SWE-QA: Can Language Models Answer Repository-level Code Questions?

Weihan Peng[†], **Yuling Shi**[†], **Yuhang Wang**,
Xinyun Zhang, **Beijun Shen**, **Xiaodong Gu**[✉]
Shanghai Jiao Tong University
{peng-weihan, yuling.shi, lingbo_2022, xinyunz, bjshen, xiaodong.gu}@sjtu.edu.cn
GitHub: https://github.com/peng-weihan/SWE-QA-Bench
Hugging Face: https://huggingface.co/datasets/swe-qa/SWE-QA-Benchmark

Abstract

Understanding and reasoning about entire software repositories is an essential capability for intelligent software engineering tools. While existing benchmarks such as CoSQA and CodeQA have advanced the field, they predominantly focus on small self-contained code snippets. These setups fail to capture the complexity of real-world repositories, where effective understanding and reasoning often require navigating multiple files, understanding software architecture, and grounding answers in long-range code dependencies. In this paper, we present SWE-QA, a repository-level code question answering (QA) benchmark designed to facilitate research on automated QA systems in realistic code environments. SWE-QA involves 720 high-quality question-answer pairs spanning diverse categories, including intention understanding, cross-file reasoning, and multi-hop dependency analysis. To construct SWE-QA, we first crawled 77,100 GitHub issues from 12 popular repositories. Based on an analysis of naturally occurring developer questions extracted from these issues, we developed a two-level taxonomy of repository-level questions and constructed a set of seed questions for each category. For each category, we manually curated and validated questions and collected their corresponding answers. We evaluated six advanced LLMs on SWE-QA under various context augmentation strategies. Experimental results highlight the promise of LLMs.

1 Introduction

Understanding and reasoning about entire software repositories is an essential capability for intelligent software engineering tools. Real-world development rarely involves reasoning over isolated functions or small code snippets; instead, developers must navigate large, interconnected codebases, trace dependencies across multiple files, and synthesize architectural knowledge to answer complex questions.

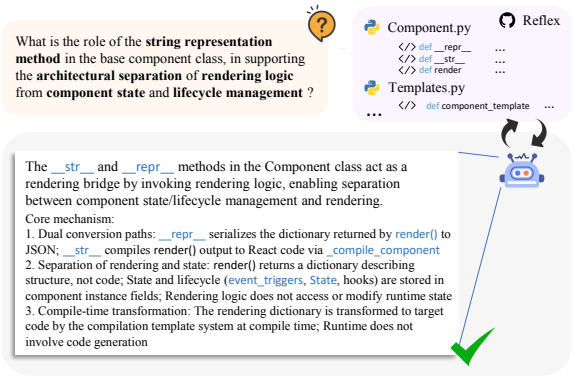

Figure 1: Repository-level Code QA Example

Recent advances in large language models (LLMs) have shown promise for code understanding (Feng et al., 2020; Wang et al., 2021; Zhu et al., 2024; Huang et al., 2024; Yang et al., 2025), yet most existing evaluations (Huang et al., 2021; Liu and Wan, 2021; Sahu et al., 2024; Lee et al., 2022; Li et al., 2024b) target isolated code snippets, functions, or APIs. These benchmarks fail to capture the complexity of real-world repositories, including architecture, cross-file dependencies, lifecycle flows, and design rationales, which require a deeper, multi-hop understanding of code structure, semantics, and intent (Zhang et al., 2023; Ouyang et al., 2024). While recent works like CodeRepoQA (Hu et al., 2024), CoreQA (Chen et al., 2025), and Spyder-CodeQA (Strich et al., 2024) have begun to address repository-level understanding, they still lack comprehensive coverage of the diverse reasoning patterns and multi-hop dependencies essential for realistic software development scenarios.

To address this limitation, we propose a repository-level code question answering (QA) benchmark designed to evaluate LLMs on realistic repository-based questions. SWE-QA comprises

[†]Equal contribution. [✉]Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 8230–8245
July 2-7, 2026

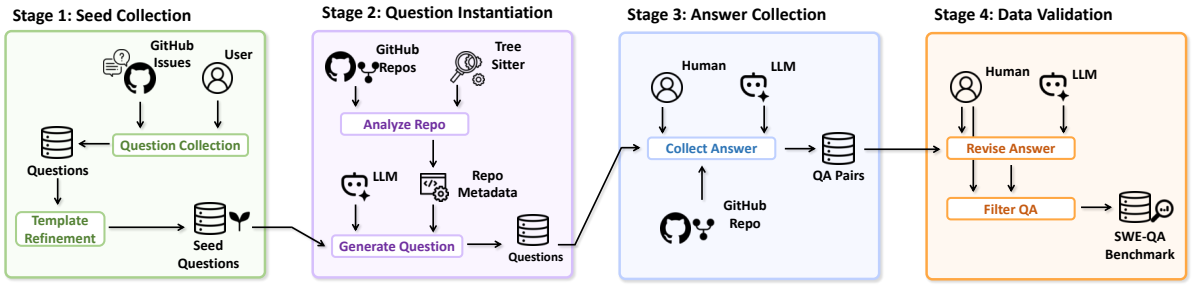

Figure 2: Workflow of benchmark construction.

720 high-quality QA pairs that necessitate a deep understanding of intentions, cross-file reasoning, and multi-hop dependency analysis. To capture the diverse reasoning requirements inherent in real world, we crawled 77,100 GitHub issues from 12 repositories used by SWE-bench (Jimenez et al., 2024). Based on an analysis of naturally occurring questions raised from these issues, we developed a two-level taxonomy of repository-level questions and constructed a set of seed questions for each category. Guided by our taxonomy and seed templates, we instantiated candidate questions from 15 repositories (12 from SWE-Bench and 3 from SWE-Bench-Live (Zhang et al., 2025)) using LLMs, then manually screened and refined them to obtain 48 high-quality QA pairs per repository. Each question was answered based on retrieved context by a strong LLM, and preliminary answers were reviewed for correctness, completeness, and clarity. This process produces high-quality reference answers grounded in code context, forming a reliable and scalable benchmark with diverse reasoning requirements.

We evaluated six advanced LLMs on SWE-QA using various context augmentation methods. Direct prompting performs poorly, while standard RAG methods significantly improve performance. Agent-based frameworks further enhance results, with OpenHands achieving 70.79 using GPT-5.1. These results are further validated by human evaluation, which shows a high level of agreement with LLM-as-a-Judge scores, indicating the reliability of LLM-based evaluation. Further analysis shows that models perform well on open-ended *How* and *Why* questions but struggle with constrained *What* and locational *How* questions requiring multi-hop reasoning. Performance also varies across repositories, with some like "pylint" proving particularly challenging. Overall, the experimental results highlight the promise of LLMs, particularly in the agent framework, in addressing repository-level QA, while also revealing open challenges and pointing to future research directions.

In summary, our contributions are as follows:

- **SWE-QA**: a repository-level code QA benchmark comprising 720 high-quality question-answer pairs from 15 diverse open-source Python repositories for evaluating comprehensive repository understanding.
- Flexible QA generation pipeline: enables efficient creation of question-answer datasets for any new repository using seed questions.
- LLM evaluation under context augmentation: assesses the ability of LLMs to answer questions with various methods, including RAG and diverse Agent Frameworks.

2 Benchmark Construction

In this section, we introduce SWE-QA, a novel benchmark designed for repository-level code question answering. As shown in Figure 2, our benchmark construction pipeline consists of four main stages: seed question collection, question instantiation, answer collection, and data validation. Each of these stages is detailed in the following subsections.

2.1 Seed Collection and Taxonomy Construction

To ensure SWE-QA reflects the complexities of real-world software engineering, we first conducted an empirical study to understand the types of questions developers pose when working with large codebases. We systematically collected and analyzed a large corpus of questions from GitHub issues to build a comprehensive taxonomy of repository-level questions. This taxonomy serves as the foundation for our benchmark construction.

Our data collection process began by crawling 77,100 GitHub issues from the 12 popular repositories used in SWE-bench (Jimenez et al., 2024) (details of collected issues are available in Ap-

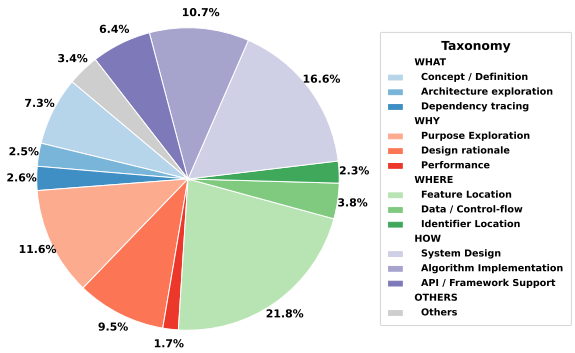

Figure 3: Distribution of question types.

pendix A.1). To focus on substantive discussions, we filtered for issues with a body length of at least 1,000 characters, resulting in a dataset of 41,955 issues. Given that issues often contain extensive descriptive text, we employed a large language model to parse each issue and extract explicit questions related to code understanding (see Prompt 1 in Appendix E for details). This process yielded 127,415 distinct questions, with an average of 3.04 questions per issue.

We first randomly sampled 1,000 questions, which were manually analyzed by two annotators with three years of development experience using an iterative top-down and bottom-up open coding process to identify recurring patterns and developer intentions. The two annotators performed the coding in parallel, and any disagreements were resolved through discussion among the authors until full consensus was reached. This yielded a structured two-level taxonomy for repository-level QA, as summarized in Table 1. The first level categorizes questions based on their primary interrogative: *What*, *Why*, *Where*, and *How*. The second level further classifies questions into 12 fine-grained user intentions, such as dependency tracing, design rationale clarification, and algorithm analysis, which reflect common software engineering activities.

With the taxonomy fixed, we then used a strong LLM (GPT-5) to classify the remaining 126,415 questions into the Level-1/Level-2 categories using concise labeling prompts with consistency checks. The resulting distribution, illustrated in Figure 3, reveals that *How* questions are the most frequent (35.2%), focusing on implementation details like system design and algorithms. *Where* questions follow at 28.4%, indicating that developers often need to locate features, data flows, or specific identifiers. *Why* questions, which probe design rationales and purpose, make up 23.1% of the corpus. Finally, *What* questions, seeking definitions or architectural summaries, account for the remaining 13.3%. This distribution underscores that a significant portion of developer queries are centered on procedural and locational knowledge, highlighting the need for QA systems that can reason deeply about code structure and implementation.

Based on this taxonomy, we created a set of abstract seed templates for each user intention category. These templates, such as "What are the subclasses that inherit from the <Class> class?" and "How does <Module> implement <Feature> in case of <Condition>?", capture the essence of recurring questions. As detailed in Table 1, these templates serve as the blueprint for generating diverse, context-specific question instances tailored to individual repositories.

2.2 Question Instantiation and Expansion

The objective of this stage is to generate context-specific question instances tailored to a target repository. To extract relevant contextual information, we parse the structure of each repository using `tree-sitter`[1](see Appendix A.2 for details). We instantiate questions by selecting a compact subgraph around a focal element (e.g., a class or method) and combining it with seed templates from Stage 1. The subgraph ensures sufficient context without overwhelming the prompt (see Prompt 2 in Appendix E for details). For a concrete example of question generation, we refer to Appendix A.3, where a full instance is presented.

2.3 Answer Collection

Once the questions are instantiated, we proceed to generate initial reference answers for each question. This stage, illustrated as Stage 3 in Figure 2, leverages a retrieval-augmented generation (RAG) pipeline designed to ground answers firmly in the repository's context. The process involves two key steps:

Step 1: Context Retrieval. For each question, we first build a comprehensive index of the target repository's code elements, including functions, classes, and their inter-dependencies. Using this index, we retrieve relevant code snippets, documentation, and architectural metadata through a combination of semantic similarity matching and structural dependency analysis. This ensures a rich, relevant context for answer generation.

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

Type	Intention	Definition	Example
What	Architecture exploration	Identify components or structures of the system	What is the semantic relationship between the expiration threshold computed in the test initialization method and the early-exit optimization tested across multiple test methods?
	Concept / Definition	Understand meaning of code elements	What does the method that enables the test helper class representing pathlib.Path objects return when invoked by the standard library path conversion function?
	Dependency tracing	Relationships or dependencies among code elements	What is the dependency chain in the test function that verifies subdomain routing through configuration, route registration, and request context creation?
Why	Design rationale	Explain why certain design decisions are made	Why does the assignment name processing method defer frame node visitation until the local variable type attribute is accessed rather than visiting frames upfront?
	Purpose Exploration	Understand function or module purpose	Why does the guess_filename function's validation logic prevent security vulnerabilities when extracting filenames from file-like objects in multipart form data handling?
	Performance	Understand performance considerations	Why does the preprocessing transformer that generates univariate B-spline basis functions use an identity coefficient matrix during the fitting process?
Where	Data / Control-flow	Localize variables or control statements	Where in the serialization pipeline for the JSON tag class that handles Markup API objects are the lower-level helper functions that the conversion method delegates to?
	Feature Location	Identify where a feature is implemented	Where in the AxLine class is the coordinate transformation logic applied to reconcile between data coordinates and display coordinates before computing the line endpoints?
	Identifier Location	Find where an identifier is defined or used	Where in the LocalPath class can I locate the code logic responsible for assigning the absolute path value to the strpath attribute during initialization?
How	System Design	Explain overall system operation	How does the abstract base class of muscle activation models decouple symbolic equation representation from numerical solver implementations via abstract property interfaces?
	Algorithm Implementation	Understand algorithm steps	How does the documentation generation system determine which class members should be excluded from generated documentation?
	API / Framework Support	Show usage of APIs or frameworks	How does the test helper function validating dialect-specific SQL statements ensure the parsed tree matches the expected segment hierarchy and statement count via recursive validation?

Table 1: Taxonomy of repository-level questions, including intention, definition, and example for each type.

Step 2: Initial Answer Generation. With the retrieved context, we utilize a powerful LLM, assisted by 4 human experts with at least three years of experience in software development using tools like Cursor, to generate a preliminary answer. This process is guided by a prompt that emphasizes factual accuracy, completeness, and strict adherence to the provided context. The prompt explicitly directs the model to cite code locations and avoid introducing information not present in the retrieved materials, thus minimizing hallucination. The resulting question-answer pairs serve as the input for the subsequent data validation stage.

2.4 Data Validation

To ensure the highest quality and reliability of our benchmark, all preliminary QA pairs undergo a rigorous data validation process, as depicted in Stage 4 of Figure 2. This multi-phase procedure is conducted by experienced developers with deep familiarity with the target repositories and involves both answer revision and quality filtering.

Step 1: Expert Answer Revision. Each generated answer is manually reviewed by our expert team. With the assistance of LLM-powered tools like Cursor, reviewers meticulously verify the factual accuracy of every claim, assess the completeness of the explanation, and refine the language for clarity and precision. Each question is answered independently by two experts, and their answers are cross-validated; if there is significant disagreement, a third expert also participates in answer generation and the three discuss to reach a final consensus. This human-in-the-loop approach allows for nuanced corrections that automated systems might miss, ensuring each answer is not only correct and complete but also easy to understand.

Step 2: Quality Filtering. After revision, the QA pairs are subjected to a final filtering step. Pairs are discarded if they fail to meet our quality standards. The criteria for filtering include, but are not limited to: questions that are ambiguous or poorly formulated, answers that remain factually incorrect or incomplete after revision, or answers that cannot be sufficiently grounded in the repository's code and documentation. We also enforce per-repository balance across Level-1 categories (*What*, *Why*, *Where*, *How*), yielding exactly 48 finalized pairs per repository. This stringent filtering ensures that only the most clear, correct, and valuable QA pairs are included in the final SWE-QA benchmark.

2.5 Statistics of SWE-QA

The benchmark comprises 720 questions meticulously curated from 15 Python repositories. In total, these repositories encompass 13,300 files, 22,522 classes, 142,404 functions, and over 3.4 million lines of code, presenting a substantial and realistic challenge for code understanding models. The average question length is 28.62 words, and the average answer length is 266.64 words. To further characterize the complexity of SWE-QA, we conduct a quantitative analysis of the reasoning

Dataset	Year	Source	Test Data Size	Repo Level?	Module Reasoning?	Multi -hop?	Cross file?	Human Verified?
CoSQA (Huang et al., 2021)	2021	Bing Search Logs	1046	✗	✗	✗	✗	✓
CodeQA (Liu and Wan, 2021)	2021	GitHub Code Comments	Java: 11,978 Python: 7,009	✗	✗	✗	✗	✓
CodeQueries (Sahu et al., 2024)	2022	ETH Py150 Open (GitHub Python code)	29,033	✗	✗	✓	✗	✗
CS1QA (Lee et al., 2022)	2022	Python Programming Courses Chat Logs	1,847	✗	✗	✗	✗	✓
ProCQA (Li et al., 2024b)	2024	StackOverflow	∼500,000	✗	✗	✗	✗	✓
InfiBench (Li et al., 2024a)	2024	Stack Overflow	234	✗	✗	✗	✗	✓
CoReQA (Chen et al., 2025)	2025	GitHub Issues and Comments	1,563	✓	✗	✗	✓	✓
SWE-QA (Ours)	2025	GitHub Repositories	720	✓	✓	✓	✓	✓

Table 2: Comparison between SWE-QA and existing code QA benchmarks. SWE-QA focuses on repository-level multi-hop questions that require complex reasoning and deep repository understanding.

chain and structural requirements of its questions. On average, answering each question involves 8.71 functions across 3.19 files, with a reasoning chain depth of 4.72 and a dependency chain depth of 2.96. Moreover, 90.9% of the questions exhibit a reasoning chain depth greater than one, and 77.6% require cross-file knowledge to be answered correctly, indicating that SWE-QA systematically emphasizes multi-hop reasoning and non-trivial code dependencies. To ensure a balanced representation of question types, we selected an equal number of What, Why, Where, and How questions, with each repository contributing exactly 48 samples. Detailed statistics for each repository can be found in Appendix A.4.

Table 2 presents a comprehensive comparison between SWE-QA and existing code QA benchmarks. While traditional benchmarks like CoSQA (Huang et al., 2021) and CodeQA (Liu and Wan, 2021) focus on isolated code snippets, SWE-QA operates at the repository level, similar to recent benchmarks such as CodeRepoQA (Hu et al., 2024) and CoreQA (Chen et al., 2025). However, SWE-QA is unique in its comprehensive design, incorporating categorized, multi-hop questions that require cross-file context, with all question-answer pairs being human-verified. This multi-faceted design addresses the limitations of existing repository-level benchmarks, which often lack one or more of these critical dimensions. For instance, CodeRepoQA (Hu et al., 2024) lacks human verification and categorization, while Spyder-CodeQA (Strich et al., 2024) and CoreQA (Chen et al., 2025) do not feature multi-hop questions. SWE-QA's realistic construction setting imbues the dataset with the following unique features:

Repository-Level Granularity and Complexity. Unlike benchmarks targeting isolated code elements (e.g., functions or classes), SWE-QA evaluates repositories, reflecting real-world tasks that require cross-file context, architectural understanding, and dependency tracking. This design challenges models to answer questions in complex, interconnected codebases, with substantially higher cognitive and structural demands than existing code QA datasets.

Pipeline-Level Extensibility. SWE-QA serves as both a benchmark and a modular pipeline for generating new repository-level QA instances applicable to any open-source project. Leveraging static code analysis, LLM prompting, and human filtering, new benchmarks can be continuously and semi-automatically created, ensuring scalability and adaptability to evolving codebases.

3 Evaluation

To showcase the usefulness of SWE-QA, we assess the performance of language models on repository-level code question answering using the proposed benchmark. Our objective is to uncover novel insights that have not been previously explored through comprehensive and in-depth comparisons of existing language models.

3.1 Experiment Setup

Model Selection. We evaluate six representative LLMs: Kimi K2 (Moonshot-AI, 2025), Qwen3-Coder-30B-A3B-Instruct (Qwen Team, 2025a), Qwen3-Coder-480B-A35B-Instruct (Qwen Team, 2025b), GLM-4.6 (Zhipu AI (Z.ai), 2025), Gemini 2.5 Pro (Google-AI, 2025) and GPT-5.1 (OpenAI, 2025).

Studied Methods. For each model, we study five approaches: *Direct Prompting*, two retrieval-augmented generation variants (*Sliding Window RAG* and *Function Chunking RAG*), and two agent-based frameworks (*OpenHands* and *SWE-agent*).

We further include two commercial programming tools, Tongyi Lingma[2] and Cursor-agent[3], as system-level baselines. Detailed configurations are reported in Appendix B.1.

Metrics. We adopt an LLM-based evaluation (LLM-as-Judge) to assess repository-level code question answering performance (see Prompt 3). Prior studies have demonstrated the reliability of LLM-as-Judge across both natural language generation (Liu et al., 2023; Song et al., 2024) and software engineering tasks (Wang et al., 2025a; He et al., 2025). Given a model's output and the gold answer, an LLM—Claude Sonnet 4.5 (Anthropic, 2025) in our experiments—evaluates answer quality along five dimensions: **correctness**, **completeness**, **relevance**, **clarity**, and **Coherence**. Each dimension is scored on a 20-point scale, yielding a total score of 100 (see Appendix B.2 for details). To mitigate potential self-evaluation bias, we enforce strict judge–candidate separation, anonymize systems, randomize answer order, and complement automated evaluation with a human study (Appendix C.1).

3.2 Performance of Language Models

Table 3 summarizes the overall performance of different language models on all QA pairs in SWE-QA, revealing clear performance gaps across both methods and model choices for repository-level question answering.

Impact of Context Augmentation Methods. We first examine the effect of context augmentation. Direct prompting without repository context consistently yields the lowest performance across all models (e.g., Kimi K2 scores 51.47/100), highlighting the necessity of grounded code context. Both Sliding Window RAG and Function Chunking RAG substantially improve results by retrieving relevant code snippets (e.g., Kimi K2 improves to 62.44 with Sliding Window RAG). Agent-based frameworks further boost performance by enabling iterative reasoning and tool use, leading to the best results among open methods (e.g., Kimi K2 reaches 67.72 with SWE-agent). Overall, richer and more structured context access translates into steadily improved answer quality.

Impact of Language Model Choice. Model choice also plays a critical role. GPT-5.1 achieves the best overall performance, reaching 70.79 when combined with OpenHands. Notably, strong open-source models perform competitively: GLM-4.6 with OpenHands attains 70.15, closely matching GPT-5.1. In contrast, smaller models such as Qwen3-Coder-30B-A3B-Instruct benefit less from agent frameworks, with agent-based methods performing on par with or worse than RAG. This suggests that effective agent planning, tool invocation, and long-term memory remain challenging for smaller-capacity models.

Performance of Commercial Programming Tools. We further evaluate two commercial programming tools, Tongyi Lingma and Cursor. Both exhibit strong performance, with Cursor achieving 70.66—second only to GPT-5.1 with OpenHands—and Tongyi Lingma scoring 69.07. As highly integrated systems combining proprietary LLMs with advanced retrieval and orchestration, their results provide a strong reference point and underscore the effectiveness of end-to-end, tool-augmented solutions for complex repository-level question answering.

3.3 Taxonomy-Aware Analysis

To understand how performance varies across different types of repository-level questions, we conduct a taxonomy-aware analysis using OpenHands. Table 4 breaks down the scores for each model across the 12 question intentions defined in our taxonomy.

The results reveal a discrepancy between high-level, conceptual questions and low-level, implementation-focused queries. Models consistently achieve the highest scores on *Why* questions (average 69.77), particularly "Design rationale" (71.48), and perform well on *How* questions related to "API / Framework Support" (68.75). This suggests models excel when the required information is explicitly expressed in natural language (e.g., docstrings, comments, architectural notes).

Performance is lower on questions that require deep procedural or locational understanding. *What* questions (e.g., explore architectures or trace dependencies) yield the lowest average score (65.81), with "Architecture exploration" at 61.84. *Where* questions (66.76), which demand precise location, also remain challenging. These categories often require reconstructing dispersed logic across files and implicit control paths beyond inline documentation, stressing code tracing and dependency reasoning.

[2] https://lingma.aliyun.com/, v2.5.16
[3] https://cursor.com, v2025.09.04

Model	Evaluation Metrics					Overall
	Correctness	Completeness	Relevance	Clarity	Reasoning	
Commercial Tools						
Tongyi Lingma	12.02	10.99	16.78	15.67	13.61	69.07
Cursor	12.11	11.44	17.21	16.01	13.89	70.66
Open-Source Frameworks						
Qwen3-Coder-30B-A3B-Instruct	7.39	5.42	14.23	14.66	9.11	50.80
+ Function Chunking RAG	10.88 (+3.49)	8.72 (+3.30)	15.61 (+1.38)	16.24 (+1.58)	11.95 (+2.84)	63.40 (+12.60)
+ Sliding Window RAG	11.22 (+3.83)	9.00 (+3.58)	15.81 (+1.58)	16.71 (+2.05)	12.12 (+3.01)	64.86 (+14.06)
+ SWE-agent	9.12 (+1.73)	8.64 (+3.22)	14.98 (+0.75)	12.92 (-1.74)	11.77 (+2.66)	57.44 (+ 6.64)
+ OpenHands	11.09 (+3.70)	10.40 (+4.98)	15.94 (+1.71)	15.08 (-0.58)	13.36 (+4.25)	65.88 (+15.08)
Qwen3-Coder-480B-A35B-Instruct	8.14	6.09	15.00	15.50	9.88	54.61
+ Function Chunking RAG	10.61 (+2.47)	8.56 (+2.47)	16.08 (+1.08)	16.07 (+0.57)	11.63 (+1.75)	62.96 (+ 8.35)
+ Sliding Window RAG	11.36 (+3.22)	9.11 (+3.02)	16.34 (+1.34)	16.51 (+1.01)	12.06 (+2.18)	65.39 (+10.78)
+ SWE-agent	11.09 (+2.95)	10.59 (+4.50)	15.49 (+0.49)	13.76 (-1.74)	13.06 (+3.18)	64.00 (+ 9.39)
+ OpenHands	11.89 (+3.75)	11.20 (+5.11)	16.21 (+1.21)	15.37 (-0.13)	13.90 (+4.02)	68.57 (+13.96)
Kimi K2	7.41	4.93	15.23	15.75	8.15	51.47
+ Function Chunking RAG	9.78 (+2.37)	7.93 (+3.00)	16.55 (+1.32)	15.11 (-0.64)	10.70 (+2.55)	60.08 (+ 8.61)
+ Sliding Window RAG	10.44 (+3.03)	8.55 (+3.62)	16.48 (+1.25)	15.65 (-0.10)	11.32 (+3.17)	62.44 (+10.97)
+ SWE-agent	11.84 (+4.43)	11.72 (+6.79)	15.88 (+0.65)	14.48 (-1.27)	13.80 (+5.65)	67.72 (+16.25)
+ OpenHands	11.74 (+4.33)	11.33 (+6.40)	15.21 (-0.02)	14.66 (-1.09)	13.24 (+5.09)	66.18 (+14.71)
GLM-4.6	8.71	6.09	15.79	16.53	9.53	56.64
+ Function Chunking RAG	9.48 (+0.77)	7.52 (+1.43)	16.07 (+0.28)	14.41 (-2.12)	9.99 (+0.46)	57.46 (+ 0.82)
+ Sliding Window RAG	9.02 (+0.31)	7.85 (+1.76)	16.33 (+0.54)	16.08 (-0.45)	10.44 (+1.91)	60.72 (+ 3.08)
+ SWE-agent	12.31 (+3.60)	12.37 (+6.28)	16.15 (+0.36)	14.81 (-1.72)	14.21 (+4.68)	69.85 (+13.21)
+ OpenHands	11.91 (+3.20)	12.70 (+6.61)	16.90 (+1.11)	13.82 (-2.71)	14.82 (+5.29)	70.15 (+13.51)
Gemini 2.5 Pro	7.84	5.59	16.28	15.83	9.18	54.71
+ Function Chunking RAG	9.91 (+2.07)	7.53 (+1.94)	16.79 (+0.51)	15.32 (-0.51)	10.47 (+1.29)	60.02 (+ 5.31)
+ Sliding Window RAG	11.52 (+3.68)	8.83 (+3.24)	16.85 (+0.57)	16.66 (+0.83)	11.82 (+2.64)	65.68 (+10.97)
+ SWE-agent	10.62 (+2.78)	10.11 (+4.52)	15.96 (-0.32)	13.81 (-2.02)	13.26 (+4.08)	63.76 (+ 9.05)
+ OpenHands	10.75 (+2.91)	9.64 (+4.05)	16.78 (+0.50)	15.20 (-0.63)	13.27 (+4.09)	65.63 (+10.92)
GPT-5.1	9.62	7.37	16.23	16.86	11.33	61.41
+ Function Chunking RAG	11.38 (+1.76)	9.50 (+2.13)	17.12 (+0.89)	16.15 (-0.71)	12.43 (+1.10)	66.57 (+5.16)
+ Sliding Window RAG	11.45 (+2.38)	9.44 (+2.07)	17.01 (+0.78)	16.28 (-0.58)	12.61 (+1.28)	66.79 (+5.38)
+ SWE-agent	12.94 (+3.32)	11.44 (+4.07)	14.80 (-1.43)	16.34 (-0.52)	13.68 (+2.35)	69.20 (+7.79)
+ OpenHands	12.26 (+2.64)	11.60 (+4.23)	16.82 (+0.59)	15.59 (-1.27)	14.52 (+3.19)	70.79 (+9.38)

Table 3: Comparative performance of different language models on SWE-QA. GPT-5.1 achieves the best performance among all models. In general, RAG-enhanced methods outperform direct inference, and agent-based methods further improve upon RAG methods.

3.4 Cross-Repository Generalization

To assess the generalization capabilities of the models, we analyze their performance across the 12 different repositories in SWE-QA, again using OpenHands. The results, detailed in Table 5, indicate that while performance is generally consistent, it can be significantly influenced by the specific characteristics of each repository.

On average, most repositories present a similar level of difficulty, with scores clustering around the 70-point mark. For instance, "django" (67.39), "astropy" (68.29) and "scikit-learn" (71.34) show comparable results. However, we observe notable outliers. The "flask" repository appears to be easier on average (75.42). Conversely, "pylint " (62.01) and "conan" (64.51) are more challenging. This variance aligns with factors such as codebase size, architectural complexity, plugin or hook systems, API surface clarity, and unconventional patterns that increase reasoning depth.

Furthermore, the 12 repositories selected from SWE-Bench achieve an average score of 68.59, whereas the 3 repositories drawn from SWE-Bench-Live average 64.98, representing a decrease of 3.61 points. This gap may be related to the reduced data leakage in SWE-Bench-Live, which makes the evaluation more challenging.

4 Related Work

Several benchmarks have been developed to evaluate code question answering (QA) systems, from snippet-level to emerging repository-level assessments. CodeQueries (Sahu et al., 2024) evaluates single- and multi-hop reasoning over Python code

Question Type	Qwen3-Coder-30B-A3B-Instruct	Qwen3-Coder-480B-A35B-Instruct	Kimi K2	GLM-4.6	Gemini 2.5 Pro	GPT-5.1	Average
What	62.59	67.28	64.58	67.51	64.52	68.39	65.81
Architecture exploration	58.95	64.24	60.25	63.40	60.33	63.89	61.84
Concept / Definition	65.60	69.94	68.30	70.86	67.47	71.10	68.88
Dependency tracing	63.19	67.72	65.14	68.34	64.84	70.17	66.57
Why	69.33	69.42	66.16	72.57	68.04	73.09	69.77
Design rationale	70.90	70.72	67.54	74.75	69.72	75.26	71.48
Purpose Exploration	69.74	70.14	66.77	73.39	69.16	74.15	70.56
Performance	67.33	67.43	64.06	69.50	65.21	69.90	67.24
Where	65.92	69.08	63.98	69.01	63.29	69.28	66.76
Data / Control-flow	64.44	67.82	62.10	67.18	61.83	66.94	65.05
Feature Location	65.60	68.88	63.51	68.55	63.05	68.51	66.35
Identifier Location	67.72	70.62	65.25	71.28	64.93	72.39	68.70
How	65.68	68.50	70.00	71.51	66.68	72.40	69.13
System Design	65.31	68.01	69.49	70.76	66.15	71.75	68.58
Algorithm Implementation	66.37	68.88	70.47	72.34	67.28	73.23	69.76
API / Framework Support	65.41	68.59	70.04	71.39	65.78	71.28	68.75

Table 4: Results across different question types by OpenHands. What and Where question types present greater challenges. Among the subtypes, Architecture exploration and Data / Control-flow are the most challenging.

Repository	Qwen3-Coder-30B-A3B-Instruct	Qwen3-Coder-480B-A35B-Instruct	Kimi K2	GLM-4.6	Gemini 2.5 Pro	GPT-5.1	Average
From SWE-Bench	66.57	69.92	66.65	70.78	66.06	71.54	68.59
astropy	67.42	67.20	68.54	69.39	67.22	69.95	68.29
django	66.92	68.02	66.67	69.54	65.02	68.15	67.39
flask	74.38	73.50	75.41	76.99	75.34	76.91	75.42
matplotlib	72.13	74.07	73.99	72.26	68.45	75.81	72.78
pylint	55.95	63.77	62.51	62.38	62.35	65.13	62.01
pytest	60.57	71.66	59.02	71.03	57.91	71.48	65.28
requests	70.42	76.40	68.53	78.32	67.00	79.00	73.28
scikit-learn	70.08	71.65	70.98	72.60	70.58	72.17	71.34
sphinx	64.51	65.18	62.82	66.22	65.09	69.15	65.50
sqlfluff	63.53	66.58	61.87	69.70	63.65	70.62	65.99
sympy	63.53	67.89	61.87	67.21	63.17	67.50	65.20
xarray	69.44	73.09	67.58	73.71	67.00	72.67	70.58
From SWE-Bench-Live	63.11	63.18	64.30	67.63	63.89	67.78	64.98
conan	62.28	62.55	63.98	67.49	64.03	66.72	64.51
reflex	66.10	64.12	64.93	67.12	64.77	68.16	65.87
streamlink	60.93	62.87	63.99	68.28	62.88	68.45	64.57

Table 5: Results across different repositories by OpenHands. Performance varies substantially across different repositories. Repositories from SWE-Bench-Live present greater challenges compared to those from SWE-Bench.

with annotated answer spans. CoSQA (Huang et al., 2021) uses real user queries from Bing labeled by answer correctness. CodeQA (Liu and Wan, 2021) generates Q&A pairs for Python and Java methods via templates. CS1QA (Lee et al., 2022) provides 9,237 Q&A pairs from introductory Python courses, including tasks like question type classification and code line selection. Recent repository-level benchmarks include CodeRepoQA (Hu et al., 2024) with 585,687 entries across five languages, CoreQA (Chen et al., 2025) from GitHub issues across 176 repositories, Spyder-CodeQA (Strich et al., 2024) with 325 Python Q&A pairs, InfiBench (Li et al., 2024a) for freeform real-world questions, and ProCQA (Li et al., 2024b) from StackOverflow. These benchmarks highlight the challenge of handling long-range dependencies and multi-file contexts (Wang et al., 2025b). However, most evaluations focus on isolated snippets or functions and do not capture repository-level complexity. To address this gap, we propose a repository-level QA benchmark to evaluate LLMs on realistic, multi-file code understanding tasks.

5 Conclusion

In this paper, we present SWE-QA, a new benchmark for evaluating LLMs on realistic repository-level code questions. SWE-QA consists of 720 high-quality question–answer pairs spanning 15 Python repositories from SWE-Bench and SWE-Bench-Live, and is designed to capture practical reasoning challenges such as multi-hop dependencies and cross-file context. In future work, we intend to extend SWE-QA to additional programming languages and dynamically evolving repositories to enable broader and more robust evaluation of AI-assisted code intelligence tools.

Limitations

Despite the strengths demonstrated by SWE-QA, several limitations remain. First, SWE-QA focuses on Python repositories, which may limit the generalizability of our findings to other programming languages, despite the language-agnostic design of our methods. Second, our evaluation relies on a combination of LLM-as-Judge and limited-scale human evaluation; although the latter is used to validate the former, both may introduce biases and may not fully capture all nuances of answer quality. Third, the benchmark is constructed from static code snapshots and does not reflect the dynamics of evolving repositories. Finally, while we select 15 diverse repositories, they may not cover the full spectrum of software projects, particularly smaller or highly domain-specific codebases.

Acknowledgments

This paper is supported by the National Key Research and Development Program of China (Grant No. 2023YFB4503802) and the Natural Science Foundation of Shanghai (Grant No. 25ZR1401175).

References

Anthropic. 2025. Claude Sonnet 4.5 model. https://www.anthropic.com/news/claude-sonnet-4-5/.

Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and Shuiguang Deng. 2025. Coreqa: uncovering potentials of language models in code repository question answering. *arXiv preprint arXiv:2501.03447*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13.

Google-AI. 2025. Gemini-2.5-Pro model. https://ai.google.dev/gemini-api/docs/models?hl=en#gemini-2.5-pro.

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*.

Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. 2024. Coderepoqa: A large-scale benchmark for software engineering question answering. *arXiv preprint arXiv:2412.14764*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20, 000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5690–5700. Association for Computational Linguistics.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, and 1 others. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

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*.

Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. CS1QA: A dataset for assisting code-based question answering in an introductory programming course. *CoRR*, abs/2210.14494.

Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. 2024a. Infibench: Evaluating the question-answering capabilities of code large language models. *Advances in Neural Information Processing Systems*, 37:128668–128698.

Zehan Li, Jianfei Zhang, Chuantao Yin, Yuanxin Ouyang, and Wenge Rong. 2024b. Procqa: A large-scale community-based programming question answering dataset for code search. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13057–13067.

Chenxiao Liu and Xiaojun Wan. 2021. Codeqa: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2618–2632. Association for Computational Linguistics.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. 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.

Moonshot-AI. 2025. kimi-k2-0905-preview model. https://platform.moonshot.ai.

OpenAI. 2025. gpt-5.1-2025-11-13 model. https://openai.com/index/gpt-5-1-for-developers/.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. Repograph: Enhancing ai software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*.

Qwen Team. 2025a. qwen3-coder-30b-a3b-instruct model. https://qwenlm.github.io/blog/qwen3-coder/.

Qwen Team. 2025b. qwen3-coder-480b-a35b-instruct model. https://qwenlm.github.io/blog/qwen3-coder/.

Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish K. Shevade. 2024. Codequeries: A dataset of semantic queries over code. In *Proceedings of the 17th Innovations in Software Engineering Conference, ISEC 2024, Bangalore, India, February 22-24, 2024*, pages 7:1–7:11. ACM.

Hwanjun Song, Hang Su, Igor Shalyminov, Jason Cai, and Saab Mansour. 2024. Finesure: Fine-grained summarization evaluation using llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 906–922.

Jan Strich, Florian Schneider, Irina Nikishina, and Chris Biemann. 2024. On improving repository-level code qa for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 209–244.

VoyageAI. 2024. voyage-code-3: more accurate code retrieval with lower dimensional, quantized embeddings. https://blog.voyageai.com/2024/12/04/voyage-code-3.

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, and 1 others. 2024a. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024b. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 165–177. IEEE Computer Society.

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.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

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*.

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, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, and 1 others. 2025. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*.

Zhipu AI (Z.ai). 2025. glm-4.6 model. https://docs.bigmodel.cn/cn/guide/models/text/glm-4.6.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Benchmark Construction

A.1 Distribution of Collecting Issues

The distribution of issues collected across repositories is shown in Figure 4. Sympy contributed the largest proportion of issues, accounting for 17.8%, followed by Xarray with 15.2%. In contrast, Flask and Requests contributed relatively fewer issues, with 3.5% and 5.3%, respectively.

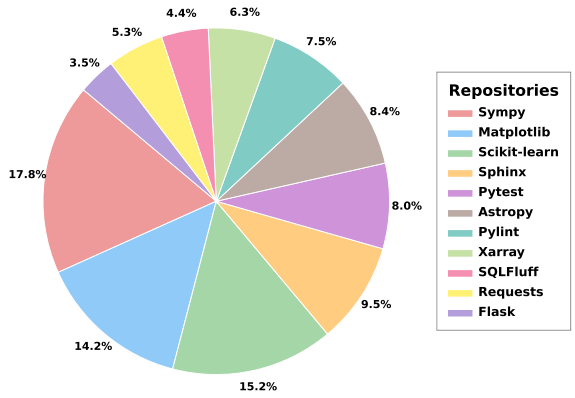

Figure 4: Distribution of collected issues.

A.2 Code Repository Parsing

To extract relevant contextual information, we parse the structure of each repository using `tree-sitter`[4], a language-agnostic parsing tool. This process produces a typed graph of core elements and their relationships (Figure 5), where nodes include **Repository**, **File**, **Code Snippet**, **Class**, **Method**, **Attribute**, **Function**, **Parameter**, and **Variable**. Edges represent containment relationships (e.g., Class → Method/Attribute, File → Code Snippet). Additionally, each function tracks the functions it calls and those that call it, while each file records its imports, revealing inter-file dependencies. Overall, this structure captures both type-level and call-level dependencies, providing the necessary foundation for multi-hop reasoning.

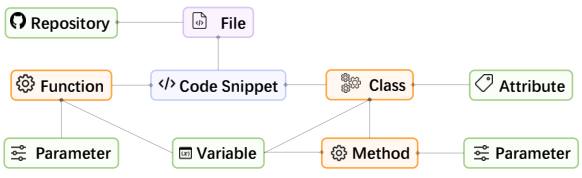

Figure 5: Core elements and their relations extracted from code repositories.

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

A.3 Illustrative Example of Question Instantiation

Figure 6 illustrates an instance of the question instantiation process. The left panel summarizes the focal function and its surrounding context; the top-right panel shows the five seed questions from the selected taxonomy category; the middle-right panel depicts how the LLM synthesizes a single, non-compound question tailored to the target function; and the bottom-right panel presents the curated reference answer with repository-grounded constraints (e.g., version guards and backend differences). All structural elements and the complete seed set are provided to the LLM to support candidate question generation.

A.4 Detailed Statistics of SWE-QA

Table 6 provides a comprehensive statistical overview of SWE-QA across repositories. The 15 repositories span a wide range in size, from 36 to 2,845 files and from 598 to 33,994 functions, offering a thorough representation of repositories of varying scales. Analyzing the benchmark scores in Table 5, we observe that Flask and Requests, each containing fewer than 100 files, achieve the two highest scores, whereas Pylint, with over 2,000 files, attains the lowest score. This indicates a strong correlation between the difficulty of repository comprehension and the inherent complexity of the repository. Additionally, the average question length is 28.62 words, and the average answer length is 266.64 words. For answers, the average edit distance after manual corrections is 42.17 words.

B Experimental Details

B.1 Method Configurations

The configurations of all the methods studied are described in the following.

- **Direct Prompting**: The model answers the question without any retrieved context. This baseline evaluates its internal knowledge and serves as a reference for measuring gains from retrieval-augmented generation (RAG) or agent-based methods.
- **Function Chunking RAG** (Wang et al., 2024b): This approach partitions code based on semantic boundaries, parsing the repository into function-level chunks. The embedding model used is `voyage-code-3` (VoyageAI, 2024) with 2048

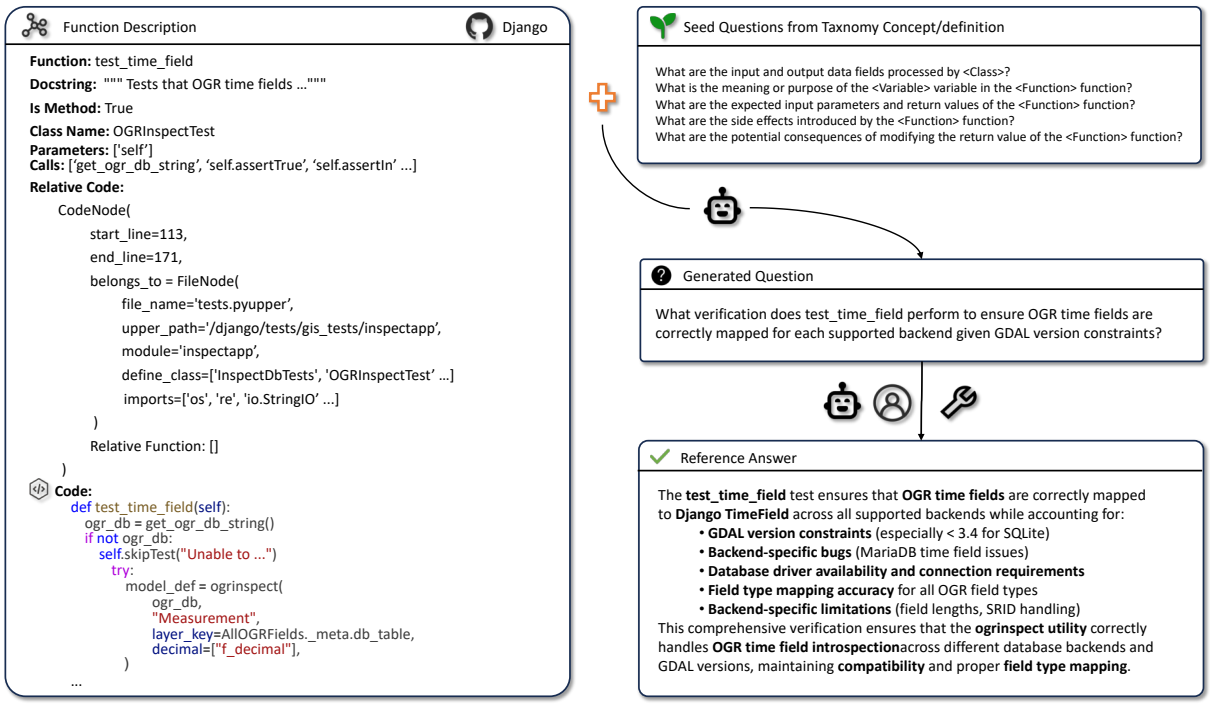

Figure 6: Illustrating question instantiation process.

dimensions. A Top-K setting with K=10 is adopted for the retrieval.

- **Sliding Window RAG** (Zhang et al., 2023): This method employs a sliding window to extract code snippets by dividing lengthy files into overlapping segments with a chunk size of 500 lines and an overlap of 100 lines. The embedding model used is the same in Function Chunking RAG. A Top-K setting with K=10 is adopted for the retrieval.
- **OpenHands** (Wang et al., 2024a): This method uses openhands-sdk[5], with the agent set to `get_default_agent` (including tools such as *terminal*, *file_editor*, *task_tracker*, *finish*, *think*, etc.).
- **SWE-agent** (Yang et al., 2024): This method adapts the original agent, which was designed to handle Issues, to QA tasks, retaining the original agent's action space and memory structure.
- **Commercial Tools**: Tongyi Lingma uses its proprietary model; Cursor runs in its default "auto" mode that automatically selects the best model based on the user query with built-in retrieval and orchestration. These commercial tools are evaluated to reflect the performance of current state-of-the-art closed-source solutions.

[5] https://github.com/OpenHands/software-agent-sdk, version 1.1.0

Unless otherwise specified, OpenHands and SWE-agent is configured with a maximum of **10** reasoning–action iterations per question. We set all LLM decoding temperatures to 0 to avoid the influence of randomness. All experiments were conducted on a system equipped with an Intel Xeon Gold 6254 CPU and an NVIDIA A100-80G GPU. The implementations of all methods are publicly available at https://github.com/peng-weihan/SWE-QA-Bench/Scripts/.

B.2 LLM-as-Judge Evaluation Protocol

Given a model output and the corresponding gold answer, we use Claude Sonnet 4.5 (Anthropic, 2025) as an automatic judge to assess answer quality along five dimensions: (1) *correctness*, factual accuracy; (2) *completeness*, coverage of all aspects of the question; (3) *relevance*, pertinence to the given question; (4) *clarity*, structural organization and readability; and (5) *coherence*, logical coherence of the reasoning process.

To improve reliability, each instance is evaluated five times per dimension, with majority voting used to determine dimension-level scores that are then aggregated into a single instance-level score. To mitigate bias, candidate systems are anonymized and answer orders are randomly shuffled across trials. The judge model is fixed, distinct from all candidate models, and never evaluates its own outputs. Overall, this protocol provides a robust assessment

Repository	# Files	# Classes	# Functions	# LOC	# Questions	Avg. Words (Q)	Avg. Words (A)
astropy	964	1,909	16,264	402,824	48	32.88	175.42
django	2,845	7,240	28,355	499,240	48	30.56	362.44
flask	83	64	829	18,108	47	25.66	284.62
matplotlib	905	968	9,941	266,896	48	29.85	285.46
pylint	2,308	2,287	6,746	117,602	48	26.33	268.46
pytest	260	491	5,151	100,111	48	29.10	277.19
requests	36	70	598	11,248	48	29.29	266.73
scikit-learn	982	764	10,360	424,550	48	29.94	283.75
sphinx	743	1,064	6,841	142,146	48	28.48	278.02
sqlfluff	392	2,089	1,854	145,382	48	27.52	287.48
sympy	1,584	1,997	33,994	779,192	48	26.19	268.81
xarray	233	601	7,902	186,039	48	30.88	290.96
conan	1,074	814	6,893	174,161	48	28.02	507.92
reflex	382	974	3,005	98,312	48	27.52	76.46
streamlink	509	1,190	3,671	86,593	48	27.08	85.83
Overall	**13,300**	**22,522**	**142,404**	**3,452,404**	**720**	**28.62**	**266.64**

Table 6: Statistics of SWE-QA. The benchmark comprises 720 questions spanning over 3.4M lines of code across 15 repositories, with answers averaging 266.6 words

Model	Evaluation Metrics					Overall
	Correctness	Completeness	Relevance	Clarity	Reasoning	
GPT-5.1	8.75	7.52	18.02	18.12	14.02	66.43
+ Function Chunking RAG	10.22	10.54	18.20	18.30	16.20	73.46
+ Sliding Window RAG	10.05	10.67	18.42	18.40	16.08	73.62
+ SWE-agent	13.54	14.07	18.66	18.50	17.02	81.79
+ OpenHands	13.88	13.98	18.78	18.47	17.22	82.33

Table 7: Human evaluation results. The result shows high agreement with LLM-as-a-Judge, validating the reliability of automated assessment methods.

of both semantic accuracy and logical coherence. The complete LLM-as-Judge prompt is provided in Prompt 3.

C Additional Experimental Results

C.1 Human Evaluation

While the LLM-as-Judge approach offers a scalable evaluation method, it is susceptible to inherent biases. To complement our automated metrics and obtain a more reliable assessment of answer quality, we conduct a human evaluation. Specifically, we recruited three professional software engineers, each with over three years of development experience, who are not co-authors of this paper. For each question, we presented the participants with the reference answer and the answers generated by fives approaches based on the GPT-5.1 model. To ensure fairness, answers were randomized and participants were blind to the generating approach. Each participant was asked to rate the answers on a 20-point scale across the same five dimensions used in our automated evaluation (Section 3.1). This experimental design is consistent with established practices in related research (Geng et al., 2024; Liu et al., 2023).

The results of our human evaluation are presented in Table 7. From the results, although the scores given by human evaluators are generally higher than those from the LLM-as-a-Judge, the overall trends remain highly consistent: both RAG methods show substantial improvement over Direct prompting, and the Agent Framework provides further gains.

C.2 Cost Analysis

In our experiments, we measured the cost of each model across different methods in terms of token usage, as shown in Table 8. Compared to direct responses, RAG incurs over ten times more token consumption. Most notably, agent frameworks exhibit an order-of-magnitude increase, consuming

Model	Qwen3-Coder-30B-A3B-Instruct	Qwen3-Coder-480B-A35B-Instruct	Kimi K2	GLM-4.6	Gemini 2.5 Pro	GPT-5.1	Average
Direct	94/63	94/65	94/34	94/76	94/51	94/87	94/63
Function Chunking RAG	3,042 / 64	3,042 / 106	3,042 / 92	3,042 / 240	3,042 / 82	3,042 / 196	3,042 / 130
Sliding Window RAG	6,302/73	6,302/112	6,302/110	6,302/233	6,302/99	6,302/201	6,302/138
SWE-agent	133,293/6,026	165,206/4,546	154,295/1,398	70,870/1,902	111,652/6,793	122,833/6,100	126,026/4,627
OpenHands	115,879/2,366	108,703/2,128	101,594/1,787	42,433/2,158	74,368/1,116	69,192/2,127	87,045/1,930

Table 8: Token Usage per Question (Input/Output). Direct prompting consumes the fewest tokens, while agent-based methods such as SWE-agent and OpenHands require significantly more input and output tokens. This highlights the trade-off between model reasoning complexity and token consumption.

approximately 100× more tokens than direct responses (and 10× more than RAG).

However, this massive cost discrepancy highlights a critical cost-performance trade-off in repository-level reasoning. Taking GLM-4.6 as an example, employing OpenHands instead of Function Chunking RAG improves Correctness from 9.48 to 11.91 and Completeness from 7.52 to 12.70. This substantial improvement—particularly the nearly 70% increase in Completeness—indicates that while RAG methods are cost-efficient, they often fail to comprehensively capture all useful context for complex, multi-hop repository queries. Therefore, for practitioners, choosing between RAG and agent frameworks requires carefully balancing budget constraints against the necessity for exhaustive code comprehension.

Moreover, different LLMs exhibit significant variation in token usage even when using the same agent framework, which is likely related to their intrinsic tool-calling efficiency and planning capabilities.

D Case Study

To provide deeper insights into the limitations of current LLMs and reveal common failure patterns in repository-level code comprehension, we conduct a detailed error analysis. Table 9 illustrates a representative case study involving the integration of Sphinx Glossary Term Validation, generated by the best-performing configuration (GPT-5.1 combined with OpenHands).

As shown in the table, the model is capable of identifying high-level system architectures and trigger mechanisms (e.g., `pytest.mark.sphinx` and the transform pipeline), which earns it respectable scores in Relevance and Clarity. However, it severely lacks analysis granularity. The generated response fails to capture the core components handling the translation (`i18n`) and completely ignores critical, multi-hop implementation details such as the `_NodeUpdater` class and the `compare_references()` method. As a result, its Correctness and Completeness scores are extremely low.

This case study exemplifies a prevalent issue: while modern LLM agents can grasp surface-level code structures, they frequently struggle to navigate and extract precise, deep dependency chains across multiple files. To further support benchmark users and practitioners, we have collected and annotated 100 typical error cases, which are available in our anonymous repository.

E Prompts Used in SWE-QA

Prompt 1: Extracting code questions from issues

```
You are given a GitHub issue from the {REPOSITORY_NAME}
repository. Extract or rewrite it into one or more
short, clear, concise questions about understanding the
{REPOSITORY_NAME} codebase, APIs, or system design.
Rules:
1. Only include questions answerable by code, logic, or
documentation.
2. Ignore bug reports, environment issues, or problems
that require fixing code.
3. Each question should ideally be ≤ 20 words.
IMPORTANT:
- Be STRICT in quality control: if the issue doesn't
contain meaningful questions about code understanding,
return an empty questions array.
- It's better to return no questions than to generate
low-quality or irrelevant questions.
- Only extract questions that genuinely help understand
the {REPOSITORY_NAME} codebase, APIs, or system design.
Input:
GitHub issue from {REPOSITORY_NAME} repository:
Title: {ISSUE_TITLE}
Body: {ISSUE_BODY}
Output JSON format:
{
issue_number: <number>,
questions: [{question: <string>}. . . ]
}
```

Prompt 2: Generating repository-level code questions

```
You are an expert software research assistant.
Given:
1. A function/class description extracted from a
software repository.
2. A list of seed questions as candidates from the
{CATEGORY} category.

Task:
1. Based on the seed questions and the function/class
description, generate **one single question** that is:
- As difficult and complex as possible,
- Requires multi-hop reasoning or deep technical
understanding,
- Not answerable by simple retrieval or direct lookup
(i.e., not solvable by basic RAG methods),
- Clearly related to the function/class description,
```

Case Study: Error Analysis of Sphinx Glossary Term Validation

Question: *"What is the integration mechanism between the test function that validates glossary term reference consistency and the translation processing transform system?"*

Reference Answer:
The mechanism is connected through the Sphinx docutils transform pipeline. It includes:
- **Test Setup:** The test is configured using the `@sphinx_intl` decorator.
- **Transform Registration:** The `setup()` in `sphinx/transforms/i18n.py` adds the `Locale` transform (priority 20) to the pipeline.
- **Glossary Validation:** The `Locale` transform's `_NodeUpdater.update_pending_xrefs()` specifically handles `refdomain='std'` and `reftype='term'` cases. It compares the number of glossary term references between the original and translated texts using `compare_references()`. If there is a mismatch, a warning of type `'inconsistent_references'` is generated.

Error Response Analysis

1. Insufficient Analysis Granularity
- **Issue:** In this specific response, the model stays at a high-level system architecture level (e.g., mentioning `SphinxTransformer`, `app.add_transform()`, and `SphinxTestApp`), completely ignoring the core components that handle translation (i18n) and consistency validation.
- **Missing Details:** The generated answer lacks descriptions of the `Locale` transform, `_NodeUpdater` class, and key methods like `update_pending_xrefs()` and `compare_references()`. It also fails to mention the warning type generated, such as `'inconsistent_references'`.

2. Right Direction, Lacking Depth
- **Issue:** The model correctly points out that the test is indirectly triggered by `pytest.mark.sphinx` and `app.build()` through the transform pipeline (which earns it high clarity and relevance scores). However, at the implementation level, it wrongly attributes the responsibility to a general `SphinxDomains` instead of the specialized i18n pipeline.
- **Result:** Due to the lack of specific class names and method-level evidence in this response, the correctness score (5/20) and completeness score (4/20) evaluated by the judge are extremely low.

Table 9: A representative case study of an incorrect response generated by GPT-5.1 paired with OpenHands. Although the model successfully identifies high-level interactions, its specific answer completely misses the critical, multi-hop implementation details required for repository-level comprehension.

```
- Technically precise and detailed,
- Reflects the style and intent of the original seed
questions but goes significantly deeper,
- **Must not be a compound question** (e.g., no use of
!and!, !or!, or comma-based subquestions),
- **Must be not too long and syntactically simple**,
- **Must be specific to the {CATEGORY} category**.

2. The question should encourage advanced analysis,
integration of multiple concepts, or insight beyond
surface-level information.
3. Output only the single refined question without
additional explanation or commentary.

Input:
1. Function/Class Description:
{DESCRIPTION}
2. Seed Questions from {CATEGORY}:
{SEED_QUESTIONS}
```

Prompt 3: LLM-as-a-Judge

```
You are a STRICT and RIGOROUS evaluator. You must rate
the candidate answer STRICTLY against the reference
answer. Be CONSERVATIVE with high scores - only award
high scores (16-20) when the candidate answer is truly
excellent and closely matches the reference answer in
quality and content.

CRITICAL EVALUATION PRINCIPLES:
1. Compare the candidate answer DIRECTLY with the
reference answer point by point
2. Any deviation, omission, or inaccuracy should
result in score reduction
3. High scores (16-20) should be RARE - reserve them
only for answers that are nearly perfect
4. Be strict about factual accuracy - even minor errors
should lower the correctness score
5. Missing key points from the reference answer should
significantly reduce completeness score
6. Vague or imprecise language should lower clarity
scores
7. When in doubt between two score ranges, choose the
LOWER one

Evaluation Criteria and Scoring Guidelines (each scored
1 to 20, total score 100):
1. Correctness (STRICT - penalize any inaccuracies):
20 — ONLY if completely correct with ALL core points and
details accurate, matching reference answer precisely
16-19 — Mostly correct but must have only TRIVIAL
inaccuracies; any noticeable error reduces to 15 or
below
12-15 — Partially correct; has some errors or omissions
that affect understanding; main points may be accurate
but details are wrong
8-11 — Several errors or ambiguities that significantly
affect understanding of core information
4-7 — Many errors; misleading or fails to convey key
information correctly
1-3 — Serious errors; completely wrong or misleading
2. Completeness (STRICT - penalize missing
information):
20 — ONLY if covers ALL key points from reference
answer without ANY omission; must match reference in
depth
16-19 — Covers most key points but missing some
non-trivial information; minor omissions are
acceptable
12-15 — Missing several important key points; content
is noticeably incomplete compared to reference
8-11 — Important information largely missing; content
is one-sided or superficial
4-7 — Covers very little relevant information;
seriously incomplete
1-3 — Covers almost no relevant information; completely
```

```
incomplete
3. Relevance (STRICT - penalize off-topic content):
20 — ONLY if content is fully focused on question topic
with NO irrelevant information whatsoever
16-19 — Mostly focused but may have minor peripheral
information; any significant off-topic content reduces
score
12-15 — Generally on topic but contains some off-topic
content that detracts from answer
8-11 — Topic not sufficiently focused; contains
considerable off-topic or tangential content
4-7 — Content deviates from topic; includes excessive
irrelevant information
1-3 — Majority of content irrelevant to the question
4. Clarity (STRICT - penalize unclear expression):
20 — ONLY if language is exceptionally fluent, clear,
and precise; very easy to understand without any
ambiguity
16-19 — Mostly fluent and clear but may have minor
unclear points; any significant ambiguity reduces
score
12-15 — Generally clear but some expressions are unclear
or not concise; may require effort to understand
8-11 — Expression somewhat awkward; has ambiguity or
lacks fluency that hinders understanding
4-7 — Language obscure; sentences are not smooth;
significantly hinders understanding
1-3 — Expression confusing; very difficult to
understand
5. Coherence (STRICT - penalize weak logic):
20 — ONLY if the reasoning is exceptionally clear,
logical, and well-structured; argumentation is
excellent and matches reference quality
16-19 — The reasoning is clear and logical with solid
argumentation; minor logical gaps may exist
12-15 — The reasoning is fairly reasonable but has
noticeable logical jumps or organization issues
8-11 — The reasoning is average; has logical jumps or
organization problems that affect understanding
4-7 — The reasoning is unclear; lacks logical order;
difficult to follow
1-3 — No clear reasoning; logic is chaotic

INPUT:
Question:{question}
Reference Answer:{reference}
Candidate Answer:{candidate}

OUTPUT:
Please output ONLY a JSON object with 5 integer fields
in the range [1,20], corresponding
to the evaluation scores:
{
"correctness": <1-20>,
"completeness": <1-20>,
"relevance": <1-20>,
"clarity": <1-20>,
"cohenrence": <1-20>
}
SCORING INSTRUCTIONS:
- Read the reference answer carefully and identify ALL
key points, details, and structure
- Compare the candidate answer systematically against
the reference answer
- For each criterion, start with a conservative score
and only increase if the candidate truly deserves it
- If the candidate answer is significantly shorter,
less detailed, or less precise than the reference,
reduce scores accordingly
- If the candidate answer contains information not
in the reference (unless it's clearly relevant and
accurate), consider reducing relevance score
- When scoring, ask yourself: "Does this candidate
answer match the quality and completeness of the
reference answer?" If not, reduce scores
- Average or mediocre answers should receive scores in
the 8-15 range, not higher
- Only truly excellent answers that closely match the
reference should receive 16-20 scores

REQUIREMENT:
No explanation, no extra text, no formatting other than
valid JSON. Be strict and conservative with your scores.
```