SWE-QA: A Dataset and Benchmark for Complex Code Understanding

Laïla Elkoussy, Julien Perez[*,†]
LRE, EPITA
laila.elkoussy@epita.fr

[†]Bpifrance
julien.perez@bpifrance.fr

Abstract

In this paper, we introduce SWE-QA, a text and code corpus aimed at benchmarking multi-hop code comprehension, addressing the gap between simplified evaluation tasks and the complex reasoning required in real-world software development. While existing code understanding benchmarks focus on isolated snippets, developers must routinely connect information across multiple dispersed code segments. The dataset comprises 9,072 multiple-choice questions systematically generated from 12 Python repositories of SWE-bench, evaluating several recurrent reasoning patterns like Declaration-and-Call questions that link entity definitions to their usage, and Interacting-Entity questions that examine the dynamic relationships among multiple collaborating components. Generated through parsing-based entity extraction and Large Language Model assisted question construction with carefully validated distractors, the benchmark distinguishes genuine comprehension from superficial pattern matching. Evaluation of 15 language models (360M to 671B parameters) reveals significant challenges in multi-hop reasoning, with best performance reaching 74.41% accuracy. Dense architectures consistently outperform mixture-of-experts models by 10-14 percentage points, while reasoning-enhanced variants show inconsistent benefits.

Keywords: Corpus (Creation, Annotation, etc.), Information Extraction, Information Retrieval, Question Answering

1. Introduction

Code comprehension remains a fundamental challenge in natural language processing for software engineering, with implications extending from academic research to practical software development and automated programming assistance. While significant progress has been made in developing language models for code understanding, capturing syntax, static semantics, and even dynamic behavior (Ma et al., 2023), existing evaluation frameworks predominantly focus on localized reasoning tasks. Such frameworks often fail to capture the complex, project-scale, interconnected nature of real-world software systems, where multi-hop reasoning, interprocedural dependencies, and runtime behavior matter deeply, which is essential for code development agents (She et al., 2023; Zhang et al., 2023).

Current benchmarks typically evaluate models on isolated code snippets or self-contained problems where all necessary information is available within a single context. This misrepresents how software engineers interact with large codebases. Real-world software understanding requires reasoning across multiple, disparate code segments, a process similar to multi-hop reasoning in reading comprehension tasks like HotpotQA (Yang et al., 2018), but with added complexity from syntax, execution semantics, and dependencies.

Consider a typical developer workflow: encountering a function call in one file, navigating to its definition in another file to understand parameters and behavior, then tracing how returned values are used elsewhere across modules. This requires synthesizing information from multiple sources, understanding temporal dependencies, and reasoning about data flow patterns, skills that current benchmarks largely fail to assess. Multi-hop code comprehension arises in linking function declarations to invocations across files, tracing class hierarchies, following data pipelines, understanding event-driven architectures, and comprehending complex object interactions.

Despite its importance, existing datasets provide insufficient coverage. Benchmarks like CodeQA, CS1QA, and CodeXGLUE primarily evaluate single-hop reasoning within contained contexts. Even sophisticated benchmarks like SWE-bench, while operating on real repositories, focus on single-issue resolution rather than explicit multi-entity tracking. This gap limits assessment of language models for real-world software engineering. Multi-hop reasoning over large codebases also highlights a limitation of current large language models: finite context windows. Even with extended context, models often struggle to maintain coherence when given excessive or poorly scoped information (Liu et al., 2023; Xiao et al., 2024). This underscores the need for evaluation settings that control the amount and structure of context, ensuring performance reflects

Proceedings of the Fifteenth Language Resources and Evaluation Conference (LREC 2026), pages 1236–1251
11–16 May 2026. ©ELRA Language Resources Association (ELRA), 2026

lib/mpl_toolkits/axes_grid1/axes_size.py - Entity Definition

```python
class Scaled(_Base):
    """
    Simple scaled(?) size with absolute part = 0 and
    relative part = *scalable_size*.
    """
    def __init__(self, scalable_size):
        self._scalable_size = scalable_size
    def get_size(self, renderer):
        rel_size = self._scalable_size
        abs_size = 0.
        return rel_size, abs_size
Scalable = Scaled
def _get_axes_aspect(ax):
    aspect = ax.get_aspect()
    if aspect == "auto":
        aspect = 1.
    return aspect
```

galleries/examples/axes_grid1/demo_fixed_size_axes.py - Entity Usage

```python
# The first & third items are for padding and the second items are for the
# Axes. Sizes are in inches.
h = [Size.Fixed(1.0), Size.Scaled(1.), Size.Fixed(.2)]
v = [Size.Fixed(0.7), Size.Scaled(1.), Size.Fixed(.5)]
divider = Divider(fig, (0, 0, 1, 1), h, v, aspect=False)
# The width and height of the rectangle are ignored.
ax = fig.add_axes(divider.get_position(),
                  axes_locator=divider.new_locator(nx=1, ny=1))
ax.plot([1, 2, 3])
plt.show()
```

Question Metadata

Repo: matplotlib/matplotlib
Question ID: 133
Category: entity_declaration_call_specific
Entity: `Scaled`
Correct Answer: C

MCQ Question

Question: How does the `Scaled` class handle the case where the absolute part of the size is zero, and what implications does this have for the plot's layout?

A) When the absolute part of the size is zero, the `Scaled` class returns `abs_size = 1`, ensuring that the plot always has a minimum size.

B) The `Scaled` class handles the case where the absolute part of the size is zero by setting it to `None`, effectively removing the size from the plot.

C) When the absolute part of the size is zero, the `Scaled` class returns `abs_size = 0`, effectively ignoring the absolute part of the size. This can lead to distorted plots, especially when the aspect ratio is not suitable for the data being represented.

D) The `Scaled` class uses a heuristic to handle the case where the absolute part of the size is zero, scaling the size to a default value of 1.0.

Figure 1: Multi-hop question sampled from SWE-QA requiring cross-file reasoning: the `Scaled` class in `axes_size.py` (top left) and its use in `demo_fixed_size_axes.py` (bottom left). A correct answer must trace how `abs_size = 0` in `get_size` affects layout when `Size.Scaled(1.)` is invoked. The multiple-choice design uses targeted distractors to distinguish true cross-context understanding from superficial pattern matching, mirroring the multi-step reasoning developers perform in real codebases.

reasoning ability rather than context length.

We address this gap by introducing a novel dataset and benchmark[1] designed to evaluate multi-hop, repository-scale code comprehension. Our approach systematically constructs questions requiring understanding of complex relationships between code entities across different file segments, mimicking the reasoning processes developers use in large-scale software systems. We focus on two fundamental categories: declaration-and-call relationships connecting entity definitions with usage contexts, and interacting entity relationships requiring understanding of how multiple components collaborate. By examining multi-hop scenarios from real repositories, we bridge the gap between simplistic evaluation tasks and real software comprehension, offering an authentic assessment of the cognitive challenges faced by developers and automated systems.

Finally, we leverage a systematic methodology for generating authentic multi-hop code comprehension questions from real software repositories, ensuring relevance, quality, and diversity. Based on this, we curate a dataset with two categories of multi-hop questions and carefully designed distractors, providing a challenging benchmark that tests true understanding rather than memorization. We evaluate multiple state-of-the-art language models on this benchmark, revealing significant performance gaps and highlighting strengths and weaknesses in handling complex code relationships, with attention to scaling effects on multi-hop reasoning. We analyze common failure patterns and error types, offering actionable insights for improving model design and training strategies. Finally, we explore the relationship between model size, architecture, and reasoning performance, providing guidance for practitioners optimizing language models for complex software engineering tasks.

Our contributions are threefold. First, we present SWE-QA, a curated dataset with diverse, high-quality multi-hop questions and carefully designed distractors, enabling robust assessment of genuine code understanding beyond superficial pattern matching. Second, we introduce a systematic methodology for generating this dataset from real repositories, ensuring practical relevance. Third, we evaluate state-of-the-art language models on this benchmark, revealing performance gaps, common failure patterns, and insights for improving multi-hop reasoning over code.

2. Related Work

Code comprehension research has primarily focused on single-hop reasoning within isolated code snippets or simple queries. CodeQA (Liu et al., 2021) provides Java and Python Q&A pairs from snippets, while CS1QA (Sohn et al., 2022) collects Q&A from introductory programming courses. These datasets emphasize local understanding without requiring reasoning across dispersed code elements. CRUXEval (Gu et al., 2024) contains 800800 800 short Python functions for assessing reasoning, understanding, and execution through two tasks: CRUXEval-I (input prediction) and CRUXEval-O (output prediction). Each function, generated with Code Llama 34B and filtered for

[1] Dataset available at: https://github.com/lailanelkoussy/swe-qa

human solvability within a minute, includes input-output examples. Although it tests execution tracing more deeply than prior benchmarks, it remains limited to isolated function-level reasoning.

Code World Models (team et al., 2025) train language models on observation-action trajectories from Python environments to improve execution understanding through world modeling, yet they do not address tracing dependencies across multiple code segments in large repositories. Broader benchmarks such as CodeXGLUE (Lu et al., 2021), HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and SWE-bench (Jimenez et al., 2024) evaluate realistic programming tasks like completion, translation, and patch generation, but still lack multi-element reasoning. LocAgent (Chen et al., 2025) addresses the related problem of code localization—identifying where in a codebase changes must be made—by parsing repositories into directed heterogeneous graphs that capture files, classes, functions, and their dependencies (imports, invocations, inheritance), and leveraging LLM agents to navigate these graphs via multi-hop reasoning. While LocAgent demonstrates strong performance on file-level localization and downstream issue resolution, it is task-specific and does not provide a comprehension benchmark for evaluating cross-context understanding across diverse reasoning patterns. In natural language processing, multi-hop datasets such as HotpotQA (Yang et al., 2018), MuSiQue (Trivedi et al., 2022), WikiHop, and WikiMultihopQA (Welbl et al., 2018) combine information across sources. Applying this to code is difficult due to syntax, dependencies, and execution semantics. Table (**?**) in Appendix A summarizes the key dimensions along which existing benchmarks differ, highlighting the gap that SWE-QA is designed to fill. Our work bridges this gap by building a multi-hop code comprehension dataset that extends multi-step reasoning to software repositories, evaluating models on tracing logical dependencies and entity interactions across repository-scale codebases.

3. Methodology

Our approach comprises three steps: code processing and chunking, multi-hop question generation, and quality control through post-processing and benchmarking. This pipeline produces diverse, high-quality questions that test complex code understanding in realistic software engineering contexts.

3.1. Code Processing and Chunking

Repository Selection. We select 12 Python repositories from the SWE-bench dataset to cover diverse domains, as web frameworks, data processing, scientific computing, utilities, and varying code complexity. Preliminary analysis of entity density, file structure, and cross-file dependencies informed chunking parameters.

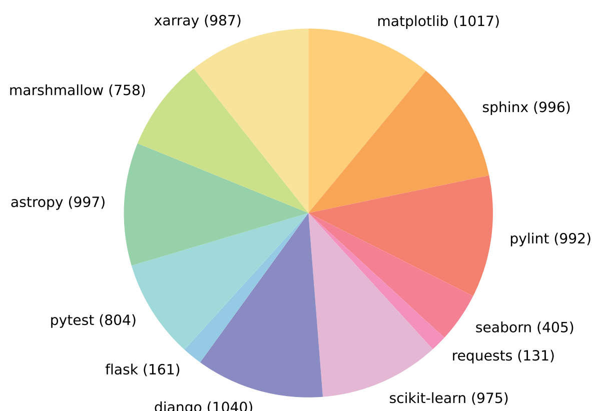

Figure 2: Distribution of questions of SWE-QA across public repositories. The repositories are 12 open source GitHub repositories that each contains the source code for a popular, widely downloaded PyPI package.

Text Segmentation. We used LangChain's recursive character splitter with Python-aware separators to preserve syntactic boundaries. Chunks of 1000 characters with zero overlap balance context and efficiency, typically keeping functions or classes intact. Separators follow a hierarchy: double newlines, class/function definitions, control structures, single newlines with indentation, and commas. Boundaries preserve syntactic integrity to avoid broken statements.

Entity Extraction. We built an entity extraction pipeline based on Abstract Syntax Trees (AST) to systematically analyze Python source code and identify structural and semantic elements. Using Python's built-in AST parser, the system detects declared entities such as classes, functions, methods, variables, and constants, capturing attributes including type, scope, inferred data type, and defining context. Function and method invocations are recorded separately as called entities, distinguishing definitions from usages.

This dual representation enables reconstruction of call graphs, inheritance hierarchies, and data dependencies. Extracted entities are mapped back to their corresponding code chunks through structural and lexical matching, allowing each chunk to reference the entities it declares and calls. The resulting bidirectional mapping links entities and

chunks even when definitions and usages span multiple segments, forming a semantic graph that supports higher-level analyses of dependency and modular structure across the codebase.

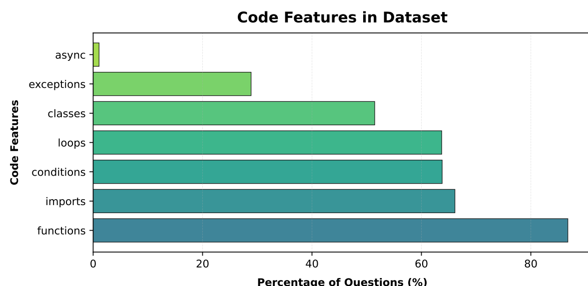

Figure 3: Ratios of code chunks showing the presence of specific programming constructs (loops, conditions, functions, classes, imports, async operations, and exceptions).

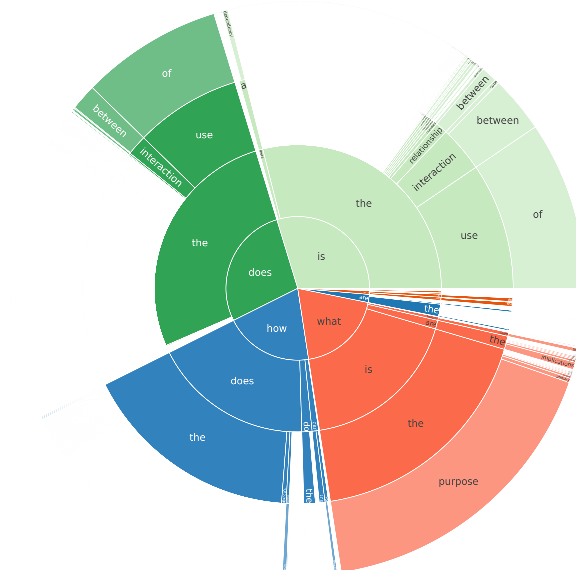

Figure 4: Distribution of the first four words of all questions, representing their frequency patterns. Empty colored blocks indicate suffixes that are too rare to show individually.

3.2. Question Categories and Generation

Multi-Hop Taxonomy. Inspired by the HotpotQA dataset, we have defined two multi-hop question types. Declaration-and-Call (DC) questions connect an entity's definition in one chunk with its usage in another, requiring reasoning about parameters, return values, state changes, or error handling. Interacting Entity (IE) questions involve three chunks where two entities interact, demanding analysis of data flow, execution order, shared state, or collaborative behaviors.

Chunk Sampling. Candidate pairs for DC and triplets for IE were enumerated across repositories and validated for genuine multi-hop relationships. Selection enforces diversity: limiting questions per entity or entity pair, balancing repositories by size and complexity, and ensuring a mix of simple and complex patterns. We capped each repository at 600 questions per category and applied quality thresholds for relationship strength and code complexity.

3.3. Question and Answer Generation

Question Generation Process. We used `Meta-Llama/Llama-3.2-3B-Instruct`, tuned for creative but precise output, with prompts specifying context, question type, and multi-hop reasoning requirements. An iterative refinement checked clarity and multi-hop validity. Semantic analysis ensured structural diversity, varied vocabulary, and appropriate complexity.

Answer Generation. Answers were generated by the same model used for question creation, conditioned on the question and relevant code chunks, then refined for conciseness through a follow-up call. Prompt validity was verified by human review on a subset before scaling to the full dataset.

Distractor Creation and Stylistic Alignment. Three distractors per question were generated in a single call conditioned on the question, code, and correct answer, ensuring technical plausibility while reflecting common misconceptions. After validating format compliance and cardinality, correct answers were adapted to match the linguistic style of distractors, preventing surface-level pattern matching. All generation and adaptation used the same model, with human-verified prompts guiding each stage.

4. Experimental Setup

We evaluate a collection of language models on SWE-QA to assess their multi-hop code comprehension capabilities and the pertinence of our corpus. The evaluated models span from small to large scales and include two `SmolLM2` variants, `Llama-3.3-70B-Instruct`, and `DeepSeek-R1`. To isolate the effect of reasoning on multi-hop question answering, we include both reasoning and non-reasoning variants of the same models, such as `Phi-4-mini` and `Qwen3-4B`. All evaluations are conducted in a zero-shot setting: models receive only the relevant code chunks, the question, and

the multiple-choice options—without any additional in-context examples. SWE-QA comprises **9,072** questions drawn, including **4,584** DC questions and **4,488** IE questions. We report overall accuracy, category-specific accuracy, repository-level performance, and retrieval quality metrics such as NDCG@k and Precision@k, where applicable. Post-processing scripts normalize model outputs to extract the selected option, addressing cases where responses include explanations or formatting variations.

Oracle Question Answering. In the first setting, models are evaluated with an *oracle retrieval*. For each question, the code chunks that are relevant are solely provided, without any distractors. Models must return a single answer based on these oracle-provided chunks, the question, and the multiple-choice options. This setting establishes an upper bound on comprehension performance by removing retrieval errors.

Retrieval-Based Evaluation. The second setting evaluates retrieval performance. All code chunks of each repository are embedded using the `Salesforce/SFR-Embedding-Code-400M_R` (Liu et al., 2025) model which is optimized for code/text retrieval. For each question, a maximum inner product search is performed on the repository's vector database to retrieve the ten most relevant chunks. Retrieval quality is measured using NDCG@k and Precision@k, providing insight into the difficulty of locating relevant information compared to the ideal oracle scenario.

Noisy Oracle Comprehension. Finally, we test the best-performing models under a *noisy oracle* setting. Each question is presented with the relevant chunks plus a set of distractor chunks. These distractors are selected based on the most likely but irrelevant chunks retrieved in the second experiment. This setting evaluates whether models can still leverage the correct answer when the context is contaminated with retrieval-induced noise, better reflecting real-world code understanding pipelines.

Dataset Validation and Correction LLM-based answer generation occasionally produces incorrect labels. To improve label quality, we employed consensus-based validation on the oracle question answering results. We excluded three small models (SmolLM2-360M, SmolLM2-1.7B, DeepSeek-R1-Distill-Qwen-1.5B) to prevent capacity-related noise from corrupting consensus signals. Using two criteria: fewer than 3 models selecting the generated answer, and at least 11 of 12 retained models agreeing on an alternative. This identified 66 candidate mislabeled questions. Spot-check validation on 10 randomly sampled questions confirmed that the consensus answer was correct in all cases (100% precision), justifying batch correction of all 66 questions. All results reported below reflect corrected ground truth labels, ensuring performance metrics reflect genuine code comprehension rather than label artifacts.

5. Experiments

All results reported below reflect corrected ground truth labels after applying our validation procedure described in Section 4.

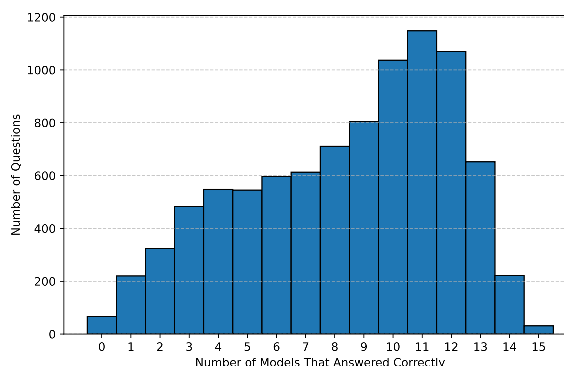

Figure 5: Histogram showing question difficulty, measured by the number of models that answered each question correctly. Fewer correct responses indicate higher difficulty.

5.1. Oracle Question Answering

Overall Performance. Across 15 models, accuracy spans 74.41%–19.94%, highlighting the difficulty of multi-hop code reasoning. `Llama-3.3-70B-Instruct` leads with 74.41%, 2.4 points above `gemma-3-4b-it`, demonstrating that optimized smaller models can approach large-scale performance. `Qwen3-4B-Instruct` follows at 70.05%, while `Llama-3.2-3B-Instruct` shows rare balance across question types, hinting at architecture-driven inter-entity strengths. Figure 5 shows a broad distribution of question difficulty: nearly half of the questions are correctly answered by at least 10 models, while roughly one-tenth are solved by at most three, providing a meaningful spectrum to differentiate model capabilities.

Architecture and Reasoning Effects. The two MoE model families evaluated underperform their dense counterparts across scales. `gpt-oss-20b` variants cluster around 60% accuracy, 10 points below dense models like `gemma-3-4b-it`, with negligible variation across reasoning depths.

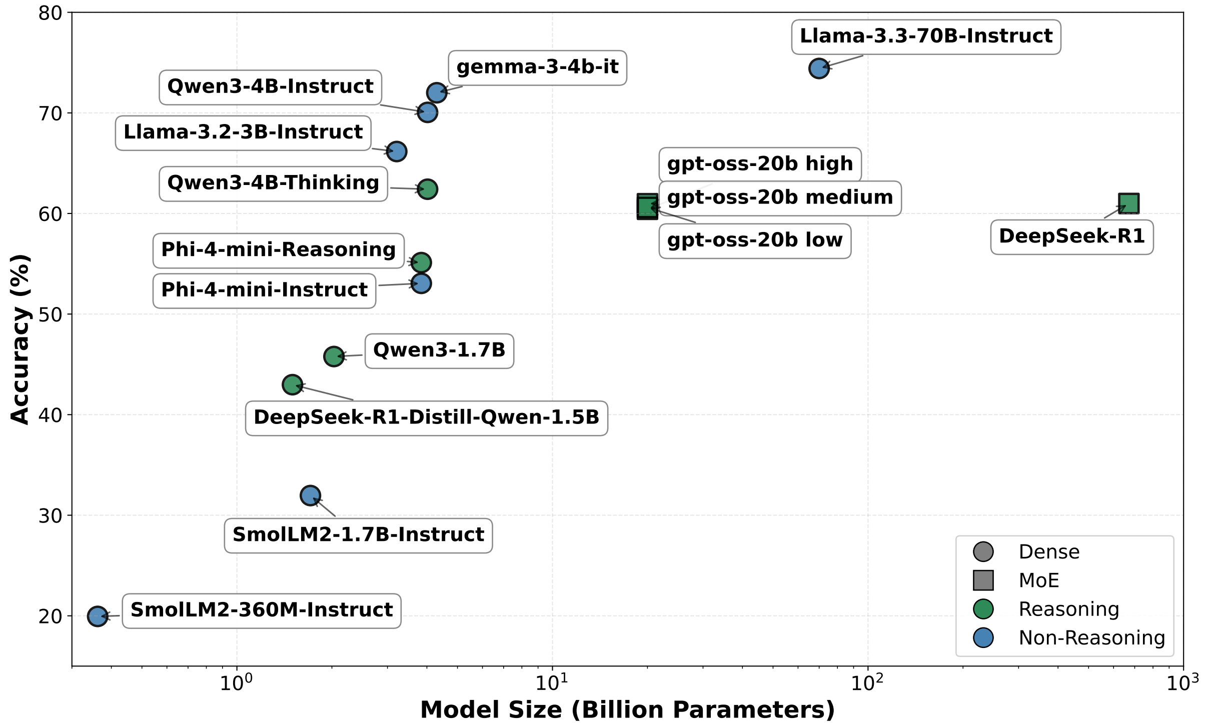

Figure 6: Model accuracy in the Oracle Question Answering setting as a function of parameter count. Circle markers denote dense architectures, squares indicate MoE models. Green corresponds to reasoning-tuned models, while blue represents standard instruction-tuned models.

DeepSeek-R1, despite 671B total and 37B active parameters, achieves only 60.98%, suggesting a potential scaling plateau not observed in the dense models evaluated. One possible explanation is that multi-hop reasoning demands unified knowledge access, while MoE routing may fragment information across expert boundaries, making cross-context integration more difficult.

Reasoning-enhanced variants show inconsistent benefits. Qwen3-4B-Thinking underperforms its instruct counterpart by 7.6 points, particularly on IE questions, indicating that extended reasoning at small scales may amplify error propagation. Phi-4-Mini gains only two points from reasoning, showing limited benefit under capacity constraints. Based on the models evaluated, neither extended reasoning nor larger MoE scale appears to consistently improve multi-hop comprehension, though broader evaluation across more model families would be needed to substantiate this observation.

Model	DC (%)	IE (%)
Llama-3.3-70B-Instruct	**78.75**	**69.98**
gemma-3-4b-it	75.32	68.60
Qwen3-4B-Instruct	74.58	65.41
Llama-3.2-3B-Instruct	66.79	65.50
Qwen3-4B-Thinking	69.85	54.81
gpt-oss-20b (medium)	67.95	53.83
DeepSeek-R1	68.23	53.58
gpt-oss-20b (low)	67.62	53.43
gpt-oss-20b (high)	67.51	53.16
Phi-4-Mini-Reasoning	59.77	50.35
Phi-4-Mini-Instruct	58.09	47.90
Qwen3-1.7B	48.32	43.18
DeepSeek-R1-Dist-Qwen-1.5B	46.72	39.14
SmolLM2-1.7B-Instruct	33.92	29.94
SmolLM2-360M-Instruct	19.28	20.61

Table 1: Category-level accuracy results for Oracle Question Answering, showing performance on DC and IE questions.

Category-Specific Patterns. As shown per Table 1, models consistently perform better on DC than IE questions, with gaps going from 5 to 15 points. Reasoning models such Qwen3-4B-Thinking, DeepSeek-R1, and gpt-oss-20b show the largest disparities, confirming that reasoning over interacting entities across three code locations is substantially harder than single-entity tracing. Top performers such as Llama-3.3-70B-Instruct, gemma-3-4b-it, and Qwen3-4B-Instruct exhibit smaller gaps (6–9 points), while Llama-3.2-3B-Instruct stands out with nearly balanced results, suggesting architectural features that enhance inter-entity reasoning even at small scale.

Metric	$k=3$	$k=5$	$k=10$
Precision@k	0.2288	0.1626	0.0961
Recall@k	0.3305	0.3875	0.4535
F1@k	0.2643	0.2248	0.1567
Hit Rate@k	0.5909	0.6620	0.7341
MRR@k	0.4945	0.5108	0.5206
NDCG@k	0.3430	0.3726	0.3998

Table 2: Retrieval-based evaluation results with $k = 3, 5, 10$.

Model	Overall (%)	DC (%)	IE (%)
Llama-3.3-70B	74.31 ▼ 0.1	77.94 ▼ 0.81	70.61 ▲ 0.63
gemma-3-4b-it	70.63 ▼ 1.37	73.97 ▼ 1.55	67.22 ▼ 1.38
Qwen3-4B-Instruct	68.79 ▼ 1.26	73.01 ▼ 1.57	64.48 ▼ 0.93

Table 3: Difference of model performance in Noisy Oracle Question Answering compared to Oracle Question Answering

Model Size vs. Performance Analysis. Performance scales with size but is conditioned by architecture and training methods. `Llama-3.3-70B-Instruct` leads, confirming benefits from scale, though `gemma-3-4b-it`'s near parity shows the power of optimization. `DeepSeek-R1`'s weak result despite 37B active parameters shows that scale alone is insufficient. The 3–4B range emerges as a performance "sweet spot," showing the widest variance driven by architecture and inference strategy. Below 2B parameters, accuracy collapses indicating a lower bound near 3B parameters of current models for effective multi-hop reasoning.

5.2. Retrieval-Based Evaluation

Table 2 shows moderate retrieval effectiveness. Precision@k declines from 0.23 ($k = 3$) to 0.10 ($k = 10$), while Recall@k increases from 0.33 to 0.45, reflecting the typical precision-recall tradeoff. The highest F1@k of 0.26 at $k = 3$ indicates smaller retrieval sets better balance relevance and noise. Hit Rate@k above 0.59 and stable MRR@k around 0.5 show relevant chunks frequently appear in top ranks, though NDCG@k (0.34–0.40) suggests limited ranking quality. Retrieval is sufficient for downstream tasks but leaves room for improvement.

5.3. Distractor and Oracle Comprehension

Table 3 reveals distinct robustness patterns under retrieval noise. `Llama-3.3-70B-Instruct` shows negligible degradation (0.04-point drop), with slight DC decline but improved IE performance, suggesting noise can aid distractor elimination. `gemma-3-4b-it` degrades modestly ($\sim$1 point) across categories, while `Qwen3-4B-Instruct` exhibits the largest decline yet retains strong accuracy. Noise tolerance scales with model size, though architecture also matters.

6. Discussion

6.1. Architectural Insights

MoE vs. Dense Models. In our experiments, the two MoE model families tested underperform their dense counterparts, potentially pointing to structural limitations in multi-hop reasoning, though caution is warranted given the small sample. `DeepSeek-R1` achieves only 60.98%, similar to `gpt-oss-20b` and below dense counterparts like `Llama-3.3-70B-Instruct` and `gemma-3-4b-it`. This parity across 4–37B active parameters may suggest a scaling plateau not observed in dense architectures, though differences in training data and optimization could equally account for the gap. Possible contributing factors include fragmented expert knowledge hindering cross-context integration, routing difficulties in activating coherent expert sets, or scaling inefficiency. The wider DC–IE gaps of approximately 14-15 pts in MoE models compared to 6–9 pts in dense models are consistent with this pattern, though they do not establish a causal architectural explanation.

Reasoning-Enhanced Inference. Reasoning capacity has not yet shown consistent benefits for multi-hop code comprehension. `Qwen3-4B-Thinking` underperforms its non-reasoning counterpart by 7.6 points, `Phi-4-Mini` gains only marginally, and `gpt-oss-20b` exhibits near-identical outcomes across reasoning intensities. These results indicate that extended reasoning, regardless of depth, does not reliably improve performance on this task.

6.2. Qualitative Analysis of Challenges

To better understand what makes certain questions particularly challenging, we conducted a qualitative analysis of failure patterns. We identify three primary axes of difficulty that characterize the reasoning requirements in multi-hop code comprehension. First, multi-entity reasoning refers to questions requiring simultaneous tracking of multiple interacting entities (classes, functions, variables) and their relationships across code segments. Second, multi-hop reasoning denotes questions demanding a sequential chain of logical steps connecting information across dispersed code locations, where each step depends on the previous one. Third, execution modeling encompasses questions necessitating mental simulation of code execution to

Question	Multi-Entity	Multi-Hop	Execution Model
Does the use of `set_ylim` in the `set_lim_and_transforms` method interact with the `tight_layout` parameter in a way that affects the appearance of the figure, and if so, what is the expected outcome?	`set_ylim`, `set_lim_and_transforms`, `tight_layout`, figure object	Setting axis limits triggers transform calculations, which affect layout adjustments and final figure appearance	Trace how axis limit changes propagate through the layout engine
How does the order of inheritance for class `D`, which inherits from both `B` and `C`, affect the type checking and instantiation of its subclasses `E` and `F`?	Classes `B`, `C`, `D`, `E`, `F`, and their inheritance relationships	Method Resolution Order (MRO) of `D` affects type checking for `E` and instantiation of `F`	Execute Python's MRO algorithm and track type resolution
Does the use of `filter_metadata_in_routing_methods` in the provided code introduce unnecessary dependencies or side effects that could impact the performance or behavior of the `ConsumingRegressor.fit` method?	Filter function, routing methods, metadata objects, `ConsumingRegressor`, `fit` method	Filtering metadata affects routing, which impacts the data available to `fit`	Trace metadata flow through filtering, routing, and consumption

Table 4: Examples of challenging questions requiring combinations of reasoning capabilities. Questions shown were answered correctly by at most two models, both top performers, demonstrating the difficulty frontier of current multi-hop code comprehension.

trace value transformations, state changes, or control flow across multiple components.

These axes are not mutually exclusive; the most challenging questions typically combine all three dimensions. To illustrate these reasoning requirements, we analyzed questions that were answered correctly by at most two models, and only by top performers (Llama-3.3-70B-Instruct or gemma-3-4b-it). This selection criterion eliminates questions likely solved by random guessing while focusing on tasks that remain challenging yet theoretically within reach of current capabilities.

Table 4 presents three representative examples exhibiting distinct patterns of complexity. The first question requires understanding how `set_ylim` method calls propagate through transform calculations to affect layout adjustments—exemplifying multi-hop reasoning through a rendering pipeline with moderate entity complexity. The second question demands analysis of Python's Method Resolution Order (MRO) across a five-class inheritance hierarchy, requiring precise execution modeling of type resolution alongside tracking multiple class relationships. The third question combines all three dimensions: tracking metadata flow through filtering and routing functions while modeling side effects on the `fit` method's behavior.

These examples reveal that model failures often stem not from inability to understand individual code constructs, but from difficulty maintaining coherent reasoning chains across multiple logical steps while simultaneously tracking entity states and simulating execution semantics. The architecture-specific performance gaps observed in our quantitative results (Section 6.1) likely reflect differential capabilities along these reasoning axes, with the MoE models in our sample tending to struggle more when entity tracking and execution modeling must be performed jointly, possibly reflecting routing constraints across expert boundaries.

7. Conclusion

We introduced SWE-QA, a benchmark for multi-hop code comprehension that evaluates language models on complex reasoning tasks across real software repositories. SWE-QA captures core challenges of large-scale code understanding and mirrors the cognitive demands faced by developers navigating extensive codebases.

The evaluation of fifteen models shows that multi-hop reasoning remains a major obstacle, with persistent weaknesses in handling dispersed and interdependent code contexts. Our analysis highlights three primary axes of difficulty: multi-entity reasoning, multi-hop reasoning, and execution modeling, revealing gaps in current architectures and training approaches.

Future work should expand human annotation to strengthen label reliability, explore systematic prompt design to isolate methodological effects, and extend SWE-QA to additional programming languages. Controlled comparisons across architectures, targeted fine-tuning, and human baselines can further clarify the limits of current models. Explicit modeling of entity relationships and execution dynamics may ultimately enable more structured and execution-aware reasoning.

Overall, this benchmark establishes a foundation for advancing study of complex code comprehension and guiding the development of models with deeper compositional reasoning capabilities.

8. Limitations

Dataset Construction. SWE-QA is restricted to Python repositories from SWE-bench and to 2–3 hop reasoning chains, limiting generalization to other programming languages and to deeper multi-hop scenarios. Complex code patterns such as asynchronous control flow, metaprogramming, and cross-module dynamic dispatch are largely absent, as they fall outside what static syntactic analysis can reliably model. Questions were generated by `Llama-3.2-3B-Instruct`; larger generation models may produce higher-quality or more diverse questions. Fixed 1000-character chunking may not optimally suit all code structures or context windows, and training data contamination for the evaluated models cannot be fully excluded.

AST-Based Generation Bias. Entity extraction relies on Python's built-in AST parser, which captures only statically analyzable, syntactically explicit relationships. As a result, benchmark questions are inherently grounded in the structural view of code that the AST provides: declared entities, explicit call sites, and syntactic inheritance chains. This introduces a systematic bias: models and systems that navigate code through similar structural or symbolic lenses may be artificially advantaged. In particular, DC questions, which trace a declaration to its call site, directly mirror the entity relationships that AST-based code navigation tools expose. Systems with access to AST-backed tooling, such as language servers or MCP-enabled tools offering *go-to-definition* and *find-references* capabilities, operate on the same representation used to generate the questions and therefore have an inherent structural advantage. Conversely, dynamic code behaviors invisible to static parsing, including runtime polymorphism, decorator-induced modifications, and reflective patterns, are systematically underrepresented. Results should therefore be interpreted with the caveat that the benchmark measures comprehension of statically identifiable code structure, and comparisons involving AST-tool-assisted systems may not reflect genuine differences in code understanding.

Evaluation Protocol. All models were evaluated in zero-shot settings without systematic prompt engineering, potentially underestimating models sensitive to prompt format or those that benefit from structured reasoning. Though `gpt-oss-20b` was tested across thinking intensities, other models used default settings and reasoning hyperparameters were not exhaustively explored. Retrieval used a single embedding model, leaving open whether alternative retrieval strategies would change the relative difficulty of the retrieval-based setting.

Architectural Conclusions. MoE underperformance was observed on only two model families, limiting the strength of architectural generalizations. Correlations between architecture and performance do not establish causality: observed differences may reflect training data composition, optimization choices, or scale rather than fundamental architectural constraints. Claims about MoE limitations should therefore be treated as preliminary observations warranting controlled investigation rather than definitive conclusions.

9. Ethical Considerations

This work uses publicly available code repositories and focuses on advancing code comprehension capabilities. All code processing and analysis respect the original licenses and usage terms of the source repositories.

10. Bibliographical References

Jacob Austin et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, pages 16568–16621. PMLR.

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

Jade Liu et al. 2021. Codeqa: A question answering dataset for source code comprehension. *Findings of EMNLP*.

Nelson F. Liu, Alex Tamkin, and et al. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2025. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval.

Shuai Lu et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*.

Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in language models for code intelligence: A taxonomy and survey. *arXiv preprint arXiv:2310.17903*.

Changyoon Sohn et al. 2022. Cs1qa: A dataset for assisting code-based question answering in an introductory programming course. *NAACL*.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. 2025. Cwm: An open-weights llm for research on code generation with world models.

Harsh Trivedi et al. 2022. Musique: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*.

Johannes Welbl et al. 2018. Constructing datasets for multi-hop reading comprehension across documents. *Transactions of the Association for Computational Linguistics*.

Tony Xiao, Ankit Patel, Jascha Sohl-Dickstein, and Zhenhai Chen. 2024. Scaling context length in language models: A practical investigation. *arXiv preprint arXiv:2402.10590*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

A. Comparison to Existing Benchmarks

Benchmark	Task	Scope	Multi-hop	Cross-file	Real Repos
CodeQA	QA	Snippet	✗	✗	✗
CS1QA	QA	Snippet	✗	✗	✗
CRUXEval	I/O Prediction	Function	✗	✗	✗
Code World Models	World Modeling	Function	✗	✗	✗
CodeXGLUE	Various	File	✗	✗	Partial
HumanEval	Generation	Function	✗	✗	✗
MBPP	Generation	Function	✗	✗	✗
SWE-bench	Patch Gen.	Repository	Implicit	✓	✓
LocAgent	Localization	Repository	Implicit	✓	✓
SWE-QA (Ours)	**MCQ**	**Repository**	✓	✓	✓

Table 5: Comparison of existing code benchmarks across key dimensions. SWE-QA is the only benchmark to jointly support explicit multi-hop reasoning, cross-file context, and real repository grounding.

B. Prompt Specifications

This section documents all prompts used in the `QuestionMaker` module. For each prompt, we explicitly distinguish between:

- **System Instructions** — Instructions sent as the system message
- **User Message** — Content sent as the user message

Placeholders enclosed in `{}` are programmatically substituted before sending. An explanation of the placeholders is available in Section B.10.

B.1. Entity-Specific Question Generation

Type: System + User message
Purpose: Generate a focused question about a specific entity

System Instructions

```
You will be given one or more code snippets, possibly from multiple files.
A specific entity (such as a class, function, or variable) will be identified.

Entity of Focus: {entity_name}

Task:
- Write one clear and concise question about this entity.
- The question should highlight something a developer might consider,
  such as purpose, behavior, interactions, or improvements.
- Keep the question short and direct.
- Do not explain the code or provide an answer.

Output format:
Question: <your question here>
```

User Message

```
<JOINED CODE CHUNKS CONTAINING ENTITY>
```

B.2. Interacting Entities Question Generation

Type: Single user prompt (no separate system message)
Purpose: Generate a question about the interaction between two entities

User Prompt Template

User Prompt

```
You are given two code entities, {entity_A} and {entity_B},
along with a snippet where they interact.

Your task is to write one clear and concise question about their relationship.

Input:
- {entity_A} Definition Code:
{entity_A_definition_code}

- {entity_B} Definition Code:
{entity_B_definition_code}

- Interaction Code:
{entity_interaction_code}

Guidelines:
- Ask about design, abstraction, dependencies, or side effects.
- Keep the question short and direct.
- Do not explain the code or provide answers.

Output:
Question: <your question here>
```

B.3. Question Extraction

Type: Single user prompt
Purpose: Extract only the final question from generated text

User Prompt Template

User Prompt

```
Extract only the question from the following text.
Return the question exactly, with no extra words or labels:

{generated_text}
```

B.4. Answer Generation for Code Comprehension

Type: System + User message
Purpose: Generate a detailed answer to a comprehension question

System Instructions

```
You are an expert in evaluating code comprehension. The user will provide
code and a question about it.

Your goal is to generate one relevant answer in English.

The answer should focus on:
- Essential mechanisms of how the code works
- Important design decisions
- Potential pitfalls or unexpected behaviors
```

```
Provide a clear and thorough answer demonstrating deep understanding.
```

User Message

```
<JOINED CODE CHUNKS>

<Question about the code>
```

B.5. MCQ Answer Sanitization

Type: Single user prompt
Purpose: Convert a verbose answer into a concise MCQ-style answer

User Prompt

```
You are an expert Python developer and technical writer.

I will give you:
1. A Python code snippet
2. A question about that code
3. A detailed answer

Your task is to sanitize the answer:
- Remove fluff and redundancy
- Keep only what directly answers the question
- Make it short, clear, and direct
- Do not repeat the question
- Do not rephrase the code

Input Code:
{code}

Question:
{question}

Original Answer:
{answer}

Sanitized Answer:
```

B.6. Distractor Generation for MCQs

Type: Single user prompt
Purpose: Generate exactly three plausible distractor answers for a programming MCQ

User Prompt

```
You are an expert MCQ generator specializing in programming assessments.

Given the following:

Code:
{code}

Question:
{question}

Correct Answer:
```

```
{answer}

Generate exactly 3 plausible distractor answers (incorrect but believable
    options)
to be used in a multiple-choice question. Each distractor should:

1. Be contextually relevant to the code and question
2. Represent a different level of Bloom's Taxonomy (e.g., Understanding,
    Applying, Analyzing)
3. Be plausible  choices a well-meaning but mistaken student might select
4. Be similar in structure or terminology to the correct answer
5. Avoid being trivially or obviously incorrect

Return ONLY the distractors as a valid Python list of dictionaries:

[
    {
        "option": "Distractor text here",
        "bloom_level": "Bloom's taxonomy level (e.g., Understanding,
    Applying, Analyzing)"
    },
    ...
]
```

B.7. Correct Answer Adaptation to Match Distractor Style

Type: Single user prompt
Purpose: Rephrase the correct answer to match the style and structure of generated distractors

User Prompt

```
You are an expert MCQ generator specializing in programming assessments.

Given the following:

Code:
{code}

Question:
{question}

Correct Answer:
{answer}

Here are 3 distractor answers generated for this question:
{distractor_examples}

Rephrase the correct answer so that it resembles the distractors in style,
    structure,
and terminology, but remains fully correct.

Return ONLY the adapted answer as a string, with no extra explanation or
    formatting.
```

B.8. Multiple-Choice Question Prompt for Model Benchmarking

Type: Single user prompt
Purpose: Ask a model to select the correct answer letter (A–D) for a code-related MCQ

User Prompt

```
You are given a piece of code, a related question, and four multiple-choice
    options
labeled A through D. Analyze the code, read the question carefully, and
    choose the
correct answer by responding with only the letter (A, B, C, or D).

Code:
{code}

Question:
{question}

Options:
{formatted_options}

Answer (respond with A, B, C, or D only):
```

Note: The `formatted_options` placeholder contains the code snippet, question text, and options dictionary formatted as shown in Section B.10.

B.9. Answer Extraction Using Helper Model

Type: System + User messages
Purpose: Extract only the correct answer letter from a model's response, optionally processing reasoning output

System Instructions

```
Extract the letter corresponding to the correct answer from the following
    response.
The output must be only the letter, with no extra explanation or characters.
```

User Message Sequence

User Message

```
Example conversation history:
1. User: "The answer to the question is A."
   Assistant: "A"
2. User: "B."
   Assistant: "B"
3. User: "C"
   Assistant: "C"

Followed by the actual response to extract:
{conclusion}
```

Note: The `conclusion` is extracted from the benchmark model output, with optional removal of `<think>...</think>` tags if reasoning extraction is enabled.

B.10. Placeholders Explained

`{entity_name}` The name of the specific entity being analyzed

`{entity_A}`, `{entity_B}` Names of two interacting entities

{entity_A_definition_code} Code defining entity A

{entity_B_definition_code} Code defining entity B

{entity_interaction_code} Code showing how the entities interact

{generated_text} Text from which to extract a question

{code} The source code snippet for the MCQ

{question} The text of the multiple-choice question

{answer} The correct answer to the question

{distractor_examples} The three generated distractor options

{formatted_options} Options A–D formatted as:

```
A. option_text
B. option_text
C. option_text
D. option_text
```

{conclusion} The benchmark model output after optional thought extraction

{llm_output} Raw model output potentially containing `<think>` tags

All placeholders are replaced with actual content before sending to the model.