# FASTCODE: FAST AND COST-EFFICIENT CODE UNDERSTANDING AND REASONING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Repository-scale code reasoning is a cornerstone of modern AI-assisted software engineering, enabling Large Language Models (LLMs) to handle complex workflows from program comprehension to complex debugging. However, balancing accuracy with context cost remains a significant bottleneck, as coding agents repeatedly process full code accumulated in their exploration history. To address this, we introduce FastCode, a framework that decouples repository exploration from content consumption. FastCode employs a structural scouting mechanism to navigate a lightweight semantic-structural map of the codebase, allowing the system to trace dependencies and pinpoint relevant targets without loading code bodies into the exploration context. By integrating structure-aware navigation with cost-aware context selection, the framework constructs high-value contexts for final reasoning while reducing repeated token overhead. Extensive evaluations on the SWE-QA, LongCodeQA, GitTaskBench, and LOC-BENCH benchmarks show that FastCode achieves favorable trade-offs between reasoning performance and token efficiency, highlighting its effectiveness for repository-scale code reasoning. Source code is available at `https://anonymous.4open.science/r/FastCode`.

## 1 INTRODUCTION

LLMs are increasingly embedded in real-world software engineering workflows, from answering developer questions to assisting with debugging, refactoring, and feature development (Wang et al., 2025a; Li et al., 2025). Many of these workflows fundamentally require *repository-scale code reasoning*: the ability to interpret a natural language request, identify the relevant parts of a large codebase, trace cross-file dependencies, and synthesize a grounded response or actionable plan (Peng et al., 2025; Deshpande et al., 2024). In modern engineering organizations, stronger repository-level understanding improves developer productivity and reduces costs by minimizing time navigating unfamiliar code and preventing avoidable regressions.

At the same time, repository-scale code reasoning is more challenging than snippet-level understanding. It must reconcile local implementation details with the global architecture, which often involves multi-hop dependency tracing across files, symbols, and abstraction layers (Chen et al., 2025; Liu et al., 2024; 2025). This difficulty surfaces in common and economically important tasks such as repository-level question answering (QA), code localization, and multi-hop comprehension for repair. In practice, *token and compute budgets* limit how much of the repository can be inspected per query, making it difficult to gather enough relevant evidence for grounded answers without pulling in excessive and potentially irrelevant context.

Despite rapid progress, existing approaches still face a core bottleneck: **the tension between accuracy and context cost**. General LLMs struggle to reliably answer repository-level questions without sufficient code context. Providing the entire repository is a straightforward solution, but incurs substantial token costs and can introduce irrelevant information. Standard retrieval-augmented generation (RAG) pipelines reduce this burden by selecting relevant code snippets, yet similarity-based retrieval may overlook dependencies needed for cross-file reasoning (Zhang et al., 2023). Systems such as DeepWiki (DeepWiki, 2025) and CodeWiki (Hurley et al., 2025) provide broader repository context through generated documentation. Building and maintaining these artifacts incurs upfront LLM costs, while their summaries may omit implementation details needed for the task.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Coding agents address gaps in the initial context by interleaving reasoning with repository exploration. Systems such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025b), Cursor (Anysphere, 2026), and Claude Code (Anthropic, 2026) use successive searches, code inspection, and tool feedback to investigate unresolved questions until they judge the context sufficient. However, when earlier reasoning and tool outputs are retained in subsequent LLM inputs, full code read during exploration is processed repeatedly. Since code content typically accounts for a substantial share of the context, token costs grow with both the amount of code inspected and the number of exploration rounds.

To balance reasoning accuracy with context cost, we introduce FastCode, a framework that decouples repository exploration from content consumption. Its structural scouting mechanism uses signatures, metadata, and dependency relations to iteratively identify task-relevant code without loading code bodies into the exploration context. FastCode loads the selected code together once for final reasoning, preserving iterative context gathering while avoiding repeated code processing during exploration. The framework comprises three components: (i) a semantic-structural representation that combines lightweight metadata with a multi-layered dependency graph; (ii) structure-guided navigation that iteratively identifies and refines relevant code candidates; and (iii) a cost-aware policy that regulates exploration and context assembly based on context sufficiency and budget constraints.

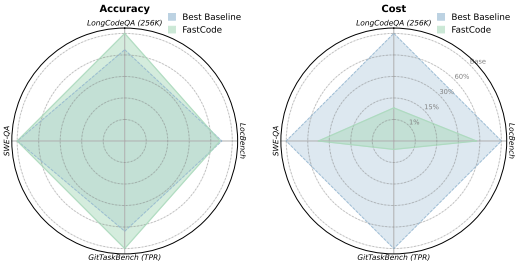


Figure 1: Accuracy and cost trade-off comparison

We evaluate FastCode on challenging repository-level benchmarks, including SWE-QA (Peng et al., 2025), LongCodeQA (Rando et al., 2025), GitTaskBench (Ni et al., 2025), and LOC-BENCH (Chen et al., 2025). Experimental results demonstrate that FastCode achieves favorable trade-offs between reasoning performance and token efficiency. The main contributions are:

- We introduce a framework for repository-scale code understanding and reasoning that decouples iterative exploration from full-code consumption. This separation preserves adaptive context gathering while keeping code bodies out of the exploration history, allowing the selected code to be consumed once for final reasoning and reducing repeated token overhead.
- We develop a structure-guided navigation strategy regulated by a cost-aware control policy. The strategy traces dependencies and intent cues over a lightweight semantic-structural map to identify relevant code, while the policy evaluates context sufficiency and the incremental value of further exploration to guide termination and final context assembly.
- Extensive experiments on four repository-level benchmarks validate the practical efficacy of the proposed framework. The results confirm that our framework improves repository-level reasoning accuracy and achieves substantial token savings, demonstrating the value of structural scouting for resource-constrained applications.

## 2 Related Work

### 2.1 Retrieval & Graph-Augmented Code Reasoning

Repository-scale code reasoning is fundamentally constrained by limited context budgets, motivating RAG as a standard approach for grounding model outputs in relevant code evidence. Early code RAG methods primarily relied on dense semantic retrieval to select code chunks similar to the query (Feng et al., 2020; Zhang et al., 2023). However, representing code as flat text can obscure critical structural signals (e.g., imports, call relations, and data/control dependencies), which limits multi-hop reasoning performance (Ding et al., 2023). To better preserve program structure, recent work incorporates graph- or dependency-aware representations for retrieval and context construction (Liu et al., 2025; 2024; Chen et al., 2025). For example, GraphCoder (Liu et al., 2024) leverages static structure to retrieve structurally related nodes (e.g., $k$-hop neighborhoods) to enrich the context, while LocAgent (Chen et al., 2025) combines graph indices with agentic navigation to iteratively localize relevant code.

Despite these advancements, it remains an open challenge to cost-efficiently pinpoint target contexts in complex repositories across diverse query scenarios.

### 2.2 Autonomous Coding Agent

Autonomous coding agents have emerged as a practical approach to repository-scale code reasoning, where a system must localize relevant evidence in a large codebase and execute grounded actions (e.g., debugging, or patching) to satisfy a natural language request (Jimenez et al., 2024). Representative frameworks such as MapCoder (Islam et al., 2024), CodeAgent (Zhang et al., 2024), SWE-agent (Yang et al., 2024), and OpenHands (Wang et al., 2025b), as well as commercial systems like Cursor (Anysphere, 2026) and Claude Code (Anthropic, 2026), typically follow an iterative "Observe–Think–Act" paradigm: the model repeatedly invokes environment tools (e.g., bash/Python, search, file open/edit, tests) to interact with the repository, progressively gather context, and perform operations. Despite their effectiveness in repository-level tasks, current agents suffer from high token consumption driven by the context accumulation of multi-turn interactions. In this work, we aim to bridge this efficiency gap by significantly reducing token usage while maintaining competitive reasoning accuracy.

## 3 Preliminary

### 3.1 Repository Representation

We formalize the target repository $\mathcal{R}$ as a **semantic map**, structured as a directed graph $\mathcal{G} = (\mathcal{U}, \mathcal{E})$. Here, $\mathcal{U}$ represents the discrete landscape of hierarchical code units (e.g., files, classes), while $\mathcal{E} = \{G_{\text{dep}}, G_{\text{inh}}, G_{\text{call}}\}$ denotes the topological pathways capturing dependencies, inheritance, and call flows. Crucially, each unit $u \in \mathcal{U}$ serves as a navigable landmark, comprising *lightweight metadata* (e.g., signatures) for rapid scouting and a *full implementation body* for deep reasoning. This dual representation enables the agent to explore the map and assess relevance without the computational overhead of traversing full code bodies.

### 3.2 Task Objective

Given a natural language query $q$ and the structured repository $\mathcal{G}$, our goal is to generate a solution $y$ (e.g., an answer or an executable patch) by navigating this map. Unlike static retrieval, we formulate the reasoning process as a joint optimization of relevance and efficiency. The system aims to identify an optimal context $\mathcal{C}^*$ that maximizes semantic sufficiency while penalizing token consumption:

$$y = \mathcal{M}(q, \mathcal{C}^*), \quad \mathcal{C}^* = \arg\max_{\mathcal{C} \subset \mathcal{U}} S(\mathcal{C}; q), \tag{1}$$

where $S(\mathcal{C}; q) = \text{Rel}(\mathcal{C} \mid q) - \lambda \Omega(\mathcal{C})$ measures the utility of a candidate context by balancing semantic relevance and context cost. $\mathcal{M}$ denotes the reasoning model that generates $y$ by processing the full content of $\mathcal{C}^*$. The optimization objective balances two competing goals: $\text{Rel}(\cdot)$ quantifies the relevance of the selected context $\mathcal{C}$, while $\Omega(\cdot)$ is the computational overhead, modulated by a penalty coefficient $\lambda$.

## 4 Methodology

To optimize the trade-off between reasoning accuracy and computational cost in repository-scale tasks, we propose FastCode, an agentic framework that reformulates context acquisition as a structure-guided navigation process. As illustrated in Figure 2, our approach consists of three components: (i) a *Code Semantic-Structural Representation* that unifies hybrid indexing with a multi-layered dependency graph; (ii) a *Codebase Context Navigation* mechanism that employs a scouting mechanism to explore repository layouts without incurring the cost of full-content reading; and (iii) a *Cost-Aware Context Management* policy that dynamically regulates information gathering to maximize reasoning confidence within a limited token budget.

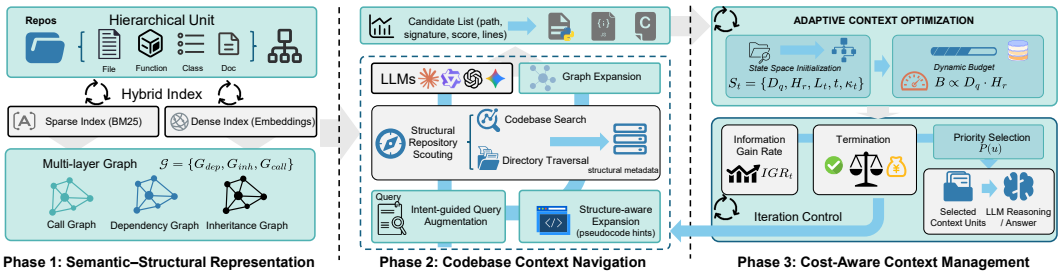



Figure 2: The overall framework of FastCode.

### 4.1 Semantic–Structural Code Representation

Standard RAG approaches treat code as flat textual chunks, severing intrinsic structural dependencies like inheritance and imports. To bridge this semantic gap, FastCode constructs a holistic representation that preserves the codebase's topological integrity. By decomposing the repository into interconnected semantic units rather than isolated segments, our framework enables precise navigation from global architecture to local logic through multi-grained hybrid indexing and symbol-aware relation modeling.

#### 4.1.1 Hierarchical Code Units

Optimal retrieval granularity varies with query intent: architectural reasoning demands file-level context, whereas debugging requires function-level specifics. To support this diversity, we parse the repository into a hierarchical tree spanning four levels: **File**, **Class**, **Function**, and **Documentation**. Crucially, instead of indexing only raw implementations, we extract lightweight metadata for each node—such as type signatures, docstrings and line ranges. This constructs a token-efficient skeletal view of the repository, enabling the subsequent agent to evaluate relevance without incurring the computational cost of reading full code bodies.

#### 4.1.2 Multi-Grained Hybrid Indexing

Code retrieval demands dual capabilities: pinpointing exact identifiers and understanding abstract intent. To reconcile these requirements, we implement a Dual-Index Mechanism that synergizes sparse and dense representations:

**(1) Sparse Index (Lexical):** Leveraging BM25, we target precise code tokens (e.g., function signatures) to preserve explicit keyword matches often diluted in vector spaces.

**(2) Dense Index (Semantic):** We employ an embedding model to map multi-grained units into high-dimensional vectors. This captures implicit semantics, identifying synonymous implementations even without lexical overlap.

#### 4.1.3 Symbol-Aware Relation Modeling

To mitigate context fragmentation across file boundaries, we construct a code relationship graph using efficient Abstract Syntax Tree (AST)-based analysis, avoiding the overhead of heavy compiler infrastructure. We formalize the repository as a multi-layered directed graph $\mathcal{G} = \{G_{dep}, G_{inh}, G_{call}\}$, capturing three distinct topological dimensions:

- **Dependency Layer** ($G_{dep}$)**:** Captures file-level imports as module dependencies.
- **Inheritance Layer** ($G_{inh}$)**:** Captures class hierarchies via base-class derivation.
- **Call Layer** ($G_{call}$)**:** Captures execution flow by linking invocation sites to definitions.

### 4.2 Codebase Context Navigation

Localizing query-related code elements is central to effective repository-scale reasoning, yet it is hindered by two challenges: the semantic ambiguity of natural language queries and the high token cost of iterative retrieval-and-reading loops. In standard agentic workflows, judging file relevance

often requires repeatedly loading file contents, which quickly inflates context consumption. To address this, we introduce a navigation phase that performs low-cost scouting over lightweight semantic-structural signals, pruning the search space before committing to full-text reading.

#### 4.2.1 QUERY AUGMENTATION

To improve repository navigation, we augment each query with intent-aware rewriting, keyword expansion, and pseudocode hints. The query is first categorized and rewritten for semantic retrieval, while domain terms, APIs, and symbol candidates are expanded to strengthen BM25-based lexical matching. This separates high-level concept matching from precise symbol localization. To capture algorithmic structure beyond terminology, we further use the LLM to generate pseudocode hints, including possible implementation skeletons, library calls, control flows, or class hierarchies. These hints serve as structural anchors for retrieving code with similar logic or API usage, improving recall when textual overlap is limited.

#### 4.2.2 STRUCTURAL REPOSITORY SCOUTING

To address the overhead of iteratively reading code bodies in existing coding agents, we propose a structural scouting strategy: the agent scores candidates using lightweight metadata, decoupling exploration from consumption and allocating context-window resources only to high-value targets.

**Tool-Assisted Exploration** We provide the agent with the following two code navigation tools:

- *Directory Traversal:* Lists files under a specified directory to identify relevant modules.
- *Codebase Search:* A regex-enhanced scanner that returns file path and match counts and without loading full files.

Tool call results are parsed into structural metadata (e.g., file paths, class/function signatures) rather than full implementation details. The agent uses these lightweight signals to filter candidates—prioritizing files with high match density and relevant signatures while discarding irrelevant ones.

**Adaptive Scouting Workflow** We structure the navigation as an iterative decision process over rounds $t \in \{1, \ldots, T\}$.

**Initialization (Round 1):** The system assesses the query and generates up to two streams:

- *Alignment Parameters:* Refined intent and keywords to guide the Semantic/Lexical Retriever.
- *Exploratory Tool Calls:* Optional execution commands for the exploration tools if the query requires specific pattern matching (e.g., finding usage of a specific API).

These streams are complementary: retrieval identifies similar files, while tool-based exploration handles pattern matching and structural navigation.

**Graph Expansion:** After deduplicating candidates from both streams, we apply graph extension to capture non-lexical dependencies: the system queries Call, Dependency, and Inheritance graphs to include logically related units, revealing structural relationships beyond initial matches.

**Iterative Refinement (Round $t$):** The merged candidates are presented to the agent as a consolidated metadata list. For each candidate unit (File, Class, or Function level), the system exposes a multi-dimensional profile, including the: **(i) Provenance:** Source of discovery (retrieval, tools call, or graph neighbor) with corresponding relevance scores. **(ii) Structural Identity:** Signatures, inheritance bases, docstrings, and relationships to existing candidates (for graph-expanded units). **(iii) Cost Metric:** Total line count. Based on this aggregated view, the agent decides which units to keep and whether to terminate scouting or issue new tool calls to resolve remaining uncertainty.

### 4.3 COST-AWARE CONTEXT MANAGEMENT

To balance accuracy and cost, FastCode treats context construction as a state-aware adaptive process. By monitoring the evolving reasoning state against the remaining budget, the framework dynamically regulates information gathering—deciding when to acquire details or terminate the search within resource limits. Computational definitions and default settings are provided in Appendix A.2.1.

#### 4.3.1 STATE SPACE INITIALIZATION

We formalize the iterative retrieval as a trajectory through a multi-dimensional state space. At each iteration step $t$, the agent observes a state vector $S_t = \{D_q, H_r, L_t, t, \kappa_t\}$, which synthesizes static environmental factors with dynamic runtime metrics to guide the policy $\pi(S_t)$:

**(1) Query Complexity ($D_q$):** Intrinsic difficulty score ($0 \leq D_q \leq 100$) estimated in the scouting phase. It quantifies reasoning depth (e.g., refactoring vs. symbol lookup) and serves as the primary scaler for the compute budget.

**(2) Repository Entropy ($H_r$):** Structural complexity factor ($0.5 \leq H_r \leq 2.0$) derived from topological analysis (e.g., file count, code depth). Larger values allow a wider exploration horizon for structurally complex repositories.

**(3) Resource Consumption ($L_t$):** Estimated code volume (in lines) represented by the candidate units at step $t$. This acts as a dynamic cost constraint to strictly enforce the adaptive line budget.

**(4) Iteration Depth ($t$):** Current recursive step count ($1 \leq t \leq T$). It serves as a hard horizon constraint to prevent infinite reasoning loops, overriding budget availability.

**(5) Epistemic Confidence ($\kappa_t$):** A self-assessed context-sufficiency score ($0 \leq \kappa_t \leq 100$) that estimates whether the candidate evidence covers the query. This signal guides the termination policy.

#### 4.3.2 ADAPTIVE CONTEXT OPTIMIZATION

Guided by the initialized state $S_t$, FastCode employs a policy $\pi(S_t)$ to orchestrate the unit selection trajectory. This policy balances estimated context sufficiency ($\kappa_t$) against exploration depth and code volume ($t$ and $L_t$) via the following mechanisms.

**Dynamic Budgeting & Iteration Control** The agent first establishes a resource boundary based on the environmental complexity. We assign a dynamic line budget $B$ using a piecewise rule based on $D_q$ and $H_r$: simple lookups receive smaller budgets, while complex queries receive a repository-scaled budget. The iteration flow is governed by the epistemic confidence $\kappa_t$:

- **Fast Path:** If initial scouting yields high confidence ($\kappa_0 \geq \tau$), the policy skips tool execution and query augmentation, using standard retrieval with graph extension before entering refinement. This preserves two-round verification while reducing latency.
- **Iterative Expansion:** If $\kappa_t < \tau$, the agent continues the retrieval loop. The policy monitors the *Information Gain Rate*, defined as $IGR_t = \frac{\kappa_t - \kappa_{t-1}}{L_t - L_{t-1}}$, which quantifies the confidence improvement per unit of context expansion. We accept a single-step confidence drop ($\Delta\kappa_t < 0$) as a corrective adjustment. Termination is triggered only after consecutive low-gain steps ($IGR_t < \epsilon$) or when the projected cost exceeds the budget.

Iteration stops upon (i) *Sufficiency:* $\kappa_t \geq \tau$; (ii) *Inefficiency:* Consecutive low-gain rounds; or (iii) *Exhaustion:* The budget or iteration limit is reached, or further expansion would exceed the budget.

**Budget-Constrained Context Selection** To keep the final context within the line budget $B$, FastCode uses a simple safeguard that prioritizes candidate code units by retrieval relevance, explicit tool evidence, and granularity. High-confidence snippets are retained before larger or weakly supported files until the budget is reached, reducing context overflow while preserving key evidence.

## 5 EVALUATION

We evaluate FastCode on repository-scale code reasoning tasks through six research questions: **RQ1:** How does FastCode perform on repo-level QA tasks? **RQ2:** How capable is FastCode in performing end-to-end execution of diverse repository tasks? **RQ3:** How effective is FastCode at localizing relevant code in large-scale repositories? (Appendix A.3.1) **RQ4:** How efficient is FastCode in terms of token usage? **RQ5:** Which components contribute most to FastCode's performance? **RQ6:** How does performance vary across different LLM?

Table 1: Comparison of baseline performance on the SWE-QA dataset (3-repository subset).

| Metric | Community Baselines | | | | Commercial Tools | | | | | Ours |
|---|---|---|---|---|---|---|---|---|---|---|
| | Direct LLM | Func RAG | Sliding RAG | SWEQA-Agent | DeepWiki | CodeWiki | Gemini Code | Claude Code | Cursor | **FastCode** |
| Correctness | 7.36 | 7.96 | 8.06 | 8.36 | 8.02 | 7.89 | 8.03 | 8.31 | 8.30 | 8.44 |
| Completeness | 3.71 | 5.75 | 5.99 | 6.94 | 5.90 | 6.49 | 6.09 | 7.25 | 8.13 | 7.94 |
| Clarity | 7.38 | 8.97 | 9.01 | 9.10 | 8.73 | 8.75 | 8.97 | 8.76 | 8.71 | 8.84 |
| Relevance | 8.94 | 9.56 | 9.65 | 9.72 | 9.36 | 9.21 | 9.68 | 9.56 | 9.39 | 9.35 |
| Reasoning | 4.91 | 7.38 | 7.51 | 8.22 | 7.52 | 7.82 | 7.92 | 8.21 | 8.64 | 8.72 |
| **Total Score** | 32.30 | 39.62 | 40.22 | 42.33 | 39.54 | 40.15 | 40.69 | 42.08 | <u>43.17</u> | **43.28** |

### 5.1 Experimental Setup

**Benchmarks** We conduct a comprehensive evaluation of FastCode across three distinct capabilities: repository-level question answering, end-to-end tasks, and file localization. Specifically, we utilize SWE-QA (Peng et al., 2025) and LongCodeQA (Rando et al., 2025) to assess cross-file reasoning and long-context comprehension; GitTaskBench (Ni et al., 2025) to validate execution-based performance on complex, actionable repository tasks; and LOC-BENCH (SWE-Bench-Lite subset) (Chen et al., 2025) to measure fault localization accuracy in real-world maintenance scenarios. Settings and detailed localization results are provided in Appendices A.2 and A.3.1, respectively.

**Baselines & Settings** The baselines and experimental settings are summarized as follows:

**(1) Direct LLM.** We evaluate standard LLMs in two settings: (i) *Zero-Shot* (for SWE-QA), where the model is prompted without any external repository context; and (ii) *Full-Context* (for LongCodeQA), where the entire repository content is fed into the model's long-context window.

**(2) RAG-based Methods.** We compare RAG baselines with different retrieval granularities: *Func RAG* retrieves function-level semantic chunks; *Sliding RAG* retrieves overlapping code spans via a sliding window; and *File BM25 RAG* retrieves relevant files using sparse keyword matching scores.

**(3) Agent & Procedure-based Methods.** SWEQA-Agent (Peng et al., 2025) is a ReAct-style tool-using agent that iteratively searches, navigates, and reads the repository to synthesize grounded answers. Agentless (Xia et al., 2025) is a lightweight, procedure-based LLM pipeline that follows a structured workflow for repository-level reasoning. MoatlessTools (Örwall, 2025) combines tool-driven exploration with semantic retrieval to search and gather code context. SWE-agent (Yang et al., 2024) provides an agent-computer interface for repository search, file inspection, and command execution. OpenHands (Wang et al., 2025b) is a generalist software-engineering agent platform supporting shell and file operations with iterative refinement. LocAgent (Chen et al., 2025) builds a heterogeneous repository graph and uses unified tools for graph-guided code localization.

**(4) Commercial Tools.** We evaluate commercial repository assistants for documentation and interactive code understanding. DeepWiki (DeepWiki, 2025) and CodeWiki (Hurley et al., 2025) ingest repositories and generate structured wiki-style documentation, including project overviews, modules, APIs, and usage notes. Gemini Code (Google, 2025) supports repository import and interactive natural-language Q&A about code structure and implementation details. Claude Code (Anthropic, 2026) is a proprietary coding assistant for code understanding, editing, refactoring, and tool-assisted task completion. Cursor (Anysphere, 2026) integrates project-wide context, chat-based assistance, and in-editor code generation and editing into an IDE.

### 5.2 Performance on Repo-QA Tasks (RQ1)

We evaluate FastCode's effectiveness on repository-scale QA tasks by comparing it against direct LLM usage, RAG & agent-based methods, and commercial tools. Specifically, community baselines and FastCode use Gemini-3-Flash as the backbone model, while commercial tools are kept in default mode. Table 1 and Table 2 present results on SWE-QA and LongCodeQA, respectively.

**Comparison with Direct LLM. (i)** *On SWE-QA*, FastCode substantially outperforms Direct LLM (Total 43.28 vs. 32.30), with the largest gains in Completeness (7.94 vs. 3.71) and Reasoning (8.72 vs. 4.91). This shows that repo QA benefits from grounded evidence: FastCode can locate relevant code context instead of relying on parametric knowledge. **(ii)** *On LongCodeQA*, FastCode improves accuracy in most matched settings, especially at longer contexts, by focusing reasoning on relevant evidence. In isolated cases, context selection may omit useful details and reduce accuracy.

Table 2: Accuracy on LongCodeQA.

| Model | 32K | 64K | 128K | 256K | 512K | 1M |
|---|---|---|---|---|---|---|
| Jamba 1.5 400B Large | 69.0 | 69.7 | 72.8 | 54.2 | - | - |
| Llama 3.1 405B Ins. | 69.9 | 72.4 | 67.4 | - | - | - |
| Llama 4 Scout | 66.4 | 73.7 | 70.7 | 63.1 | 78.7 | 76.0 |
| GPT-4o | 65.5 | 76.3 | 74.3 | - | - | - |
| GPT-4.1 | 72.6 | 73.7 | 78.3 | 72.3 | 78.7 | 80.0 |
| Qwen2.5-14B Ins. | 61.9 | **65.8** | 68.5 | 63.1 | 70.2 | 40.0 |
| + **FastCode** | **63.7** | 61.8 | **73.3** | **63.1** | **72.3** | **72.0** |
| Gemini 2.5 Pro | 75.2 | 71.1 | 71.7 | 67.7 | 68.1 | 69.8 |
| + **FastCode** | **77.9** | **79.0** | **75.0** | **75.4** | **80.9** | **80.0** |
| Claude 3.5 Sonnet | 65.5 | 69.7 | 71.7 | 66.6 | - | - |
| + File BM25 RAG | 25.55 | 31.18 | 24.83 | 25.95 | 17.19 | 12.77 |
| + **FastCode** | **69.9** | **69.7** | **72.83** | **78.5** | **76.6** | **78.0** |

**Comparison with RAG and Agentic Baselines.** FastCode achieves the best overall performance among retrieval-based and agentic baselines on SWE-QA, scoring 43.28 compared with Func RAG (39.62), Sliding RAG (40.22), and SWEQA-Agent (42.33). The gain is particularly clear in Completeness (7.94 vs. 6.94), suggesting that hierarchical indexing helps preserve code semantics at multiple granularities, while graph-guided expansion provides explicit structural links beyond chunk-level retrieval and purely tool-based exploration. On LongCodeQA, we find that naive file retrieval can be harmful: Claude 3.5 Sonnet with File BM25 RAG drops sharply from 65.5 to 25.55 in 32K subset, likely because lexical, file-level matching both misses transitive dependencies and retrieves large, low-signal files that crowd out key evidence. In contrast, FastCode maintains strong performance, reflecting more precise, dependency-complete context construction. By reducing fragmentation and noise, FastCode consistently outperforms RAG and agents under a fixed budget.

**Comparison with Commercial Tools.** FastCode surpasses commercial systems spanning wiki-style summarization tools (DeepWiki/CodeWiki), imported-repo Q&A (Gemini Code), and agentic assistants (Claude Code, Cursor). We posit that wiki-style tools emphasize broad documentation coverage rather than execution- and symbol-level grounding, while interactive assistants may still overconsume context during exploration. Agentic assistants benefit from strong interaction loops, which explains Claude Code and Cursor's competitive performance. Yet our framework demonstrates that competitive performance can also be achieved through alternative means. These findings demonstrate that the proposed framework can precisely locate target context by relying solely on lightweight metadata, thereby matching or exceeding commercial-grade systems.

### 5.3 End-to-End Repository Tasks (RQ2)

We evaluate FastCode on GitTaskBench to assess its capability in handling complex, real-world repository maintenance scenarios. As shown in the Figure 3, FastCode consistently achieves state-of-the-art performance across varying LLM backbones. Powered by Claude 3.5 Sonnet, our framework attains a Task Pass Rate (TPR) of 46.30%, surpassing strong baselines like OpenHands (40.74%) and SWE-Agent (22.23%). This performance gap widens with the stronger Claude 3.7 Sonnet model, where FastCode reaches a peak TPR of 57.41% and an Execution Completion Rate (ECR) of 74.07%. Notably, FastCode remains effective under a lightweight backbone: with Gemini-3-Flash, it achieves a TPR of 53.70%, higher than the OpenHands result reported with Claude 3.7 Sonnet. Our navigation and context management mechanisms decouple exploration from consumption: lightweight scouting prunes the search space, while dynamic token budgeting lets FastCode construct precise context to maximize reasoning accuracy within resource constraints.

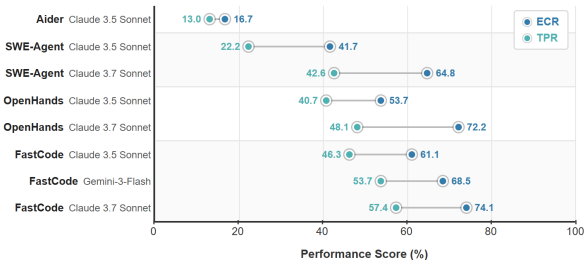
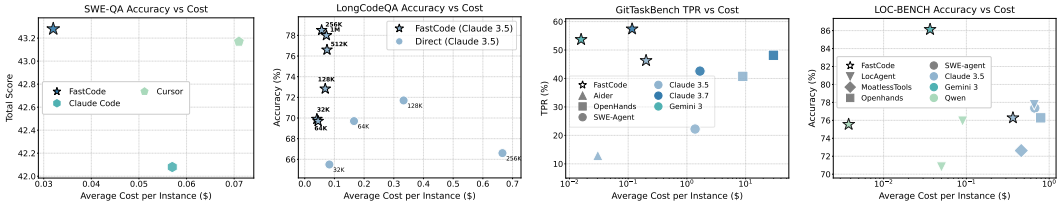



Figure 3: Baseline performance on GitTaskBench.




(a) SWE-QA (b) LongCodeQA (c) GitTaskBench (d) LOC-BENCH

Figure 4: Performance–cost trade-offs across four benchmarks (performance ↑, cost ↓).

### 5.4 Cost-Performance Trade-off (RQ4)

Figure 4 summarizes FastCode's performance–cost trade-offs across four benchmarks. On SWE-QA (Figure 4a), FastCode scores 43.28 at an average cost of $0.032, outperforming Claude Code and Cursor while reducing cost by approximately 55% relative to Cursor. On LongCodeQA (Figure 4b), costs remain nearly stable as context length grows; at 256K tokens, FastCode reduces cost by 91.4% while improving accuracy by 17.9%. On GitTaskBench (Figure 4c), FastCode with Claude 3.7 Sonnet reaches 57.41% TPR at approximately $0.102 per task, compared with 48.15% at $29.8 for OpenHands with the same backbone. On LOC-BENCH (Figure 4d), using Claude 3.5 Sonnet for both methods, FastCode achieves 76.28% Acc@1 at $0.365 per instance versus LocAgent's 77.74% at $0.660, yielding a 44.7% reduction in reported cost with competitive accuracy. Appendix A.3.1 provides detailed benchmark-level comparisons.

### 5.5 Ablation Study (RQ5)

To validate the contribution of each component, we conduct ablation experiments on a three-repository subset of SWE-QA using Qwen3-Coder-30b-A3b. As shown in Figure 5, we evaluate five variants that progressively remove core mechanisms from the full framework. Two key findings emerge:

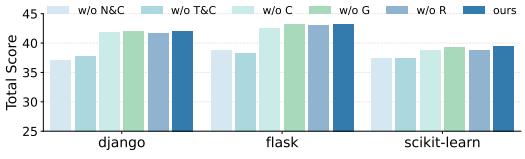



Figure 5: Ablation results on SWE-QA.

**Individual components provide complementary value.** The w/o R variant, which removes hybrid retrieval, drops by 0.60%, showing that semantic-lexical matching remains necessary for efficient initial candidate identification even with tool-based exploration. Removing graph extension (w/o G) leads to a 0.83% drop, suggesting that call, inheritance, and dependency relations capture non-lexical links missed by retrieval alone. When removing Cost-Aware Context Management (w/o C), the system loses dynamic budget allocation and epistemic confidence monitoring, resulting in a -0.92% decline as the agent fails to balance exploration depth against resource constraints.

**Navigation mechanisms are critical.** The w/o N&C variant, which removes both query augmentation and tool-based exploration (also w/o C), exhibits the most significant performance degradation (-5.06% average), demonstrating the fundamental importance of bridging the semantic gap between natural language queries and code structure. The w/o T&C variant, which retains query augmentation but eliminates tool-based scouting, shows a smaller yet substantial drop (-4.31% average). This gap confirms that while query refinement establishes semantic alignment, structural exploration through lightweight metadata is essential for efficient candidate pruning in large repositories.

## 6 Conclusion

We introduced FastCode, a framework that resolves the tension between reasoning accuracy and computational cost by decoupling repository exploration from content consumption. Through a scouting-first paradigm over a semantic-structural map, FastCode isolates high-value evidence and consolidates it into a one-shot reasoning context, avoiding the overhead of iterative full-text ingestion. Extensive evaluations across four benchmarks demonstrate that FastCode achieves favorable trade-offs between reasoning performance and token efficiency. These results validate that FastCode offers a scalable and resource-efficient solution for complex, repository-scale software engineering tasks.

## AI USE STATEMENT

We used generative AI tools to polish the language and improve the readability of the manuscript, and to assist with LaTeX formatting and the layout of text, tables, and figures. The authors are responsible for reviewing and verifying all AI-assisted revisions to ensure the accuracy and integrity of the final manuscript.

## REFERENCES

Anthropic. Claude code: Ai coding agent for terminal & IDE. https://claude.com/product/claude-code, 2026. Accessed: 2026-01-25.

Anysphere. Cursor. https://cursor.com/, 2026. Accessed: 2026-01-25.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 8697–8727, 2025.

DeepWiki. Deepwiki: Ai documentation you can talk to, for every repo. https://deepwiki.org/, 2025. Accessed: 2026-01-25.

Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. Class-level code generation from natural language using iterative, tool-enhanced reasoning over repository, 2024. URL https://arxiv.org/abs/2405.01573.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems*, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Google. Upload code folders or import your github repositories directly into the gemini app. https://workspaceupdates.googleblog.com/2025/06/upload-code-folders-and-github-repositories-to-gemini.html, 2025. Accessed: 2026-01-28.

Fergus Hurley, Pedro Rodriguez, Rafael Marques, and Omar Shams. Introducing code wiki: Accelerating your code understanding. Google Developers Blog, 2025. URL https://developers.googleblog.com/en/introducing-code-wiki-accelerating-your-code-understanding/. Accessed: 2026-01-25.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 4912–4944, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. Deepcode: Open agentic coding, 2025. URL https://arxiv.org/abs/2512.07921.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 570–581, 2024.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160, 2025.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, Sen Hu, Ronghao Chen, Bo Li, Xin Li, Chen Hu, Binxing Jiao, Daxin Jiang, and Pin Lyu. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging, 2025. URL https://arxiv.org/abs/2508.18993.

Albert Örwall. Moatless tools. https://github.com/aorwall/moatless-tools, 2025. GitHub repository. Accessed: 2026-01-28.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. Swe-qa: Can language models answer repository-level code questions?, 2025. URL https://arxiv.org/abs/2509.14635.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. Longcodebench: Evaluating coding llms at 1m context windows, 2025. URL https://arxiv.org/abs/2505.07897.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3742–3761, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, pp. 65882–65919, 2025b.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, 2024.

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

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 13643–13658, 2024.

# A APPENDIX

This appendix provides supplementary implementation details and extended experimental analysis to support the findings presented in the main text. The content is organized as follows. Section A.1 details the algorithms and construction process for Symbol-Aware Relation Modeling. Section A.2 outlines the experimental setup, including specific hyperparameters, baseline configurations, and metric definitions. Section A.3 presents additional experiments and analysis.

## A.1 DETAILS OF SYMBOL-AWARE RELATION MODELING

To instantiate $\mathcal{G}$ with high precision, we employ a two-stage resolution mechanism that operates on the static syntax trees. First, for structural connectivity ($G_{dep}$ and $G_{inh}$), we implement Global Symbol Resolution, which utilizes a deterministic module map to resolve dynamic import patterns—including aliased namespaces (e.g., `import numpy as np`) and relative paths—to unique definition IDs. Second, for execution logic ($G_{call}$), we mitigate Python's dynamic typing ambiguity via Scope-Constrained Type Inference. By tracking variable assignments within specific AST scopes (e.g., constructor initialization), our system infers instance types to link method calls (e.g., `self.loader.load()`) directly to their target definitions.

## A.2 EXPERIMENTAL SETUP

**Benchmark** We evaluate our FastCode framework on the following three classes of repository-centric benchmarks:

**(1) Repository-level Code QA.** SWE-QA (Peng et al., 2025) is a repository-level question answering benchmark with 576 human-validated QA pairs from 12 widely used open-source Python repositories, targeting holistic codebase understanding that requires cross-file, multi-hop reasoning (e.g., intent inference, dependency tracing, and feature localization). LongCodeQA (Rando et al., 2025), a subset of LongCodeBench, complements this setting with multiple-choice questions derived from real GitHub issue discussions, emphasizing long-context repository comprehension under realistic problem statements. It contains 443 instances spanning input token lengths from 32K to 1M tokens.

**(2) End-to-End Repository Tasks.** GitTaskBench (Ni et al., 2025) evaluates agentic performance on end-to-end tasks that require setting up environments, navigating existing GitHub codebases, and producing runnable outputs under practical success criteria. It contains 54 human-curated tasks across multiple modalities/domains, each paired with an automated evaluation harness (e.g., execution-based completion and pass rate) and cost-aware metrics to quantify effectiveness under realistic resource constraints.

**(3) File Localization.** We further evaluate on SWE-Bench-Lite from LOC-BENCH (Chen et al., 2025), which measures a model's ability to localize the relevant files and code regions for resolving real-world issues. Built from 274 issue–patch pairs in popular Python projects, this benchmark stresses accurate fault/feature localization as a prerequisite to downstream patch generation, covering diverse maintenance scenarios such as bug fixes and feature updates.

**Configurations** For SWE-QA, we evaluate community baselines and FastCode with Gemini-3-Flash. For DeepWiki and CodeWiki, we obtain answers via their default interaction workflow. Additionally, we use GPT-5.2 as the evaluation model to assess the performance of all baselines and our method on SWE-QA. For Gemini Code and Claude Code, we use Gemini-3-Pro and Claude Sonnet 4.5 as the backbone models, respectively, while Cursor is run in Auto mode to select the model automatically. For the remaining benchmarks, FastCode receives the repository associated with each task. For GitTaskBench, we prompt FastCode to generate an executable .sh script and run it using a predefined automation script, and the benchmark scores the run against a unified success criterion.

**Cost Calculation** We primarily employ Poe[1] and OpenRouter[2] as API endpoints. Specifically, we access Qwen3-Coder-30B via OpenRouter, while utilizing Poe for all other models. Costs are calculated based on the official pricing (*i.e.*, cost per 1M tokens). Detailed pricing is listed in Table 3. For baselines, we directly adopt the cost figures reported in their original publications (Chen et al., 2025) or relevant benchmark papers (Rando et al., 2025; Ni et al., 2025).

Table 3: Detailed API usage costs. Prices are denoted in USD per 1M tokens.

| Model | Provider | Price ($ / 1M) | |
|---|---|---|---|
| | | Input | Output |
| Gemini-3-Flash | Poe | 0.40 | 2.40 |
| Gemini-2.5-Pro | Poe | 0.87 | 7.00 |
| Claude 3.5 Sonnet | Poe | 2.60 | 13.00 |
| Claude 3.7 Sonnet | Poe | 2.60 | 13.00 |
| Qwen3-Coder-30B | OpenRouter | 0.07 | 0.27 |

**Metrics** For the four benchmarks, we follow the official evaluation metrics established for each task: (i) *SWE-QA*: We evaluate models with an LLM-as-judge rubric that scores each prediction against the reference answer on five dimensions—correctness, completeness, relevance, clarity, and reasoning quality—with a maximum score of 10 for each. (ii) *LongCodeQA*: Performance is evaluated via multiple-choice accuracy (1-of-4 options) across varying context-length intervals. (iii) *GitTaskBench*: We evaluate performance using Execution Completion Rate (ECR), checking for the generation of valid and non-empty outputs, and Task Pass Rate (TPR), which verifies if the outputs satisfy specific quality thresholds and functional requirements tailored to each task. (iv) *LOC-BENCH (SWE-Bench-Lite)*: We report Acc@$k$ ($k \in \{1, 3, 5\}$), where correctness requires all ground-truth files to be present in the top-$k$ candidates.

**Artifact Usage.** All benchmarks, baseline systems, and model APIs are used only for research evaluation following their intended experimental purposes. We cite the original creators of all used artifacts and follow their released licenses or usage terms where applicable.

### A.2.1 Cost-Aware Policy Settings

Query complexity $D_q \in [0, 100]$ is estimated by the scouting LLM using a fixed rubric ranging from symbol lookup to system-wide debugging. The context-sufficiency score $\kappa_t \in [0, 100]$ reflects query coverage and symbol/dependency completeness, rather than a calibrated correctness probability. Repository complexity $H_r \in [0.5, 2.0]$ is computed from file count $F$, mean file length $A$, and maximum directory depth $D$. Writing $c(x; a, b) = \min\{b, \max\{a, x\}\}$, we use

$$\begin{aligned} H_r &= c((h_F + h_A + h_D)/3; 0.5, 2.0)\,, \\ h_F &= c\left(\frac{\log_{10}(F+1)}{3}; 0.3, 1.5\right), \quad h_A = c(A/200; 0.5, 2.0), \quad h_D = c(D/5; 0.7, 1.3). \end{aligned} \tag{2}$$

The base line budget is $B_0 = 12{,}000$: $B = 0.6B_0$ for $D_q \le 30$, $B = 0.8B_0$ for $30 < D_q \le 60$, and $B = B_0 H_r$ otherwise. The iteration limit scales with $D_q$ and $H_r$ from $T_0 = 4$ and is clipped to $[2, 6]$. The default base sufficiency threshold is $\tau_0 = 95$, with the effective threshold $\tau$ reduced to 92 and 90 for $D_q \ge 60$ and $D_q \ge 80$, respectively. Marginal-gain checks use a minimum gain of 5 points and a base rate of 2 points per 1,000 added lines, adjusted by $D_q$ and $\kappa_t$. Two consecutive low-gain rounds trigger stopping; budget and iteration limits provide additional stopping conditions.

## A.3 Additional Experiment

### A.3.1 Detail of Cost-Performance Trade-off

This section supplements the four-benchmark comparison in Figure 4. Tables 4, 5, 6, and 7 report the underlying scores, costs, and relative improvements. The analysis below examines how these trade-offs vary with context length, backbone model, and task type.

[1] https://poe.com/api

[2] https://openrouter.ai/

Table 4: SWE-QA performance comparison

| Method | Total Score | Cost |
|---|---|---|
| Claude Code | 42.08 | 0.057 |
| Cursor | 43.17 | 0.071 |
| **FastCode** | **43.28** | **0.032** |
| *Improvement over Claude Code* | +2.85% | -43.86% |

Table 5: LongCodeQA performance: direct vs. FastCode-enhanced

| Context Length | Accuracy (%) | | Cost | | Acc. Gain |
|---|---|---|---|---|---|
| | Direct | +FastCode | Direct | +FastCode | |
| 32K | 65.5 | **69.9** | 0.083 | 0.041 | +6.72% |
| 64K | 69.7 | 69.7 | 0.166 | 0.045 | +0.00% |
| 128K | 71.7 | **72.83** | 0.333 | 0.069 | +1.58% |
| 256K | 66.6 | **78.5** | 0.666 | 0.057 | +17.87% |
| 512K | - | **76.6** | 1.331 | 0.075 | - |
| 1M | - | **78.0** | 2.600 | 0.073 | - |
| *Average Cost Reduction: **-83.01%** (for comparable lengths)* | | | | | |

**(1) Repo QA Efficiency.** On SWE-QA, FastCode achieves a total score of 43.28 with an average cost of $0.032, outperforming both Claude Code (42.08, $0.057) and Cursor (43.17, $0.071). This represents an approximately **55% cost reduction** compared with Cursor while maintaining high accuracy, validating that metadata-driven scouting can prune irrelevant code before expensive content reading. The efficiency advantage becomes more pronounced as repository size increases. On LongCodeQA, direct input approaches suffer from rapidly growing context costs, with Claude 3.5 Sonnet escalating from $0.083 at 32K tokens to $2.600 at 1M tokens. In contrast, FastCode maintains near-constant costs ($0.041–$0.073) across all scales. At the 256K context length, our approach delivers a **91.4% cost reduction** with a **17.9% accuracy improvement**, demonstrating that selective context construction effectively mitigates the context tax of long-context models.

Table 6: GitTaskBench performance comparison

| Method | ECR (%) | TPR (%) | Cost |
|---|---|---|---|
| Aider (Claude 3.5) | 16.67 | 12.96 | 0.0304 |
| SWE-Agent (Claude 3.5) | 41.67 | 22.23 | 1.38 |
| SWE-Agent (Claude 3.7) | 64.81 | 42.59 | 1.67 |
| OpenHands (Claude 3.5) | 53.70 | 40.74 | 8.95 |
| OpenHands (Claude 3.7) | 72.22 | 48.15 | 29.8 |
| FastCode (Claude 3.5) | 61.11 | 46.30 | 0.0878 |
| FastCode (Claude 3.7) | **74.07** | **57.41** | 0.1015 |
| FastCode (Gemini-3-Flash) | 68.52 | 53.70 | **0.0126** |

| Comparison | ECR Improvement | Cost Reduction |
|---|---|---|
| FastCode (3.7) vs. OpenHands (3.7) | +2.56% | -99.66% |
| FastCode (3.5) vs. OpenHands (3.5) | +13.80% | -99.02% |

**(2) End-to-End Task Efficiency.** With Claude 3.5 Sonnet for all methods, FastCode achieves 61.11% ECR and 46.30% TPR at $0.0878 per task, reducing reported cost by **93.6%** relative to SWE-Agent ($1.38) and **99.0%** relative to OpenHands ($8.95), while exceeding both baselines in ECR and TPR. With Claude 3.7 Sonnet for all methods, FastCode reaches 74.07% ECR and 57.41% TPR at $0.1015, again improving both metrics while reducing cost by **93.9%** and **99.7%** relative to SWE-Agent ($1.67) and OpenHands ($29.8), respectively.

Table 7: LOC-BENCH (SWE-Bench-Lite) performance.

| Method | Acc@1 (%) | Acc@3 (%) | Acc@5 (%) | Cost |
|---|---|---|---|---|
| *Existing Tools (Claude 3.5 Sonnet)* | | | | |
| MoatlessTools | 72.63 | 85.77 | 86.13 | 0.46 |
| SWE-agent | 77.37 | 87.23 | 90.15 | 0.67 |
| Openhands | 76.28 | 89.78 | 90.15 | 0.79 |
| LocAgent | 77.74 | 91.97 | 94.16 | 0.66 |
| *LocAgent (Fine-tuned Qwen)* | | | | |
| Qwen2.5-7B (ft) | 70.80 | 84.67 | 88.32 | 0.05 |
| Qwen2.5-32B (ft) | 75.91 | 90.51 | 92.70 | 0.09 |
| *FastCode Variants* | | | | |
| Qwen3-coder-30B | 75.55 | 87.23 | 88.32 | **0.0038** |
| Claude 3.5 Sonnet | 76.28 | 86.86 | 90.51 | 0.3650 |
| **Gemini-3-Flash** | **86.13** | **94.16** | **95.62** | 0.0364 |

**(3) File Localization Efficiency (RQ3).** Table 7 reports localization results on the SWE-Bench-Lite subset of LOC-BENCH. Using Claude 3.5 Sonnet for both methods, FastCode achieves 76.28% Acc@1 at $0.365 per instance, compared with LocAgent's 77.74% at $0.660. This reduces reported cost by **44.7%** while maintaining competitive localization accuracy. Additional FastCode configurations achieve 86.13% Acc@1 with Gemini-3-Flash and 75.55% with Qwen3-Coder-30B-A3B, providing options across proprietary and open-weight backbones.

### A.3.2 Evaluation Across LLM Backbones (RQ6)

To assess the universality of our framework, we evaluate FastCode across a spectrum of LLMs ranging from lightweight edge models (e.g., 3B parameters) to frontier cloud models. As detailed in Table 8, FastCode consistently yields significant performance improvements over both the *Direct LLM* and *SWEQA-Agent* baselines, demonstrating its robustness and model-agnostic nature.

Table 8: FastCode performance on SWE-QA under different LLMs.

| Base Model | Direct LLM | SWEQA-Agent | FastCode |
|---|---|---|---|
| Ministral-3b | 27.39 | 33.45 | **36.13** |
| Deepseek-r1-8b | 30.11 | 34.46 | **38.56** |
| Qwen3-Coder-30b-A3b | 35.77 | 37.08 | **41.44** |
| Gemini-3-Flash | 32.65 | 42.47 | **43.44** |

**Empowering Small Language Models.** A key finding is our framework's ability to bridge the capability gap for smaller models. By providing high-precision, structure-aware context, FastCode enables the *Ministral-3b* model to achieve a score of 36.13, effectively outperforming the much larger *Qwen3-Coder-30b-A3b* in the Direct setting (35.77). This suggests that superior context construction can compensate for lower parameter counts, proving that FastCode is highly effective for resource-constrained, on-device repository reasoning where massive models are deployed.

**Superiority Over Standard Agents.** While the *SWEQA-Agent* baseline improves performance via iterative tool use, it lacks the global navigation map and adaptive scouting mechanism central to our approach, which allow the model to systematically locate relevant contexts. This is particularly evident with *Qwen3-Coder-30b-A3b*: while the standard agent yields only a marginal gain (+1.31) over the direct baseline, FastCode delivers a substantial boost (+5.67). This indicates that even for models already fine-tuned for code, the bottleneck remains context quality rather than reasoning capability. FastCode unlocks the full potential of these strong backbones by replacing noisy, unstructured retrieval with precise, graph-guided navigation, ensuring consistent state-of-the-art performance across all model sizes.

### A.3.3 THRESHOLD SENSITIVITY

We vary the base sufficiency threshold $\tau_0$ using Gemini-3-Flash on the three-repository SWE-QA subset. In Table 9, $\tau_0$ is the configured threshold before query-complexity adjustments, and $\tau$ is the effective threshold in the stopping condition $\kappa_t \geq \tau$ (Section 4.3.2). *Mean final* $\kappa_t$ averages each question's context-sufficiency score at termination, while *Mean effective* $\tau$ averages the corresponding adjusted thresholds. Here, $\kappa_t$ follows Section 4.3.1 and is not a calibrated correctness probability. Budget, iteration, or low-gain stopping may leave final $\kappa_t$ below $\tau$. Higher $\tau_0$ yields higher mean final sufficiency and answer scores, with a smaller gain from 95 to 100, indicating an aggregate association between the sufficiency signal and answer quality.

Table 9: Threshold sensitivity on SWE-QA.

| **Base $\tau_0$** | **Mean final $\kappa_t$** | **Mean effective $\tau$** | **SWE-QA score** |
|---|---|---|---|
| 65 | 83.021 | 65.000 | 40.79 |
| 80 | 85.729 | 80.000 | 41.52 |
| 95 | 90.042 | 93.313 | 43.19 |
| 100 | 91.667 | 98.375 | 43.40 |

### A.3.4 FULL RESULTS ON SWE-QA

Table 10: Direct LLM results on SWE-QA across all repositories with different backbone LLMs.

| **Repository** | **DeepSeek-R1-8B** | **Gemini-3-Flash** | **Ministral-3B** | **Qwen3-Coder-30B** |
|---|---|---|---|---|
| astropy | 27.58 | 32.42 | 25.60 | 34.08 |
| sphinx | 28.54 | 32.67 | 24.60 | 34.44 |
| matplotlib | 31.52 | 33.88 | 28.38 | 37.48 |
| pylint | 28.46 | 32.79 | 27.33 | 35.56 |
| flask | 33.46 | 32.12 | 27.81 | 36.42 |
| pytest | 28.54 | 32.50 | 26.75 | 35.46 |
| django | 33.31 | 33.31 | 30.90 | 37.56 |
| xarray | 29.46 | 32.71 | 27.98 | 35.35 |
| scikit-learn | 31.48 | 31.46 | 27.25 | 35.10 |
| requests | 31.85 | 34.06 | 31.48 | 38.58 |
| sympy | 30.33 | 32.29 | 24.15 | 34.73 |
| sqlfluff | 26.75 | 31.62 | 26.44 | 34.44 |
| **Overall Average** | **30.11** | **32.65** | **27.39** | **35.77** |

To complement the backbone-level summary in Table 8, Tables 10, 11, and 12 further break down the total score over each of the 12 SWE-QA repositories under *Direct LLM*, *SWEQA-Agent*, and our method, FastCode. The gains from FastCode are not driven by a small subset of projects: for every backbone, it improves over the direct baseline on all repositories, spanning scientific libraries (e.g., *astropy*, *sympy*), developer tooling (e.g., *pylint*, *sqlfluff*), and web frameworks (e.g., *flask*, *django*). *SWEQA-Agent* is often helpful, but its per-repository behavior is less stable; for example, with *Qwen3-Coder-30B* on *django* it underperforms the direct baseline (29.77 vs. 37.56), while FastCode reaches 42.04. Overall, the repository-level results reinforce the conclusion above: improving context construction yields reliable benefits across diverse codebases and model scales.

Table 11: SWEQA-Agent results on SWE-QA across all repositories with different backbone LLMs.

| Repository | DeepSeek-R1-8B | Gemini-3-Flash | Ministral-3B | Qwen3-Coder-30B |
|---|---|---|---|---|
| astropy | 31.44 | 41.32 | 35.15 | 36.21 |
| sphinx | 32.71 | 42.08 | 29.35 | 38.40 |
| matplotlib | 35.12 | 43.48 | 33.75 | 37.67 |
| pylint | 34.29 | 42.60 | 34.85 | 37.12 |
| flask | 35.71 | 42.15 | 38.67 | 37.60 |
| pytest | 33.25 | 42.92 | 27.44 | 39.77 |
| django | 35.75 | 43.31 | 31.00 | 29.77 |
| xarray | 35.23 | 41.17 | 36.21 | 38.62 |
| scikit-learn | 35.06 | 41.54 | 35.46 | 37.21 |
| requests | 39.08 | 43.50 | 31.58 | 38.90 |
| sympy | 34.06 | 43.29 | 35.40 | 37.08 |
| sqlfluff | 31.79 | 42.27 | 32.56 | 36.60 |
| **Overall Average** | **34.46** | **42.47** | **33.45** | **37.08** |

Table 12: FastCode results on SWE-QA across all repositories with different backbone LLMs.

| Repository | DeepSeek-R1-8B | Gemini-3-Flash | Ministral-3-3B | Qwen3-Coder-30B |
|---|---|---|---|---|
| astropy | 36.40 | 41.58 | 33.88 | 40.33 |
| sphinx | 37.81 | 44.10 | 35.96 | 41.75 |
| matplotlib | 40.56 | 43.44 | 38.73 | 41.79 |
| pylint | 37.98 | 43.42 | 34.06 | 40.65 |
| flask | 40.27 | 44.98 | 38.12 | 43.19 |
| pytest | 38.65 | 44.12 | 35.00 | 41.96 |
| django | 38.50 | 44.06 | 35.25 | 42.04 |
| xarray | 37.35 | 43.04 | 35.56 | 40.65 |
| scikit-learn | 36.73 | 41.81 | 34.33 | 39.44 |
| requests | 42.19 | 44.29 | 41.08 | 43.56 |
| sympy | 37.04 | 42.77 | 34.62 | 39.79 |
| sqlfluff | 39.23 | 43.69 | 36.94 | 42.08 |
| **Overall Average** | **38.56** | **43.44** | **36.13** | **41.44** |