to detailed debugging as needed. For single-turn tasks with inherent queries (e.g., code question answering), the task description serves as the initial Goal Hint, though agents can refine it in subsequent retrieval rounds. This flexibility allows agents to seamlessly transition between broad exploration (no pruning) and focused investigation (with pruning) as their information needs evolve.

## 4 Experiments

### 4.1 Benchmarks and Agents

We evaluate SWE-Pruner on four benchmarks spanning single-turn and multi-turn scenarios. For single-turn tasks, we use **Long Code Completion** (Guo et al., 2023) (500 Python examples with 5K+ token contexts) and **Long Code QA** (Rando et al., 2025) (question answering on long code contexts up to 1M tokens), evaluating under 4x and 8x compression constraints. For multi-turn agent tasks, we use **SWE-Bench Verified** (Jimenez et al., 2024) (500 real-world GitHub issues requiring patch generation) and **SWE-QA** (Peng et al., 2025) (repository-specific question answering across three repositories). Specifically, we integrate SWE-Pruner into Mini SWE Agent (Yang et al., 2024b) for SWE-Bench Verified and OpenHands (Team, 2024) for SWE-QA, evaluating with Claude Sonnet 4.5 and GLM-4.6 backbone models. Detailed benchmark descriptions, agent configurations, and experimental settings are provided in Appendix D.

### 4.2 Baselines

We compare SWE-Pruner against representative methods for compressing code context and environment observations. Full Context and No Context establish upper and lower performance bounds. For compression baselines, we evaluate LLMLingua-2 (Pan et al., 2024) and Selective-Context (Li et al., 2023c) which perform token-level pruning based on perplexity and self-information respectively, RAG which retrieves code chunks via embedding similarity using UniXCoder (Guo et al., 2022), and LongCodeZip (Shi et al., 2025) which leverages program structure for compression. For multi-turn agent tasks, we additionally compare with LLM Summarize that generates abstractive summaries using the backbone model. All baselines are configured to match 4x and 8x compression constraints under identical experimental settings. We did not compare with agent history compression methods (Ye et al., 2025; Kang et al., 2025). While these methods effectively manage long-horizon agent interactions, they tackle a different problem to ours. They aim at compressing agents' prior interaction trajectories, while our method compresses agents' observations like repository content, making them incomparable to our approach.

### 4.3 Metrics

We evaluate both task performance and compression efficiency. Task performance is measured through Edit Similarity (ES) and Exact Match (EM) for code completion (Guo et al., 2023), Accuracy for question answering (Rando et al., 2025), Resolve Rate on SWE-Bench Verified (Jimenez et al., 2024), and Average LLM-as-a-Judge Score on SWE-QA (Peng et al., 2025). Compression efficiency is quantified via Compression Ratio ($1/\tau = |C_{\text{original}}|/|C_{\text{compressed}}|$), absolute Token Consumption, interaction Rounds, and API Cost ($).

| Agent | Rounds | Solved | Success (%) | Tokens (M) | Cost ($) |
|---|---|---|---|---|---|
| Mini SWE Agent (Claude Sonnet 4.5) | 51.0 | 353/500 | 70.6 | 0.911 | 0.504 |
| + SWE-Pruner | 41.7 | 351/500 | 70.2 | 0.701↓23.1% | 0.369↓26.8% |
| Mini SWE Agent (GLM 4.6) | 49.3 | 277/500 | 55.4 | 0.791 | 0.055 |
| + SWE-Pruner | 36.6 | 274/500 | 54.8 | 0.488↓38.3% | 0.035↓36.4% |

Table 1: Results on SWE-Bench Verified. SWE-Pruner reduces token consumption by 23–38% while maintaining comparable success rates.

| Method | Avg Score | Avg Rounds | Tokens (K) |
|---|---|---|---|
| *Streamlink* | | | |
| Claude Sonnet 4.5 | 8.36 | 23.4 | 611.2 |
| + SWE-Pruner | **8.59** ↑0.23 | 23.9 | 557.1↓8.9% |
| GLM-4.6 | 8.56 | 18.2 | 318.2 |
| + SWE-Pruner | **8.56**↑0.00 | 25.0 | 145.1↓54.4% |
| *Reflex* | | | |
| Claude Sonnet 4.5 | 8.68 | 33.2 | 1081.6 |
| + SWE-Pruner | **8.85** ↑0.17 | 32.4 | 866.8↓19.9% |
| GLM-4.6 | 8.37 | 26.1 | 142.3 |
| + SWE-Pruner | **8.23**↓0.14 | 36.7 | 101.2↓28.9% |
| *Conan* | | | |
| Claude Sonnet 4.5 | 8.70 | 23.9 | 654.7 |
| + SWE-Pruner | **8.84** ↑0.14 | 23.5 | 520.7↓20.5% |
| GLM-4.6 | 8.58 | 21.4 | 175.9 |
| + SWE-Pruner | **8.45**↓0.13 | 27.7 | 116.6↓33.7% |

Table 2: Results on SWE-QA across different repositories. SWE-Pruner achieves 29–54% token reduction on Streamlink, Reflex, and Conan repositories with minimal impact on task performance.

Table 3: Comparison of context compression strategies on SWE-Bench. SWE-Pruner achieves highest success rate with lowest token usage.

| Method | Rounds | Success (%) | Tokens (M) |
|---|---|---|---|
| Mini SWE Agent | 52.3 | 62.0 | 0.972 |
| + LLMLingua2 | 42.1 | 54.0 | 0.856 |
| + RAG | **40.2** | 50.0 | 0.771 |
| + LLM Summarize | 41.3 | 56.0 | 0.794 |
| + LongCodeZip | 44.3 | 54.0 | 0.889 |
| **+ SWE-Pruner** | 41.1 | **64.0** | **0.670** |

## 5 Results

### 5.1 Performance on Multi-Turn Tasks

**Evaluation across Diverse Tasks.** We integrate SWE-Pruner into the Mini SWE Agent (Yang et al., 2024b) framework for SWE-Bench Verified (Jimenez et al., 2024) and the OpenHands (Team, 2024) framework for SWE-QA (Peng et al., 2025), evaluating with different backbone models. On SWE-Bench Verified, SWE-Pruner achieves substantial token reductions of 23–38% across models while maintaining nearly identical success rates (less than 1% degradation). Notably, interaction rounds decrease by 18–26%. By filtering redundant information while preserving task-relevant code, SWE-Pruner enables agents to locate issues more precisely and make more decisive decisions, thereby reducing repeated exploratory file reads and completing tasks earlier. We conduct in-depth case studies in Appendix I to illustrate these behavioral differences, showing how context pruning transforms both failure-to-success scenarios (with 83.3% token reduction) and successful trajectories (with 30.2% reduction in peak prompt length). On SWE-QA, similar efficiency gains emerge with 29–54% token reduction across repositories. Interestingly, we observe that GLM-4.6 exhibits increased rounds (29–41% more) while Claude Sonnet 4.5 maintains similar round counts with minor variations (within 3%). Through trajectory analysis, we find that after pruning, GLM tends to explore more files before formulating answers, suggesting a more conservative reasoning strategy when presented with focused context. Nevertheless, the overall token consumption remains substantially lower, demonstrating that SWE-Pruner maintains efficiency advantages regardless of the agent's exploration patterns. These model-agnostic efficiency gains directly translate to proportional cost savings. Detailed analysis is provided in Appendix E.

**Comparison with Alternative Context Management Strategies.** To understand what design choices contribute to SWE-Pruner's effectiveness, we compare against three categories of baselines: token-level compression (LLMLingua2), coarse-grained retrieval (RAG), and generative summarization (LLM Summarize). Due to computational cost considerations, we conduct this comparison on a random subset with 50 samples of SWE-Bench Verified following Xia et al. (2025); Chen

| Methods | Long Code Completion | | | | | | Long Code QA | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 4x constraint | | | 8x constraint | | | 4x constraint | | 8x constraint | |
| | $1/\tau$ | ES | EM | $1/\tau$ | ES | EM | $1/\tau$ | Acc | $1/\tau$ | Acc |
| Full | 1.0 | 64.65 | 40.5 | 1.0 | 64.65 | 40.5 | 1.0 | 54.05 | 1.0 | 54.05 |
| No Context | $\infty$ | 44.90 | 13.5 | $\infty$ | 44.90 | 13.5 | $\infty$ | 38.39 | $\infty$ | 38.39 |
| Selective-Context | 3.27 | 52.48 | 22.0 | 7.49 | 48.67 | 17.0 | 3.69 | 55.36 | 7.32 | 51.79 |
| LLMLingua2 | 3.32 | 49.47 | 15.5 | 7.89 | 44.74 | 13.0 | 3.57 | 55.36 | 7.68 | 51.33 |
| RAG | 3.29 | 58.97 | 30.5 | 6.60 | 55.82 | 29.0 | 3.06 | 58.04 | 5.87 | 55.86 |
| LongCodeZip | 2.77 | 57.77 | 28.0 | 7.85 | 56.08 | 27.5 | 3.98 | 52.25 | 7.39 | 54.95 |
| **SWE-Pruner** | **5.56** | **58.63** | **31.5** | **10.92** | **57.58** | **31.0** | **13.95** | **59.46** | **14.84** | **58.71** |

Table 4: Main results on Long Code Completion and Long Code QA tasks. The table compares the performance and compression ratio ($1/\tau$) under 4x and 8x constraints. SWE-Pruner demonstrates superior effectiveness in maintaining high performance while achieving significant context compression.

et al. (2024). As shown in Table 3, our method achieves the best success rate (64%), outperforming the vanilla agent baseline (62%) despite using 31% fewer tokens. In contrast, LLMLingua2 and RAG degrade performance substantially (to 54% and 50%), likely because token-level pruning disrupts code syntax while coarse-grained retrieval misses fine-grained implementation details. LLM Summarize achieves moderate performance (56%) but incurs additional latency. These results validate our design choice of line-level granularity with task-aware filtering, striking an optimal balance between compression and information retention. The effectiveness of SWE-Pruner is further validated by consistent improvements on single-turn tasks (Table 4), where we observe similar advantages across diverse compression constraints.

### 5.2 Performance on Single-Turn Tasks

While SWE-Pruner is designed for coding agents, its query-aware, line-level pruning mechanism generalizes to single-turn tasks. We evaluate on Long Code Completion and Long Code QA under 4x and 8x compression constraints with Qwen2.5-Coder-7B-Instruct (Hui et al., 2024). These benchmarks require direct completion or question answering without iterative exploration. Notably, SWE-Pruner achieves substantially higher effective compression ratios than baselines configured at identical compression targets. We also validate with Seed-Coder-8B-Instruct (ByteDance Seed et al., 2025), obtaining similar results (see Appendix G).

On Long Code Completion (Table 4), SWE-Pruner achieves up to 10.92x effective compression under the 8x constraint while maintaining an Edit Similarity (ES) score of 57.58 and an Exact Match (EM) rate of 31.0. As compression constraints tighten, token-level methods (LLMLingua2, Selective-Context) experience rapid performance degradation—Selective-Context drops to 48.67 ES under 8x constraint—whereas SWE-Pruner maintains stable performance through line-level granularity that preserves syntactic structure. Under the 4x constraint, SWE-Pruner achieves 5.56x compression with 58.63 ES and 31.5 EM, outperforming all baselines. On Long Code QA, the advantages become even more pronounced. SWE-Pruner achieves 14.84x compression under 8x constraint with 58.71% accuracy, substantially exceeding other baselines. Under the 4x constraint, our method achieves 13.95x compression while maintaining 59.46% accuracy, demonstrating that task-aware line-level pruning excels at identifying relevant code segments for question answering. This highlights the synergy between query-aware filtering and line-level granularity, validating that our approach generalizes effectively to single-turn code understanding tasks.

### 5.3 Efficiency Impact

Beyond token reduction, a critical consideration for practical deployment is the computational overhead introduced by the skimmer itself. As shown in Figure 4, our approach maintains remarkably low and stable first token latency across all sequence lengths, remaining below 100ms even at 8K tokens. In contrast, larger generative models exhibit exponential latency growth, with Qwen3-32B exceeding 1200ms and closed-source models incurring even more overhead. Since our skimmer employs only a 0.6B encoder, its computational cost is negligible and readily amortized by the reduced decoding cost from context compres-

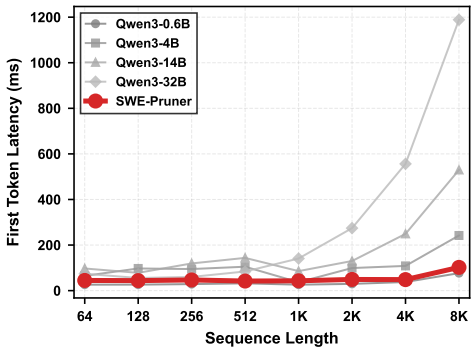



Figure 4: First token latency comparison across different sequence lengths. SWE-Pruner maintains consistently low latency below 100 ms.

sion. This lightweight characteristic validates the practical viability of our approach for real-world applications. Detailed latency measurements are provided in Table 6.

## 6 Related Works

**Prompt Compression** Prompt compression has been extensively studied for both natural language and code. Token-level pruning methods such as LLMLingua (Jiang et al., 2023, 2024), Selective Context (Li et al., 2023c), and AttentionRAG (Fang et al., 2025), along with embedding/retrieval-based approaches like RAG (Lewis et al., 2020), XRAG (Cheng et al., 2024), and repo-level retrieval for code completion (Zhang et al., 2023), can reduce prompt length on single-round tasks but often fail to preserve syntactic structures essential for code correctness (Shi et al., 2025). Code-specific methods including DietCode (Zhang et al., 2022), SlimCode (Wang et al., 2024b), and LongCodeZip (Shi et al., 2025) address structural concerns but are primarily evaluated on single-round proxy tasks (e.g., code search, completion) rather than multi-round agentic workflows. Moreover, these approaches typically apply static, content-only compression that lacks contextual awareness, leading to fixed behaviors such as indiscriminately removing comments (Yang et al., 2024a) regardless of task requirements. In contrast, SWE-Pruner performs line-level pruning with dynamic, query-conditioned thresholding that adapts to the agent's current task stage, enabling context-aware compression validated on end-to-end benchmarks like SWE-Bench (Jimenez et al., 2024) without repository-specific retraining.

**Agent Context Management** Modern coding LLMs support context windows of 128k tokens or more (Achiam et al., 2023; Team et al., 2024), yet they remain insufficient for large-scale codebases and suffer from documented performance degradation on long contexts (Liu et al., 2023a; Laban et al., 2025; Li et al., 2023a). Agentic systems (Yao et al., 2023; Wei et al., 2022) for software engineering (Yang et al., 2024b; Wang et al., 2024a; Xia et al., 2024; Bouzenia et al., 2024; Qin et al., 2024; Liu et al., 2024) face critical challenges in managing multi-turn interaction contexts. In NLP domains, trajectory management approaches employ LLM-based summarization when contexts overflow (Cursor, 2025; Anthropic, 2025), fixed-size truncation (Gao et al., 2025), or simple observation masking (Lindenbauer et al., 2025). For long-horizon scenarios, recent frameworks such as SUPO (Lu et al., 2025) and FoldGRPO (Sun et al., 2025) learn to manage history through reinforcement learning, while others like COMPASS (Wan et al., 2025), ACON (Kang et al., 2025), AgentDiet (Xiao et al., 2025), and AgentFold (Ye et al., 2025) introduce hierarchical oversight or proactive folding policies. These methods primarily optimize prior interaction histories, whereas SWE-Pruner serves as lightweight middleware at the agent–environment boundary to prune the environment's observation (i.e., file content); it is thus orthogonal to and can be seamlessly combined with such learned history managers.

## 7 Conclusion

Pruning long code context for agents requires both fine-grained, structure-preserving compression and dynamic, task-aware filtering. Our approach, SWE-Pruner, addresses these challenges through lightweight binary classifiers trained on synthetically diversified code-question pairs and an adaptive, query-conditioned thresholding mechanism. By integrating seamlessly as middleware within agentic workflows, SWE-Pruner reduces token usage by 23–38% on SWE-Bench and 29–54% on SWE-QA, achieves up to 14.84× compression on single-turn benchmarks. These results demonstrate that line-level, context-aware pruning effectively addresses context-window constraints across both agentic workflows and general code understanding tasks while reducing costs and improving efficiency.

## Limitations

We acknowledge several limitations. First, our implementation focuses on Python repositories, though our approach does not rely on Python-specific features and demonstrates effective generalization across different codebases. Comprehensive multilingual support remains future work. Second, we mitigate data leakage by selecting recently collected repositories from SWE-QA that postdate our training data, though continuous evaluation on newly released repositories remains important. Finally, while our lightweight neural skimmer significantly reduces token consumption, it introduces marginal latency overhead that could be further optimized through distillation or early-exit mechanisms.

## References


Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2025. Claude code: Built for developers.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, and 8 others. 2025. Seed-Coder: Let the code model curate data for itself. *Preprint*, arXiv:2506.03524.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, and 1 others. 2024. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. 2025. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. 2024. xRAG: Extreme Context Compression for Retrieval-augmented Generation with One Token.

Cursor. 2025. Cursor – the ai code editor.

Yixiong Fang, Tianran Sun, Yuling Shi, and Xiaodong Gu. 2025. Attentionrag: Attention-guided context pruning in retrieval-augmented generation. *arXiv preprint arXiv:2503.10720*.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, and 1 others. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*.

Google. 2025. Gemini cli: Command-line ai assistant.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR.

Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: Llms as the new software judges. *arXiv preprint arXiv:2503.02246*.

Minghao Hu, Junzhe Wang, Weisen Zhao, Qiang Zeng, and Lannan Luo. 2025. Flowmaltrans: Unsupervised binary code translation for malware detection using flow-adapter architecture. *arXiv preprint arXiv:2508.20212*.

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

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. 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*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *ICLR*.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. 2025. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. 2025. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*.

Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023a. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023c. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. 2025. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023a. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023b. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. 2025. Scaling llm multi-turn rl with end-to-end summarization-based context management. *arXiv preprint arXiv:2510.06727*.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. 2025. Understanding code agent behaviour: An empirical study of success and failure trajectories. *arXiv preprint arXiv:2511.00197*.

Jesse Mu, Xiang Li, and Noah Goodman. 2023. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36:19327–19352.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, and 1 others. 2024. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 963–981.

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

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. Longcodebench: Evaluating coding llms at 1m context windows. *arXiv preprint arXiv:2505.07897*.

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*.

Haoyang Song, Haochen Liu, Lingfeng Sha, Hongyu Zhang, and Xiaodong Gu. 2024. Finecodeeval: Fine-grained code evaluation using llm as a judge. *arXiv preprint arXiv:2406.12170*.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. 2025. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1

others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

OpenHands Team. 2024. Openhands: An open platform for ai software developers as generalist agents. https://github.com/All-Hands-AI/OpenHands.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Tree-sitter. 2024. Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

Guangya Wan, Mingyang Ling, Xiaoqi Ren, Rujun Han, Sheng Li, and Zizhao Zhang. 2025. Compass: Enhancing agent long-horizon reasoning with evolving context. *arXiv preprint arXiv:2510.08790*.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025a. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1955–1977.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2024a. OpenHands: An Open Platform for AI Software Developers as Generalist Agents.

Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024b. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE):586–608.

Yifei Wang, Feng Xiong, Yong Wang, Linjing Li, Xiangxiang Chu, and Daniel Dajun Zeng. 2025b. Position bias mitigates position bias: Mitigate position bias through inter-position knowledge distillation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1495–1512.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2025. Improving the efficiency of llm agent systems through trajectory reduction. *arXiv preprint arXiv:2509.23586*.

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2024a. Less is More: Doc-String Compression in Code Generation. *Preprint*, arXiv:2410.22793.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024b. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Xinwei Yang, Zhaofeng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025. Elaboration: A comprehensive benchmark on human-llm competitive programming. *arXiv preprint arXiv:2505.16667*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023*.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. 2025. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*.

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

Lei Zhang, Yunshui Li, Jiaming Li, Xiaobo Xia, Jiaxi Yang, Run Luo, Minzheng Wang, Longze Chen, Junhao Liu, and Min Yang. 2024. Hierarchical Context Pruning: Optimizing Real-World Code Completion with Repository-Level Pretrained Code LLMs. *Preprint*, arXiv:2406.18294.

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

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. 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*, pages 1073–1084.

Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. 2015. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1529–1537.

# A Empirical Results on GLM Model

In our main experiments, we demonstrated that Claude Sonnet 4.5 dedicates 76.1% of its token budget to read operations when solving software engineering tasks on SWE-Bench Verified. To verify whether this token consumption pattern generalizes across different model architectures, we extend our analysis to GLM-4.6, an open-source large language model with fundamentally different training methodology and architecture. We conduct parallel analysis on 178 GLM-4.6 agent trajectories using the same experimental setup as in Section 3, categorizing token usage into read, edit, and execute operations. The results reveal consistent patterns that validate the model-agnostic nature of context pruning needs in coding agents.

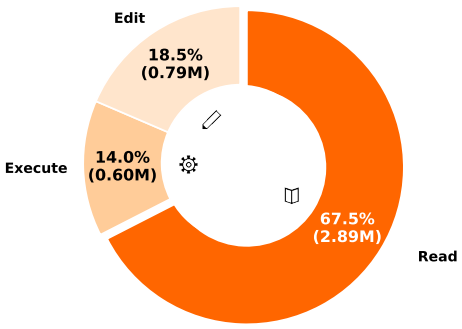



Figure 5: **Token cost distribution** over different tool calls for Mini-SWE-Agent with GLM-4.6 on SWE-Bench Verified. Read operations dominate token consumption at 67.5%, further validating the necessity of context pruning mechanisms across different backbone models.

As illustrated in Figure 5, GLM-4.6 exhibits remarkably similar token consumption patterns to Claude Sonnet 4.5. Read-type operations account for 67.5% of total tokens (2.89M), demonstrating that the dominance of codebase exploration remains consistent across different model architectures. Edit and execute operations consume 18.5% (0.79M) and 14.0% (0.60M) respectively, with their relative proportions slightly different from Claude Sonnet 4.5 but still maintaining read operations as the overwhelming majority. This cross-model consistency strongly reinforces our motivation for context pruning: regardless of the underlying model architecture, training methodology, or parameter scale, coding agents universally spend the majority of their token budget on codebase exploration rather than reasoning or editing. These findings establish that SWE-Pruner addresses a fundamental inefficiency inherent to the agentic workflow itself, rather than model-specific behavior.

# B Model Architecture and Inference Details

## B.1 Model Architecture

The neural skimmer extends the Qwen3-Reranker-0.6B backbone with two specialized heads: a CRF-based pruning head for line-level filtering and a reranking head for document-level scoring. An overview of the architecture is shown in Figure 6.

**Multi-Layer Feature Fusion.** To capture representations at different semantic levels, we extract and concatenate hidden states from three intermediate layers (layers 7, 14, and 28) of the backbone. These fused features are processed through a self-attention block followed by a multi-head attention layer (8 heads, hidden size 256) to refine contextual representations.

**CRF-Based Pruning Head.** The pruning head formulates line-level retention as a structured sequence labeling problem using a Conditional Random Field. Let $\mathbf{x}$ denote the input token sequence and $\mathbf{y}$ the corresponding label sequence in space $\mathcal{Y} = \{\text{retain}, \text{prune}\}$. The CRF negative log-likelihood is:

$$\mathcal{L}_{\text{CRF-NLL}}(\mathbf{x}, \mathbf{y}) = \log Z(\mathbf{x}) - \text{score}(\mathbf{x}, \mathbf{y}) \quad (5)$$

where the score function combines emission and transition potentials:

$$\text{score}(\mathbf{x}, \mathbf{y}) = \text{start}_{y_1} + \sum_{t=1}^{T} \text{emissions}_{t,y_t} + \sum_{t=2}^{T} \text{transitions}_{y_t,y_{t-1}} + \text{end}_{y_T} \quad (6)$$

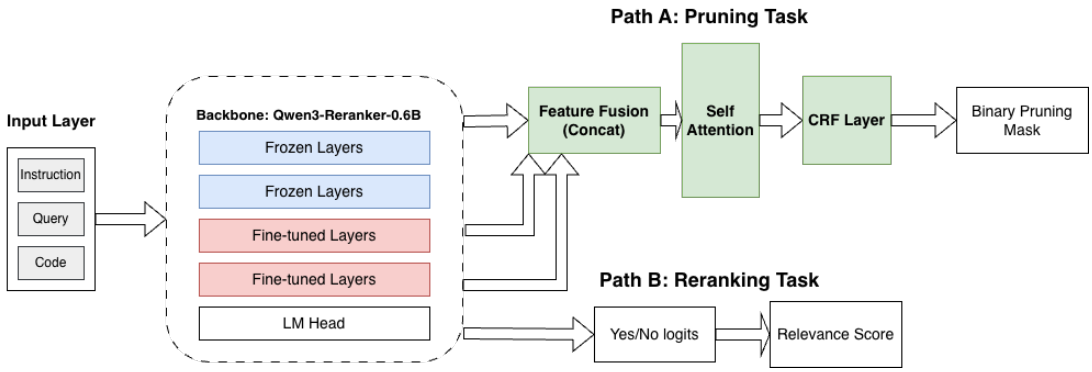



Figure 6: **SWE-Pruner Model Architecture.** The model consists of a lightweight reranker backbone with multi-layer feature fusion, followed by dual heads for pruning and reranking. The pruning head employs a CRF layer to model structured retention decisions, while the reranking head produces document-level relevance scores.

and the partition function normalizes over all possible label sequences:

$$\log Z(\mathbf{x}) = \log \sum_{\mathbf{y}' \in \mathcal{Y}} \exp(\text{score}(\mathbf{x}, \mathbf{y}')) \quad (7)$$

Emissions $\mathbf{E}_t = \text{MLP}(\mathbf{h}_t) \in \mathbb{R}^2$ represent local confidence for each token, while transitions $\mathbf{T} \in \mathbb{R}^{2\times 2}$ capture dependencies between adjacent decisions. This structured formulation encourages coherent pruning patterns that respect syntactic boundaries.

**Reranking Head.** The reranking head reuses the original language modeling head from Qwen3-Reranker, producing a scalar relevance score for the entire document via mean squared error against teacher-provided scores.

### B.2 Inference Strategy

During inference, the pruning head computes token-level scores through forward propagation, which are then aggregated to line-level scores via averaging (as detailed in Algorithm 1). The CRF layer applies Viterbi decoding to find the optimal label sequence, ensuring structurally coherent pruning decisions. Lines are retained if their average token score exceeds the threshold $\tau = 0.5$. The reranking head simultaneously produces document-level scores, enabling the system to perform both granular pruning and coarse-grained relevance assessment in a single forward pass.

Algorithm 1 summarizes the complete scoring and aggregation pipeline. The algorithm first computes token-level relevance scores for all input tokens (Step 2), then iterates through each line to aggregate token scores via averaging (Steps 4-7), and finally applies the threshold-based retention criterion (Step 8) to produce the final set of kept lines.

**Algorithm 1** Token Scoring and Line-level Aggregation

**Require:** Input text $X = \{x_1, x_2, ..., x_n\}$, token-level scoring model $f(\cdot)$, threshold $\tau$
**Ensure:** Set of kept lines $L_{kept}$
1: $L_{kept} = \emptyset$
2: Compute token scores $S = \{s_1, s_2, ..., s_n\}$ where $s_i = f(x_i)$
3: **for** each line $l_i \in X$ **do**
4: Let $T_i$ be the set of tokens in line $l_i$
5: Compute line score $\bar{s}_i = \frac{1}{|T_i|} \sum_{t \in T_i} s_t$
6: **if** $\bar{s}_i > \tau$ **then**
7: $L_{kept} \leftarrow L_{kept} \cup \{l_i\}$
8: **end if**
9: **end for**
10: **return** $L_{kept}$

## C Training Dataset for the Neural Skimmer

**Code Source and Preprocessing.** We construct our training dataset from code snippets sampled from the GitHub Code 2025 dataset[3], a meticulously curated collection comprising over 1.5 million repositories. This dataset employs a dual-perspective design that balances proven quality with emerging innovation: it includes both high-quality repositories (above 2 stars) representing established patterns and practices, as well as newly-created 2025 repositories capturing contemporary development trends. The dataset undergoes exten-

[3] https://huggingface.co/datasets/nick007x/github-code-2025

sive preprocessing to remove binary files, build artifacts, configuration noise, and minified code, ensuring that only clean, meaningful source code is retained. This rigorous curation provides us with a diverse, polyglot code corpus spanning multiple programming languages and coding paradigms, which is essential for training a pruning model that generalizes across varied agentic scenarios.

**Agentic Task Taxonomy.** To ensure the skimmer generalizes across diverse real-world coding scenarios, we design a comprehensive taxonomy of nine distinct agentic task types that reflect common information needs in software development workflows. These tasks encompass both exploratory activities and focused interventions. *Code Summarization* (`code-summarize`) targets high-level understanding by requesting concise summaries of code functionality for integration or review purposes. *Code Refactoring* (`code-refactor`) focuses on improving code quality through readability, modularity, or structural enhancements. *Relevant Part Identification* (`find-relevant-part`) and *Code Location* (`code-locate`) address navigation needs by asking to identify where specific features, bugs, or logic are implemented. *Code Optimization* (`code-optimize`) requests efficiency improvements in terms of performance, resource usage, or scalability. *Code Explanation* (`code-explain`) seeks understanding of particular algorithms or design choices without requiring full code walkthroughs. *Code Debugging* (`code-debug`) targets actionable assistance for resolving specific issues, exceptions, or edge cases. *Feature Addition* (`feature-addition`) involves extending existing code with new capabilities while maintaining integration with current logic. Finally, *Code Completion* (`code-completion`) represents a unique scenario where the query itself is a code snippet requiring contextual completion, simulating realistic code intelligence workflows. This taxonomy ensures broad coverage of task diversity while maintaining clear distinctions between query intents. Detailed generation instructions for each task type are provided in Table 5.

**Data Generation Pipeline.** We synthesize queries and line-level labels for 200,000 sampled code snippets from 195,370 files across 5,945 repositories. For each code snippet $C$, we employ Qwen3-Coder-30B-A3B-Instruct (Bai et al., 2023) as the teacher LLM to generate task-oriented queries and corresponding line-level retention masks. Query generation employs a temperature of 0.7 and top-p sampling with $p = 0.9$ to balance diversity and coherence. To ensure representativeness, we randomly sample across all nine task types, three snippet length levels (short, medium, long), and three relevance levels (low, medium, high). Following initial generation, we apply a rigorous quality control mechanism using Qwen3-Next-80B-A3B-Thinking as an LLM-as-a-Judge filter (Wang et al., 2025a; Liu et al., 2023b; Song et al., 2024; He et al., 2025). This judge model evaluates the reasoning quality, annotation consistency, and task alignment of each sample, retaining approximately 1/6 of candidates that meet high-quality standards. The complete generation prompts and judge criteria are detailed in Appendix J.

**Dataset Statistics.** Following filtering and quality enhancement procedures, we obtain 61,184 training samples with verified line-level annotations. The resulting dataset exhibits natural query length distributions with an average of 39.98 words and a median of 24.00 words per query, reflecting realistic information needs in agentic workflows. The average character count per query is 291.69, with a median of 169.00, indicating a balanced mix of concise and detailed information requests. This final corpus provides diverse, high-quality supervision for training the neural skimmer to handle varied agentic coding scenarios.

# D Experimental Details

This section provides comprehensive details about our experimental setup, including hardware configuration, training hyperparameters, benchmark specifications, agent frameworks, and baseline configurations.

## D.1 Hardware and Training Configuration

**Hardware.** All model training is conducted on a server equipped with 8 NVIDIA H100-80G GPUs. The lightweight neural skimmer (0.6B parameters) is deployed on a single GPU for inference, introducing negligible latency overhead compared to the main agent LLM.

**Training Hyperparameters.** We fine-tune the Qwen3-Reranker-0.6B backbone with a global batch size of 128, obtained from a per-device batch size of 16 on 8 GPUs with tensor paral-

Table 5: Agentic Tasks Taxonomy used for Query Synthesis

| Task Type | Instruction for Query Generation |
|---|---|
| `code-summarize` | Summarize the main purpose or functionality of the code, but do not explain every line. Frame your query as a developer seeking a summary for integration or review. |
| `code-refactor` | Suggest a refactoring or improvement for the code. Your query should be practical, such as asking to improve readability, modularity, or performance. |
| `find-relevant-part` | Ask to locate or identify the part of the code that implements a specific feature or logic. Your query should be about finding where something is handled in the code. |
| `code-optimize` | Request an optimization for the (core logic maybe) code, such as improving efficiency, reducing resource usage, or enhancing scalability. |
| `code-locate` | Ask to pinpoint the location of a bug, feature, or important logic within the code. |
| `code-explain` | Request an explanation for a particular logic, algorithm, or design choice in the code, but do not ask for a full code walkthrough. |
| `code-debug` | Ask for help debugging a specific issue, exception, or edge case in the code. Your query should be actionable and focused. |
| `feature-addition` | Request to add a new feature or capability to the code, specifying what should be added and how it should interact with existing logic. |
| `code-completion` | This is a special query format. In code completion, the query should be CODE instead of text, which means you should image yourself as a developer write other code snippet(query) that can used the code given for completion. The completion will be the next line for query, but you should keep it in your mind and never write the completion in query. QUERY like a PUZZLE. |

lelism. We use the AdamW optimizer with a learning rate of $3 \times 10^{-5}$ and weight decay of 0.01, training for 3 epochs with a dropout rate of 0.4. The CRF-based pruning head fuses hidden states from three intermediate transformer layers (layers 7, 14, and 28). Only the last two transformer layers of the backbone are fine-tuned, along with an additional feature fusion module consisting of a self-attention block followed by multi-head attention (8 heads, hidden size 256) and a CRF layer. The weight balancing parameter is set to $\lambda = 0.05$. Detailed architecture specifications are provided in Appendix B.

**Inference Configuration.** The pruning threshold is set to $\tau = 0.5$, tuned on a held-out validation set. Agent tasks use a maximum of 250 interaction rounds. All experiments employ Claude Sonnet 4.5 and GLM-4.6 APIs with temperature 0 for deterministic generation, averaged over three random seeds where applicable.

### D.2 Benchmark and Agent Descriptions

We evaluate SWE-Pruner on both single-turn and multi-turn benchmarks spanning diverse code intelligence tasks. For single-turn evaluation, we use Long Code Completion (Guo et al., 2023), which evaluates code completion under long contexts, and Long Code QA (Rando et al., 2025), which tests code comprehension through question answering on contexts up to 1 million tokens drawn from real-world GitHub issues and documentation across multiple languages and project domains. For multi-turn agent benchmarks, we use SWE-Bench Verified (Jimenez et al., 2024), which contains 500 GitHub issues from 12 Python repositories where success is measured by automated test execution in Docker containers and patches must pass all tests without regressions,

and SWE-QA (Peng et al., 2025), which evaluates repository-specific question answering across three repositories (streamlink, reflex, conan) with answers scored via LLM-as-a-judge across five dimensions: correctness, completeness, relevance, clarity, and reasoning.

For agent frameworks, we integrate SWE-Pruner with two representative systems. Mini SWE Agent (Yang et al., 2024b) is a typical agent framework designed for SWE-Bench that operates with three tool categories: file reading (`cat`, `grep`), code editing (`sed`), and command execution. We integrate SWE-Pruner as middleware intercepting file reading operations to apply task-aware pruning. OpenHands (Team, 2024), which we use following the SWE-QA original paper, provides comprehensive tools including repository exploration, version control, and debugging. Our integration with both frameworks demonstrates that SWE-Pruner generalizes across agent architectures as a modular component.

### D.3 Baseline Configurations

All baseline methods are configured to match identical compression constraints (4x and 8x) under the same experimental conditions. For performance bounds, **Full Context** provides complete code context as an upper bound, while **No Context** provides only task instructions as a lower bound. For token-level compression baselines, **Selective-Context** (Li et al., 2023c) computes self-information $-\log P(x_i|\text{instruction})$ and removes low-information tokens, and **LLMLingua-2** (Pan et al., 2024) uses a trained token classifier (XLM-RoBERTa) to predict binary retention decisions. Both operate at sub-word granularity without code-specific syntactic constraints. For retrieval-based methods, **RAG** employs function-level chunking with UniXCoder (Guo et al., 2022; Zhang et al., 2023) embeddings, retrieving top-k chunks via cosine similarity. **LongCodeZip** (Shi et al., 2025) represents code-specific compression that combines AST-based chunking with entropy-guided compression, retaining high-entropy regions. Finally, **LLM Summarize** (for multi-turn agent tasks only) generates abstractive summaries of code files using the backbone model, trading summarization cost for reduced downstream tokens.

## E Agent Rounds and Token Consumption Analysis

Section 4.3 shows the token consumption and interaction rounds for two different backbone models: Claude Sonnet 4.5 and GLM 4.6. For Claude Sonnet 4.5, SWE-Pruner reduces average tokens per instance from 0.911M to 0.701M (23.1% reduction), while reducing average rounds from 51.0 to 41.7 (18.2% reduction). The GLM 4.6 model exhibits even more substantial improvements, with token reduction from 0.791M to 0.488M (38.3% reduction) and rounds reduction from 49.3 to 36.6 (25.7% reduction).

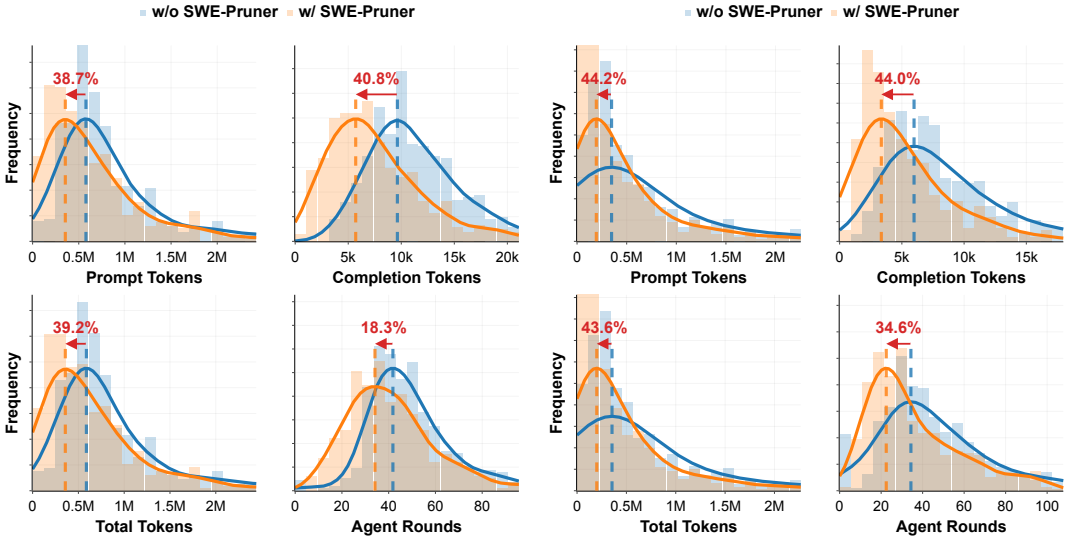
Figure 7 present the complete distribution shifts for both models, revealing consistent patterns across prompt and completion tokens. GLM 4.6 exhibits 44.2% and 44.0% reductions respectively, while Claude Sonnet 4.5 demonstrates 38.7% and 40.8% reductions. This symmetry indicates that SWE-Pruner not only reduces input context but also enables more focused agent responses, distinguishing it from naive retrieval filtering which would primarily affect prompt tokens. The variation in pruning effectiveness between models provides insights into architectural differences: GLM 4.6's more pronounced gains (34.6% round reduction vs. Claude's 18.3%) suggest greater susceptibility to context noise, while Claude's extensive context capabilities maintain reasonable focus even unpruned. The reduction in agent rounds represents a qualitative improvementfor GLM 4.6, the shift from 49.3 to 36.6 rounds translates to faster task completion and reduced cumulative latency in production environments where each round incurs API overhead and rate-limiting delays.

## F Detailed Efficiency Analysis

| Model | Input Length | | | | |
|---|---|---|---|---|---|
| | 64 | 128 | 512 | 2048 | 8192 |
| Qwen3-0.6B | 26.18 | 26.31 | 32.00 | 29.64 | 76.73 |
| Qwen3-4B | 64.75 | 97.24 | 104.93 | 99.10 | 241.97 |
| Qwen3-14B | 97.17 | 78.11 | 143.65 | 129.97 | 529.45 |
| Qwen3-32B | 73.99 | 55.46 | 84.01 | 274.22 | 1188.67 |
| SWE-Pruner | 44.70 | 43.91 | 42.05 | 49.05 | 102.00 |

Table 6: Average TTFT (ms) for different models and input lengths. SWE-Pruner maintains consistently low latency across all sequence lengths.

To complement the efficiency analysis in Section 5.3, we provide detailed first token latency




(a) Claude Sonnet 4.5 (b) GLM 4.6

Figure 7: **Comprehensive efficiency analysis on SWE-Bench Verified.** SWE-Pruner (orange) achieves substantial reductions compared to baseline (blue). (a) With Claude Sonnet 4.5: 38.7% in prompt tokens, 40.8% in completion tokens, 39.2% in total tokens, and 18.3% in agent rounds. (b) With GLM 4.6: 44.2% in prompt tokens, 44.0% in completion tokens, 43.6% in total tokens, and 34.6% in agent rounds.

measurements across different models and input lengths. Table 6 presents a comprehensive comparison demonstrating how SWE-Pruner maintains stable and low latency across various sequence lengths compared to larger generative models.

The latency analysis reveals critical insights for practical agent deployments. At 8192 tokens, SWE-Pruner achieves 102.00ms TTFTa 7.5× speedup compared to Qwen3-32B (1188.67ms) and 5.2× compared to Qwen3-14B (529.45ms)while exhibiting sublinear scaling (2.1× increase from 2048 to 8192 tokens vs. Qwen3-32B's 14.1×). Crucially, in real-world deployments with closed-source models like Claude Sonnet 4.5, API latency typically ranges from 500ms to several seconds per request, making our pruning overhead (40–50ms) less than 10% of typical roundtrip times. This marginal cost is amortized many times over through 23–54% token reductions (Section 4.3), yielding proportional savings in both inference time and API costs. Combined with 18.3–25.7% reductions in interaction rounds across models, the efficiency gains compound throughout multi-turn agent trajectories, delivering substantial end-to-end improvements despite the upfront pruning cost.

## G Single-Turn Tasks with SeedCoder

To validate the generalizability of our approach across different model architectures, we conduct additional evaluations using Seed-Coder-8B-Instruct on the Long Code Completion and Long Code QA benchmarks. Results are presented in Table 7.

The results with Seed-Coder-8B-Instruct exhibit similar patterns to those observed with Qwen2.5-Coder-7B-Instruct. On Long Code Completion, SWE-Pruner achieves 8.13x compression under 8x constraint with 56.73 ES and 28.5 EM, outperforming Selective-Context (7.49x/49.89 ES/17.5 EM) and LongCodeZip (6.53x/54.91 ES/23.0 EM). On Long Code QA, the advantages are particularly pronounced: under 8x constraint, our method achieves 14.68x compression with 55.75% accuracy, substantially exceeding all baselines including RAG (6.65x/53.57%) and LongCodeZip (7.49x/50.91%). This cross-model consistency validates that the effectiveness of task-aware, line-level pruning is not specific to a particular model architecture but represents a generalizable approach to context compression for code understanding tasks.

| Methods | Long Code Completion | | | | | | Long Code QA | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 4x constraint | | | 8x constraint | | | 4x constraint | | 8x constraint | |
| | $1/\tau$ | ES | EM | $1/\tau$ | ES | EM | $1/\tau$ | Acc | $1/\tau$ | Acc |
| Full | 1.0 | 65.03 | 40.5 | 1.0 | 65.03 | 40.5 | 1.0 | 49.11 | 1.0 | 49.11 |
| No Context | $\infty$ | 43.85 | 14.0 | $\infty$ | 43.85 | 14.0 | $\infty$ | 37.50 | $\infty$ | 37.50 |
| Selective-Context | 3.27 | 53.68 | 24.5 | 7.49 | 49.89 | 17.5 | 2.71 | 51.33 | 6.69 | 46.43 |
| LLMLingua2 | 3.95 | 48.09 | 15.0 | 7.89 | 45.53 | 13.5 | 3.70 | 43.36 | 5.46 | 39.82 |
| RAG | 3.32 | 58.21 | 31.0 | 6.78 | 56.97 | 28.5 | 3.44 | 53.98 | 6.65 | 53.57 |
| LongCodeZip | 3.96 | 56.72 | 25.5 | 6.53 | 54.91 | 23.0 | 3.29 | 51.33 | 7.49 | 50.91 |
| **SWE-Pruner** | **4.22** | **57.71** | **31.0** | **8.13** | **56.73** | **28.5** | **9.98** | **56.25** | **14.68** | **55.75** |

Table 7: Results on Long Code Completion and Long Code QA with Seed-Coder-8B-Instruct. SWE-Pruner demonstrates consistent effectiveness across different model architectures, achieving superior compression while maintaining competitive task performance.

Table 8: AST Correctness Rate after Compression. Line-level methods maintain substantially higher syntactic validity compared to token-level approaches.

| **Method** | **AST Correctness (%)** |
|---|---|
| *Baseline (No Compression)* | |
| Full Context | 98.5 |
| No Context | 98.5 |
| *Token-Level Compression* | |
| LLMLingua2 | 0.29 |
| Selective Context | 12.4 |
| Random Token Pruner | 49.6 |
| *Line-Level Compression* | |
| Function RAG | 92.3 |
| + Random Line Pruner | 78.2 |
| + **SWE-Pruner** | 87.3 |
| *Structural Compression* | |
| LongCodeZip | 89.3 |
| + **SWE-Pruner** | 76.8 |

# H Syntactic Structure Preservation Analysis

Line-level pruning preserves syntactic structure better than token-level compression. Our pruning strategy retains semantically relevant lines with minimal syntactic context for code integrity, while token-level methods disrupt AST structure through arbitrary token deletion. We evaluate AST correctness using tree-sitter (Tree-sitter) on Long Code Completion. Table 8 presents results.

Token-level methods (LLMLingua2, Selective Context) achieve near-zero AST correctness (0.29%, 12.4%), while line-level approaches maintain substantially higher validity. Our SWE-Pruner achieves 87.3% AST correctness on Function RAG, outperforming random token pruning (49.6%) and competitive with random line pruning (78.2%). This demonstrates that structure-aware labeling successfully identifies safe removals while preserving syntactic dependencies. Note that SWE-Pruner operates as a second-stage pruner on top of Function RAG's output, meaning any additional compression inherently introduces some risk of removing syntactically critical lines. The slight reduction from baseline Function RAG (92.3%) to our method (87.3%) reflects this compression-validity trade-off, where the 5% decrease enables substantially higher compression ratios while maintaining practical code validity for downstream tasks.

# I Case Study on SWE Bench

To complement the aggregate statistics from SWE-Bench experiments, we conduct an in-depth case study comparing agent behaviors under two configurations: a Baseline agent operating with full interaction histories, and a Pruner-augmented agent that applies context pruning to file observations. Both agents operate on the same set of software engineering tasks from SWE-Bench, and we analyze their trajectories through the lens of tool invocation patterns and token consumption. We select two representative instances that illustrate distinct benefits of context pruning: one where pruning enables task completion by preventing resource exhaustion, and another where pruning achieves structural efficiency gains even when both agents succeed.

## I.1 Case 1: High-Impact Scenario

The first case examines task `django__django-10554`, which requires fixing a missing deep copy in the `Query.clone()` method

for the `combined_queries` attribute. As shown in Table 9, the Baseline agent exhausts its resource limits after 164 steps, accumulating over 7 million tokens with a maximum prompt length of 87,790 tokens. In contrast, the Pruner-augmented agent completes the task successfully in 56 steps with 1.17 million tokens and a peak prompt length of 38,226 tokens. The token reduction reaches 83.3% in absolute terms, demonstrating that context pruning can transform resource-bound failures into successful completions.

Table 9: Comparison of agent behaviors on `django__django-10554`. Baseline exhausts limits while Pruner succeeds with 83.3% token reduction.

| Setting | Steps | Read | Search | Exec | Edit | Tokens |
|---|---|---|---|---|---|---|
| Baseline | 164 | 59 | 39 | 1 | 25 | 7,001,934 |
| Pruner | 56 | 20 | 10 | 0 | 11 | 1,170,160 |

Examining the trajectory reveals fundamental differences in exploration strategy. The Baseline agent engages in extensive breadth-first file reading, issuing commands such as `find . -name "*.py" | grep union` followed by repeated segmented reads using `sed -n 'x,y'` across numerous files. This results in accumulation of redundant context from tangentially related code. Meanwhile, the Pruner agent directly navigates to the core file `django/db/models/sql/query.py`, reads it with line numbers (`cat -n`), and identifies the relevant branch (`if self.combinator: ...`) for modification. By filtering out irrelevant sections during file reads, the pruner enables the agent to maintain a focused working context, avoiding the context overflow that derails the Baseline.

### I.2 Case 2: Structural Efficiency Gains

The second case analyzes task `django__django-11740`, which requires adding foreign key dependency tracking to the migration autodetector's `AlterField` operation. Unlike the previous case, both agents successfully complete this task, but the Pruner achieves substantial efficiency improvements as shown in Table 10. Although the Pruner agent takes 6 additional steps (48 versus 42), its token consumption is 6.0% lower and its maximum prompt length is reduced by 30.2%. This illustrates that pruning benefits extend beyond preventing failures to improving resource efficiency in successful trajectories.

Analysis of the trajectories shows that both agents converge on the same logical solution, modifying `autodetector.py` to adjust the `_get_dependencies_for_foreign_key` invocation. However, their reading strategies differ qualitatively. The Baseline agent performs multiple segmented reads of the target file and creates temporary validation scripts (e.g., `python /tmp/test_uuid_to_fk3.py`) to verify understanding, accumulating historical noise from these exploratory steps. The Pruner agent reads the complete file with line numbers once, directly edits the relevant section, and avoids auxiliary validation artifacts. This behavioral shift reflects a transition from defensive exploration to confident intervention, enabled by cleaner, more focused context at each decision point.

Table 10: Comparison on `django__django-11740`. Both succeed, but Pruner achieves 30.2% reduction in peak prompt length.

| Setting | Steps | Read | Search | Exec | Edit | Tokens |
|---|---|---|---|---|---|---|
| Baseline | 42 | 12 | 8 | 1 | 18 | 857,371 |
| Pruner | 48 | 14 | 8 | 0 | 13 | 806,220 |

**Discussion** These cases illustrate two distinct modes of benefit from context pruning. In the high-impact scenario, pruning prevents catastrophic context overflow that leads to task abandonment, effectively converting failures into successes. In the structural efficiency scenario, pruning does not change task outcomes but significantly reduces the computational burden and maximum context requirements, improving operational cost-effectiveness. Both patterns support the hypothesis that context pruning serves not merely as an engineering optimization but as a cognitive strategy that enables more efficient problem-solving behaviors. By reducing the volume of irrelevant information presented to the agent at each step, pruning allows the underlying language model to allocate more attention to task-critical signals, thereby improving both decision quality and resource utilization.

## J Prompt Templates

This appendix presents the complete prompt templates used in our experiments. We provide four key prompt templates that form the core of our approach. First, the Silver Label Prompt is used to generate training data by asking models to answer queries using provided code context with explicit line citations. Second, the Mini SWE Agent with

Pruner Template defines the agent system for solving software engineering tasks, which includes detailed instructions on response format, context focus questions, workflow recommendations, and command execution rules. Third, the SWE-QA Bash Tool Descriptions specify how the bash execution tool works within the agent framework, particularly highlighting the usage of context focus questions for filtering large outputs. Finally, the Quality Evaluation Prompt provides the criteria and procedure for assessing the quality of pruned code samples across three dimensions: query quality, deletion relevance, and semantic preservation.

## Silver Label Prompt

```
You are given:
- a natural-language(or code for code completion task) Query
- a code snippet split into numbered lines (1>, 2>, 3>, ...)

Question: {query}
Code Context:
{code}

Answer the Question, using ONLY information provided in the Code Context. If no useful information
is provided, you MUST output "No answer". If some parts of the Context are used to answer, you
MUST cite ALL the corresponding lines.

Use the symbols [ ] to indicate when a fact comes from a line in the context, e.g [1] for a fact from line 1.
- For multi-line context, use [line1-line2], e.g. [12-25].
- For multi context, use [line1,line2,...], e.g. [1,3,5-7].

You should only answer the given question and should not provide any additional information

HINT:
- For code, context should be wider than `the line just answer the question`, for example, if the question is about a
      variable in a class method function, include the function definition, class definition and where it is used.
- When you try to cite something, its better to cite the structure of the code.
e.g. if you want to cite B1 in the code structure below:
```
1> if cond:
2> A1
3> A2
4> else:
5> B1
```
, best citation will be [1,4,5], which keeps the structure of the `if-else` block while removing the unrelated A1, A2.

Now give your answer with citations:
```

## Mini Swe Agent with pruner template

```
agent:
  system_template: |
    You are a helpful assistant that can interact multiple times with a computer shell to solve programming tasks.

    ## Response Format

    Your response must contain exactly ONE bash code block(with optinal context_focus_question shown below) with ONE
          command (or commands connected with && or ||).
    Include a THOUGHT section before your command where you explain your reasoning process.

    <format_example>
    THOUGHT: Your reasoning and analysis here. Explain why you want to perform the action.

    ```bash
    your_command_here
    ```
    <context_focus_question>
    Optional question to focus on relevant parts of the command output.
    </context_focus_question>
    </format_example>

    ## Context Focus Question (Optional)

    The `context_focus_question` is an optional field that helps filter large command outputs to show only relevant
          information.

    **Question requirements:**
    - question **MUST be a complete, self-contained question** (not keywords or phrases)
    - question should be specific enough to filter effectively
    - questions should **NOT** contain file-level info (e.g., filenames, line numbers), as the filter model only
          processes direct command output. For such info, use tools like grep/sed and leave the question blank.
    - question should be place it immediately after the bash code block (after the closing ```)

    ### Examples of a good context_focus_question:
    - Find where the [some logic] is implemented in the [some class/function]?
    - Given [some background], [some problem]?
    - Locate the [some logic]?
    - How can the code implement [some feature]?
    - [some combination of the above, like background + origin problem + current purpose + specific attention area]

    ### Examples of a bad context_focus_question:
    - load_raw function (too vague)
    - lines 50-100 of data_loader.py (contains file-level info)
    - fix the bug in rwkv6.py (too vague)

    Gives some context for the query is better for more effective filtering.
    Failure to follow these rules will cause your response to be rejected.
  instance_template: |
    <pr_description>
    Consider the following PR description:
    {{task}}
    </pr_description>

    <instructions>
    # Task Instructions

    ## Overview
    You're a software engineer interacting continuously with a computer by submitting commands.
    You'll be helping implement necessary changes to meet requirements in the PR description.
    Your task is specifically to make changes to non-test files in the current directory in order to fix the issue
          described in the PR description in a way that is general and consistent with the codebase.

    IMPORTANT: This is an interactive process where you will think and issue ONE command, see its result, then think and
          issue your next command.

    For each response:
    1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
    2. Provide exactly ONE bash command to execute
    3. Give an optional `context_focus_question` to filter the command output(or not give for output no need to be
          filtered)

    ## Important Boundaries
    - MODIFY: Regular source code files in /testbed (this is the working directory for all your subsequent commands)
    - DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg, etc.)

    ## Recommended Workflow
    1. Analyze the codebase by finding and reading relevant files to figure out why this issue happenes(with
          context_focus_question for large outputs like read a whole file)
    **HINT**: since we have pruner enabled, prefer reading files fully instead of using grep/find first
    **ATTENTION**: pruned code might miss some details, so its **highly recommend** to use `cat -n` or `nl -ba` command
          with context_focus_question, so you can see the line numbers. If you looked the filtered output and need more
          context, you can always use `sed` without context_focus_question to read more lines before/after the relevant
          lines.
```

## Mini Swe Agent with pruner template (Part 2)

```
2. When you have enough information, edit the source code to resolve the issue, you have only one chance, no tests,
      no retry, so read widely first(with context_focus_questions) before making changes.

## Command Execution Rules
You are operating in an environment where
1. You write a single command
2. The system executes that command in a subshell
3. You see the result(if you set a context_focus_question, the output will be filtered accordingly)
4. You write your next command

Each response should include:
1. A **THOUGHT** section where you explain your reasoning and plan
2. A single bash code block with your command
3. An optional `context_focus_question` to filter the command output, actually you can put some thoughts directly as
      background in it.

Format your responses like this:

<format_example>
THOUGHT: Here I explain my reasoning process, analysis of the current situation,
and what I'm trying to accomplish with the command below.

```bash
your_command_here
```
<context_focus_question>
Your question here.
</context_focus_question>
</format_example>

Commands must be specified in a single bash code block:

```bash
your_command_here
```

**CRITICAL REQUIREMENTS:**
- Your response SHOULD include a THOUGHT section explaining your reasoning
- Your response MUST include EXACTLY ONE bash code block(with optinal context_focus_question)
- This bash block MUST contain EXACTLY ONE command (or a set of commands connected with && or ||)
- If you include zero or multiple bash blocks, or no command at all, YOUR RESPONSE WILL FAIL
- Do NOT try to run multiple independent commands in separate blocks in one response
- Directory or environment variable changes are not persistent. Every action is executed in a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write/load
      environment variables from files

Example of a CORRECT response(no context_focus_question because `ls` will not output large content, and we want to
      see the full output):
<example_response>
THOUGHT: I need to understand the structure of the repository first. Let me check what files are in the current
      directory to get a better understanding of the codebase.

```bash
ls -la
```
</example_response>

Example of an INCORRECT response:
<example_response>
THOUGHT: I need to examine the codebase and then look at a specific file. I'll run multiple commands to do this.

```bash
ls -la
```

Now I'll read the file:

```bash
cat -n file.txt
```
<context_focus_question>
Some question about your purpose for reading the file.
</context_focus_question>

</example_response>

If you need to run multiple commands, either:
1. Combine them in one block using && or ||
```bash
command1 && command2 || echo "Error occurred"
```
```

## Mini Swe Agent with pruner template (Part 3)

```
  2. Wait for the first command to complete, see its output, then issue the next command in your following response.

  ## Environment Details
  - You have a full Linux shell environment
  - Always use non-interactive flags (-y, -f) for commands
  - Avoid interactive tools like vi, nano, or any that require user input
  - If a command isn't available, you can install it

  ## Useful Command Examples

  ### Create a new file:
  ```bash
  cat <<'EOF' > newfile.py
  import numpy as np
  hello = "world"
  print(hello)
  EOF
  ```

  ### Edit files with sed:
  ```bash
  # Replace all occurrences
  sed -i 's/old_string/new_string/g' filename.py

  # Replace only first occurrence
  sed -i 's/old_string/new_string/' filename.py

  # Replace first occurrence on line 1
  sed -i '1s/old_string/new_string/' filename.py

  # Replace all occurrences in lines 1-10
  sed -i '1,10s/old_string/new_string/g' filename.py
  ```

  ### View file content:
  ```bash
  # View specific lines with numbers
  nl -ba filename.py | sed -n '10,20p'
  <context_focus_question>
  If you know what line you want to view and the range of `sed` if large, you can also set a question here.
  </context_focus_question>
  ```

  ### Any other command you want to run
  ```bash
  anything
  ```

  ## Submission
  When you've completed your work (reading, editing, testing), and cannot make further progress
  issue exactly the following command:

  ```bash
  echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && git add -A && git diff --cached
  ```

  This command will submit your work.
  You cannot continue working (reading, editing, testing) in any way on this task after submitting.
  </instructions>
action_observation_template: |
  <returncode>{{output.returncode}}</returncode>
  {% if output.output | length < 10000 -%}
  <output>
  {{ output.output -}}
  </output>
  {%- else -%}
  <warning>
  The output of your last command was too long.
  Please try a different command that produces less output.
  If you have not used context_focus_question, use it.
  If you have used it but the output is still too long, do not try to change the question, just follow the instructions
         below while also use the context_focus_question.
  If you're looking at a file you can try use head, tail or sed to view a smaller number of lines selectively.
  If you're using grep or find and it produced too much output, you can use a more selective search pattern.
  If you really need to see something from the full command's output, you can redirect output to a file and then search
         in that file.
  </warning>
  {%- set elided_chars = output.output | length - 10000 -%}
  <output_head>
  {{ output.output[:5000] }}
  </output_head>
```

## Mini Swe Agent with pruner template (Part 4)

```
    <elided_chars>
    {{ elided_chars }} characters elided
    </elided_chars>
    <output_tail>
    {{ output.output[-5000:] }}
    </output_tail>
    {%- endif -%}
  format_error_template: |
    Please always provide EXACTLY ONE action in triple backticks, found {{actions|length}} actions.

    Please format your action in triple backticks as shown in <response_example>.

    <response_example>
    THOUGHT: Here are some thoughts about why you want to perform the action.

    ```bash
    <action>
    ```
    <context_focus_question>
    Please provide a question about the context you want to focus on. This part is optional. If you don't need to filter
          the output, you can leave it blank or omit it.
    </context_focus_question>
    </response_example>

    If you have completed your assignment, please consult the first message about how to
    submit your solution (you will not be able to continue working on this task after that).

    Note: A common trigger for this error is you put > 1 bash block(action) in your response.
  step_limit: 250
  cost_limit: 3.
  pruner:
    url: http://localhost:8000/prune
    timeout: 120
    retries: 3
    min_chars: 500
    chunk_overlap_tokens: 50
    threshold: 0.5

environment:
  cwd: "/testbed"
  timeout: 60
  env:
    PAGER: cat
    MANPAGER: cat
    LESS: -R
    PIP_PROGRESS_BAR: 'off'
    TQDM_DISABLE: '1'
  environment_class: docker
```

## SWE-QA bash tool descriptions

```
_BASH_DESCRIPTION = """Execute bash commands in the workspace.
* Use this tool to run shell commands, navigate directories, search files, etc.
* Commands are executed in the workspace directory.
* Output is returned as text.
"""

_BASH_DESCRIPTION_PRUNE = (
    _BASH_DESCRIPTION
    + """
The meaning of argument `context_focus_question` (Optional):

Use `context_focus_question` to filter large command outputs for relevant information.

**Requirements:**
- Must be a complete, self-contained question (not keywords/phrases)
- Be specific for effective filtering
- Don't include file-level info (filenames, line numbers) - use grep/sed instead

**Good examples:**
- Where is [some logic] implemented in [some class/function]?
- Given [background], what's the [problem]?
- How does the code implement [feature]?

**Bad examples:**
- load_raw function (too vague)
- lines 50-100 of data_loader.py (contains file info)
- fix bug in rwkv6.py (too vague)

**IMPORTANT:** With pruner enabled, prefer `cat -n` or `nl -ba` with context_focus_question to see line numbers. Then
      you can use `sed` without filtering for more detailed context since you have line number information.

**IMPORTANT:** If the command output is small and important like `ls`, just leave context_focus_question blank.
"""
)
```

## Quality Evaluation Prompt

```
You are a code quality evaluator for a code pruning dataset. Your task is to assess the quality of a query-guided code
      deletion sample.

You will be given:
1. **Query**: A code snippet for means making completion below or a natural language question/task
2. **Original Code**: Full code snippet with line numbers
3. **Diff**: Shows which lines were removed (- prefix) and kept (no change or + prefix)

Evaluate THREE dimensions:

## 1. Query Quality
- **Good**: Realistic, specific, actionable developer question related to a partial feature/function
- **Acceptable**: Valid but generic, or slightly unclear but answerable
- **Poor**: Too vague, treats code as the subject ("explain this code")

HINT: just focus on query itself, don't take query-code relevance into consideration in this part.

## 2. Deletion Relevance
- **Appropriate**: Removes truly unrelated code while keeping necessary context
- **Minimal**: Mostly removes whitespace/comments/trivial lines, little semantic pruning
- **Excessive**: Removes too much, including code relevant to the query

HINT: Both query-code high relevance and low relevance are ok, key point is the context preserved correctly. For high
      relevance, might more code; For low relevance, might less code.

## 3. Semantic Preservation
- **Preserved**: Remaining code is syntactically valid and semantically coherent (can understand the query-relevant logic
      )
- **Partially Preserved**: Minor issues (e.g., unmatched braces, missing imports that don't affect core logic
      understanding)
- **Broken**: Code is syntactically invalid or key logic is incomprehensible

## Overall Quality Rating
Based on the above three dimensions, assign:
- **high**: All three dimensions are good/appropriate/preserved, or at most one acceptable/partially_preserved
- **medium**: Two dimensions are good/acceptable/appropriate, one has issues; or all three are acceptable
- **low**: Two or more dimensions are poor/minimal/broken, or query is fundamentally flawed

---

### Input Data:

**Query:**
{query}

**Original Code (with line numbers):**
{code_with_numbers}

**Diff (deletions marked with -):**
```diff
{diff}
```

---

### Your Task:
1. Provide concise reasoning for each dimension (1 sentences per dimension)
2. Assign ratings: query_quality (good/acceptable/poor), deletion_relevance (appropriate/minimal/excessive),
      semantic_preservation (preserved/partially_preserved/broken)
3. Determine overall_quality (low/medium/high)

Output JSON format (no code fences, just JSON):
{{
  "reasoning": "<Brief analysis covering all three dimensions>",
  "query_quality": "<good|acceptable|poor>",
  "deletion_relevance": "<appropriate|minimal|excessive>",
  "semantic_preservation": "<preserved|partially_preserved|broken>",
  "overall_quality": "<low|medium|high>"
}}
```