SWE-QA-Pro: A Representative Benchmark and Scalable Training Recipe for Repository-Level Code Understanding

Anonymous ACL submission

Abstract

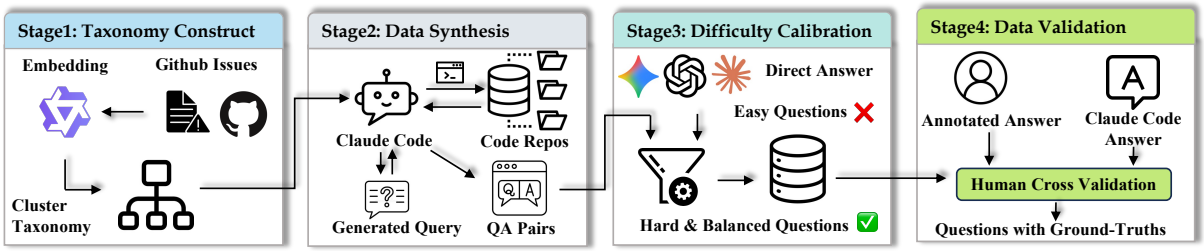
Agentic repository-level code understanding is essential for automating complex software engineering tasks, yet the field lacks reliable benchmarks. Existing evaluations often overlook the long tail topics and rely on popular repositories where Large Language Models (LLMs) can cheat via memorized knowledge. To address this, we introduce SWE-QA-Pro, a benchmark constructed from diverse, long-tail repositories with executable environments. We enforce topical balance via issue-driven clustering to cover under-represented task types and apply a rigorous difficulty calibration process: questions solvable by direct-answer baselines are filtered out. This results in a dataset where agentic workflows significantly outperform direct answering (e.g., a 13-point gap for Claude Sonnet 4.5), confirming the necessity of agentic codebase exploration. Furthermore, to tackle the scarcity of training data for such complex behaviors, we propose a scalable synthetic data pipeline that powers a two-stage training recipe: Supervised Fine-Tuning (SFT) followed by Reinforcement Learning from AI Feedback (RLAIF). This approach allows small open models to learn efficient tool usage and reasoning. Empirically, a Qwen3-8B model trained with our recipe surpasses GPT-4o by 2.3 points on SWE-QA-Pro and substantially narrows the gap to state-of-the-art proprietary models, demonstrating both the validity of our evaluation and the effectiveness of our agentic training workflow.

1 Introduction

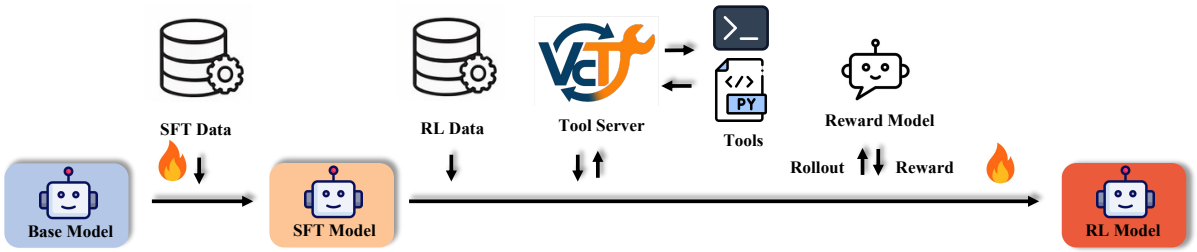
Repository-level code understanding is central to LLM-assisted software engineering. Real tasks require navigating many files, tracking control and data flow across modules, and verifying that implementations match intended designs. Snippet-centric QA benchmarks do not capture these behaviors, and knowledge-only prompting can hide weaknesses in navigation and grounding (Husain et al., 2019; Liu and Wan, 2021; Huang et al., 2021; Lee et al., 2022; Gong et al., 2024; Sahu et al., 2024; Li et al., 2024). Recent repository QA benchmarks move toward large-context, tool-using evaluation, but still focus on few projects and include many questions solvable without interacting with the codebase. (Abedu et al., 2025; Chen et al., 2025; Peng et al., 2025; Rando et al., 2025).

We focus on two concrete gaps. First, **limited diversity**: existing benchmarks concentrate on a few popular repositories. This leaves large parts of the natural task distribution uncovered and underrepresents certain semantic categories of tasks (e.g., configuration, data plumbing, infrastructure glue shown in Appendix A). Second, **uncertain need for tools**: many benchmark questions can be answered from prior knowledge or public documentation that is already covered during pretraining, so current setups do not clearly separate cases that actually requires tool-using from the cases where a single-pass, knowledge-only model with enough context and reasoning would already succeed. As a result, it is difficult to tell whether a model truly understands and operates within a particular repository, or simply recall generic knowledge.

To address these issues, we introduce SWE-QA-Pro, a benchmark and training recipe for repository-level QA. On the benchmark side, we: (i) select less-studied, long-tail repositories and ensure that each one has an executable environment so the project can be built and explored end-to-end (Badertdinov et al., 2025); (ii) use issue texts as question seeds, embed them, and run k-means clustering to form topic groups, followed by a brief human pass to merge near-duplicates and clarify topic boundaries; and (iii) for each topic, use a tool-using code model to propose QA items and draft answers, which are then edited by humans for correctness and repository grounding, with final benchmark QA items sampled across clusters to preserve diversity in Section 2.4.



(a) Benchmark Construction Pipeline



(b) Training Recipe

Figure 1: SWE-QA-Pro Benchmark and Training Pipeline.

To reduce knowledge-only questions and make tool usage meaningful, we add a simple filtering step. For each drafted item, we compare a direct-answer baseline (no tools, single turn) with a tool-using run. If the direct-answer baseline already achieves a high score, we discard the item. This preserves questions that require locating and citing concrete code rather than recalling documentation.

On the training side, we introduce a two-stage agentic recipe for improving small open models on repository-level QA. We first apply SFT to match repository-grounded answer formats, then use RLAIF to favor answers citing concrete files and symbols (Lee et al., 2023). In experiments, a tuned Qwen3-8B (Yang et al., 2025) trained with this SFT→RLAIF recipe outperforms GPT-4o (Hurst et al., 2024) and substantially narrows the gap to state-of-the-art proprietary models shown in Section 3.2 and Section 4.2.

In summary, we make two contributions, as illustrated in Figure 1:

- **Benchmark.** We release SWE-QA-Pro, a repository-level QA benchmark built from long-tail repositories with executable environments. Questions are seeded from issues, then synthesized and grounded with a tool-using code model along with human editing, followed by filtering to remove cases solvable by strong direct-answer baselines. Compared to SWE-QA, SWE-QA-Pro covers more diverse repositories and includes more questions that truly require codebase interaction (Peng et al., 2025).
- **Agent Workflow and Training Recipe.** We introduce a simple agentic workflow for repository-level QA that enables iterative codebase exploration via structured actions. Building on this workflow, we present an SFT→RLAIF training recipe that significantly improves small open-source models on SWE-QA-Pro. Using this framework, Qwen3-8B surpasses GPT-4o on SWE-QA-Pro by 2.31 points and substantially narrows the gap to several state-of-the-art proprietary models, including GPT-4.1, Claude Sonnet 4.5, and DeepSeek-V3.2 (OpenAI, 2025; Anthropic, 2025; Liu et al., 2025).

2 SWE-QA-Pro Bench

SWE-QA-Pro Bench is constructed through a four-stage pipeline, as illustrated in Figure 1a: Data Sourcing and Taxonomy, Data Synthesis and Sampling, Data Filtering and Difficulty Calibration, and Data Validation. This pipeline yields three key advantages over existing benchmarks (Table 1): (1) pull-request–driven clustering together with long-tail repository sampling ensures balanced coverage across diverse software engineering question types; (2) systematic filtering against multiple strong proprietary models removes instances solvable via memorization or pretraining artifacts, thereby isolating questions that require genuine codebase interaction; and (3) answers cross-verified by Claude Code and human annotators provide high-quality gold ground truth, enabling reliable multi-dimensional evaluation.

2.1 Data Sourcing and Taxonomy

We conducted a large-scale analysis of the GitHub Repositories in SWE-Rebench (Badertdinov et al.,

Benchmark	Repo-level	Repo Nav.	Multi-hop	Semantic Coverage	Diff. Calibration	Test Size
CodeQueries (Sahu et al., 2024)	✗	✗	✔	✗	✗	29033
InfiBench (Li et al., 2024)	✗	✗	✗	✗	✔	234
CodeReQA (Hu et al., 2024)	✔	✗	✗	✗	✗	1563
LongCodeQA (Rando et al., 2025)	✔	✗	✔	✗	✗	443
SWE-QA (Peng et al., 2025)	✔	✔	✔	✗	✗	576
SWE-QA-Pro	✔	✔	✔	✔	✔	260

Table 1: Comparison of representative code benchmarks.

2025). We processed $1,687,638$ issues spanning $3,468$ repositories by concatenating their titles and bodies, specifically filtering for contexts between 10 Byte and 16KB. We computed representations for these texts using Qwen3-8B-Embedding model.

To organize this data, we applied a hierarchical K-Means clustering algorithm, initializing with 10 clusters in the first layer and expanding to 50 in the second. We then utilized GPT-4.1 to extract semantic labels for each resulting cluster. These labels were refined through a human-verified taxonomy to eliminate semantic redundancy and enforce clear semantic boundaries between closely related categories, thereby reducing ambiguity and yielding 48 distinct task subclasses in Appendix E. This unsupervised taxonomy serves as the foundational structure for our benchmark, ensuring it covers a wide spectrum of software engineering challenges rather than a manually cherry-picked subset.

2.2 Data Synthesis and Sampling

Leveraging the derived semantic taxonomy, we employed Claude Code to synthesize the final benchmark data. To guarantee the executability and validity of the problems, we repurposed the established sandbox environments from SWE-Rebench.

For each synthesis task, we stochastically sampled 20 existing issues from the corresponding cluster and repository to serve as reference context. The agent was then tasked with automatically exploring the codebase to generate a new, self-contained problem-solution pair aligned with the specific cluster's semantics.

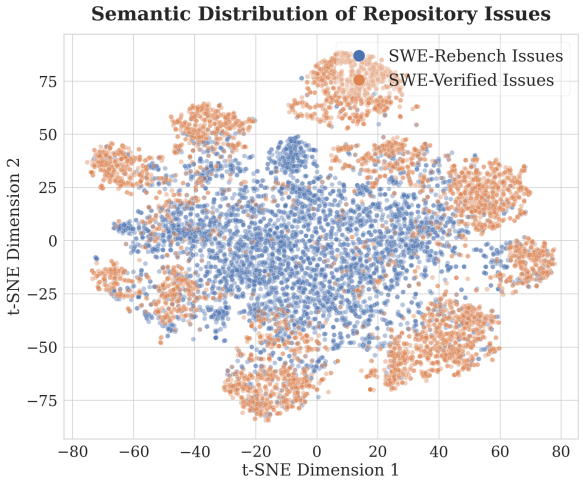
We adopted different sampling strategies for the training and test sets to balance diversity with human evaluation constraints. For the Test Set, we selected 26 repositories that efficiently cover all 48 task categories, accommodating the cognitive constraints of human annotators while ensuring comprehensive evaluation. Conversely, for the Training Set, we applied uniform sampling across the entire dataset, achieving coverage of $1,484$ repositories.

Figure 2 illustrates the effectiveness of this pipeline: our synthesized data maintains a semantic distribution that is sufficiently diverse compared to the original repository distribution between SWE-Rebench and SWE-Verified datasets.

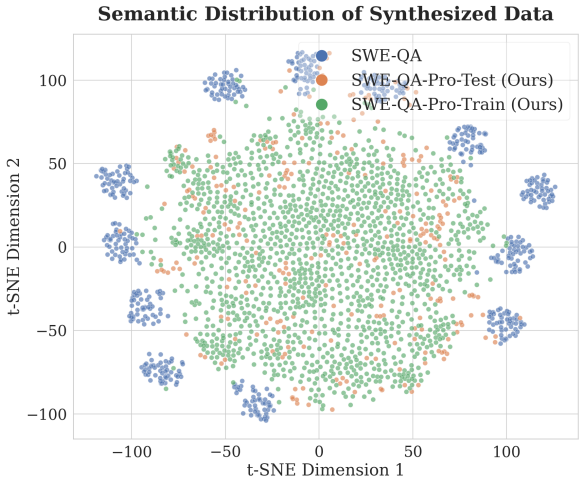
2.3 Data Filtering and Difficulty Calibration

To ensure SWE-QA-Pro focuses on non-trivial, agent-essential reasoning, we apply a multi-stage filtering and calibration pipeline. We first remove multi-query prompts and perform semantic deduplication using Qwen3-8B embeddings to ensure task independence. A key challenge in evaluating repository-level understanding is that state-of-the-art proprietary LLMs possess extensive pre-training knowledge, enabling them to answer many software engineering questions without interacting with the codebase, reading source files, or exploring repository structure. Such questions are often associated with widely known repositories (e.g., the canonical projects in SWE-Bench (Jimenez et al., 2023)), or can be resolved by inspecting only one or a small number of files, without requiring multi-hop reasoning over the repository. Empirically, this issue is reflected in existing repo-level QA benchmarks, where the performance gap between models answering with full repository exploration and those responding without any code context is often marginal. As a result, these benchmarks may fail to accurately measure an LLM's ability to explore codebases and perform grounded, repository-level reasoning (Peng et al., 2025).

To mitigate the influence of memorized knowledge and filter out trivially answerable questions, we introduce a difficulty calibration procedure based on cross-model agreement. We evaluate direct (no-repository) answers produced by three strong proprietary models, GPT-4o, Claude Sonnet 4.5, and Gemini 2.5 Pro (Comanici et al., 2025), and compare them against repository-grounded reference answers generated by Claude Code. Each direct answer is assessed using an LLM-as-a-Judge



(a) Semantic distribution of raw Issues: SWE-Rebench (All Repos) vs. SWE-Verified.



(b) Semantic distribution of QA Datasets: Comparison among our Training/Test splits and SWE-QA.

Figure 2: t-SNE visualization of semantic distributions. **(a)** Comparison of the original issue spaces, showing the broad coverage of SWE-Rebench compared to the manually curated SWE-Verified. **(b)** The distribution of our synthesized datasets (Training and Test) demonstrates high diversity and alignment with the semantic clusters of existing benchmarks.

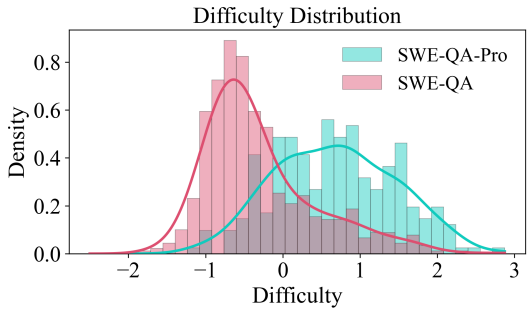


Figure 3: Difficulty Comparison between SWE-QA and SWE-QA-Pro. Higher difficulty indicates harder questions.

framework along five dimensions: correctness, completeness, relevance, clarity, and reasoning quality, as detailed in Section 4.1. For each model m, we aggregate scores across multiple independent runs and compute the average total score $\bar{s}_m(q)$ for question q.

To account for inter-model scale differences, we standardize the aggregated scores using z-score normalization:

$$z_m(q) = \frac{\bar{s}_m(q) - \mu_m}{\sigma_m}, \qquad (1)$$

where μ_m and σ_m denote the mean and standard deviation of model m's scores over all questions. We then define the difficulty of a question as the negative consensus score across models:

$$\text{Difficulty}(q) = -\frac{1}{|M|} \sum_{m \in M} z_m(q), \qquad (2)$$

where M denotes the set of evaluated models.

Under this definition, questions that consistently receive high-quality direct answers across models are assigned low difficulty and are filtered out, while questions that remain challenging without repository interaction are retained. Using the calibrated difficulty signal, we construct the QA pairs candidates pools with cluster-level coverage and approximate balance across QA types. This calibration step ensures that SWE-QA-Pro emphasizes questions that genuinely require repository exploration and multi-step reasoning, providing a more faithful evaluation of LLM agent capabilities as shown in Figure 3.

2.4 Data Validation and Statistics

To ensure the quality and reliability of QA pairs and metadata, we adopt a multi-stage annotation and validation process to mitigate hallucinations and semantic ambiguity. First, CLAUDE CODE explores each repository in a sandbox environment to produce repository-grounded reference answers, while assigned semantic clusters and QA types are cross-checked against the taxonomy and reused for difficulty calibration.

Second, human annotators independently explore the codebase to produce answers, revise ambiguous or underspecified questions, and verify QA types and semantic clusters. Their answers are compared against the CLAUDE CODE references to identify missing details or errors, and only

answers satisfying correctness, completeness, relevance, clarity, and reasoning quality are retained. A final expert review pass adjudicates remaining inconsistencies and further refines the answers.

The resulting SWE-QA-Pro benchmark contains 260 questions from 26 long-tail repositories, with 4–9 questions per semantic cluster and an approximately balanced distribution of QA types. Full statistics are reported in Appendix F, with case studies in Appendix H.1.

3 SWE-QA-Pro Agent

We introduce SWE-QA-Pro Agent, a lightweight workflow designed for repository-level code understanding in small open-source models. Unlike RAG-based approaches that require pre-built indices, our agent uses a ReAct-style loop to explore codebases directly. By combining directory traversal, keyword search, and scoped file inspection, the agent gathers evidence incrementally to reason across files under limited context budgets.

3.1 Agent Workflow

We propose SWE-QA-Pro Agent, a ReAct-based workflow for repository-level code understanding. Prior agents such as SWE-QA-Agent primarily rely on RAG-style retrieval with limited command-line support, requiring the construction of a retrieval index while still offering insufficient capacity for genuine repository exploration. This limitation is particularly evident for open-source models, where such agents often underperform strong traditional RAG baselines with offline indexing and manually designed retrieval pipelines.

In contrast, SWE-QA-Pro Agent abandons RAG-based retrieval entirely and does not require a pre-built index. Instead, it performs direct repository exploration using explicit, length-controlled Search based on keyword matching to locate relevant files, View for scoped inspection of file contents or directory structure, and constrained read-only CommandLine actions for lightweight structural and pattern-based analysis (e.g., directory traversal, symbol matching, and line-level extraction), enabling more flexible and effective context acquisition for reasoning under limited context budgets.

The agent operates in a ReAct-style loop, where it iteratively reasons over the current context, freely selects an action, and incorporates the resulting observation until sufficient evidence is collected, at which point it terminates with Finish. Detailed algorithm are provided in the Appendix B.

3.2 Agentic Training Recipe

To our knowledge, existing efforts to enhance open-source LLMs for SWE-QA focus on SFT of agentic behaviors, without leveraging reinforcement learning to optimize repository-level exploration and reasoning (Rastogi et al., 2025). Inspired by recent advances in RL for LLMs, we propose a scalable training framework that explicitly trains agentic interaction with code repositories, leading to improved exploration and understanding.

Training Data Construction. Starting from the benchmark question construction pipeline, we deduplicate and obtain 1,464 raw questions, which are randomly split into 1,000 questions for SFT and 464 questions for RL. For the SFT stage, we use `Claude Sonnet 4.5` to generate 1,000 high-quality multi-turn conversation trajectories conditioned on each question and our predefined agent action space (Search, View, and read-only CommandLine tools), resulting in tool-augmented supervision data. For the RL stage, we assign each question a high-quality reference answer generated by `Claude Code`, which serves as the ground truth for reward computation.

Two-Stage Training. Training proceeds in two stages. In the first stage, we perform supervised fine-tuning on `Qwen3-8B` using 1K tool-invocation question–answer trajectories. This stage teaches the model the tool-call syntax and instills a basic understanding of tool semantics and usage patterns. In the second stage, we apply reinforcement learning to the SFT-initialized model. For each rollout, a reward model evaluates the final answer against the ground truth along five dimensions: correctness, completeness, relevance, clarity, and reasoning quality, following the same criteria used in evaluation. Since SWE-QA answers are often complex and cannot be reliably assessed by exact-match or rule-based rewards, we adopt an LLM-as-Judge reward formulation. To mitigate reward hacking, we employ a judge model distinct from the evaluation judge and assign higher weight to correctness while down-weighting clarity, discouraging fluent but incorrect answers. The final scalar reward is computed as:

$$\mathbf{s} = \mathrm{RM}(\hat{a}, a^*) \in [1, 10]^5, \tag{3}$$

$$r = \frac{\mathbf{w}^\top \mathbf{s}}{10}, \quad \mathbf{w} = (0.3, 0.2, 0.2, 0.1, 0.2). \tag{4}$$

Model	Evaluation Metrics					Overall
	Correctness	Completeness	Relevance	Clarity	Reasoning	
Proprietary LLMs						
Gemini 2.5 Pro	2.51	2.13	8.66	8.02	4.16	25.48
Gemini 2.5 Pro + Agent	7.12	6.25	8.91	**9.34**	7.84	39.46
GPT-4.1	3.42	2.38	**9.02**	9.23	4.68	28.74
GPT-4.1 + Agent	6.86	5.90	8.89	9.13	7.68	38.47
GPT-4o	3.08	2.11	8.96	8.79	3.64	26.58
GPT-4o + Agent	5.59	4.49	8.55	8.16	6.29	33.08
DeepSeek V3.2	3.19	2.32	8.83	8.83	4.39	27.55
DeepSeek V3.2 + Agent	6.94	6.49	8.78	8.72	7.76	38.69
Claude Sonnet 4.5	3.34	2.74	8.65	8.12	4.84	27.69
Claude Sonnet 4.5 + Agent	**7.34**	**7.36**	8.88	9.03	**8.06**	**40.67**
Open-Source LLMs						
Qwen3-8B	2.84	2.16	8.59	8.66	4.36	26.61
Qwen3-8B + Agent	4.52	3.77	8.29	7.83	5.62	30.03
Qwen3-32B	3.04	2.41	8.71	8.74	5.02	27.91
Qwen3-32B + Agent	4.99	4.21	8.50	8.16	6.22	32.08
Llama-3.3-70B-Instruct	2.34	1.75	8.68	8.47	3.08	24.32
Llama-3.3-70B-Instruct + Agent	2.84	2.11	8.09	7.18	3.51	23.73
Devstral-Small-2-24B-Instruct	2.65	2.14	8.57	8.31	3.77	25.44
Devstral-Small-2-24B-Instruct + Agent	**6.61**	**5.66**	**8.81**	**9.09**	**7.13**	**37.30**
Finetuned LLMs						
SWE-QA-Pro-8B (SFT)	2.56	2.01	8.37	7.93	3.55	24.42
SWE-QA-Pro-8B (SFT) + Agent	5.66	5.45	8.40	8.21	6.61	34.34
SWE-QA-Pro-8B (SFT+RL)	2.54	2.04	8.28	7.92	3.55	24.34
SWE-QA-Pro-8B (SFT+RL) + Agent	**5.96**	**5.66**	**8.51**	**8.44**	**6.83**	**35.39**

Table 2: SWE-QA-Pro Bench evaluation results. "+agent" denotes models using the SWE-QA-Pro agent framework. Best results per scale are shown in **bold**, with second-best underlined.

where $\hat{a}$ is the generated answer, a^* the ground-truth reference, and $\mathbf{s}$ denotes scores for five evaluation dimensions. We optimize the policy using the GRPO algorithm (Shao et al., 2024), where rewards are normalized within each rollout group before computing policy gradients. This stage encourages the model to converge toward rollouts that produce high-quality, fact-grounded final answers.

4 Experiments

4.1 Experimental Setup

Model selection We evaluate 11 LLMs, including proprietary models, GPT-4o, GPT-4.1, Claude Sonnet 4.5, Gemini 2.5 Pro, DeepSeek-V3.2, open-source models, Qwen3-8B/32B, Devstral-Small-2-24B-Instruct (Rastogi et al., 2025), LLaMA-3.3-70B-Instruct (Dubey et al., 2024), and two variants of SWE-QA-Pro 8B trained with SFT and SFT+RL. All models are evaluated under both direct answering and agent-based reasoning using the SWE-QA-Pro Agent workflow.

Inference and Training Setup All inference uses temperature 0, a maximum of 25 turns, and a 32k context window on NVIDIA A100 80GB GPUs. SFT and RL are implemented using SWIFT (Zhao et al., 2025) and Verl-Tool (Jiang et al., 2025), respectively. Hyperparameters are provided in Appendix C.

Evaluation Metrics We follow the LLM-as-Judge protocol of SWE-QA, including strict judge–candidate separation, anonymization, and randomized answer order. Compared to SWE-QA, we require explicit file-path and line-number references and use a stricter judge prompt to enable finer-grained score differentiation. Each answer is scored independently three times, and scores are averaged to reduce variance. Full prompts are provided in Appendix D.

4.2 Main Results

Table 2 summarizes the evaluation results across all LLMs. As shown in Table 2, there is a substantial performance gap between the direct-answer setting and the agent-based workflow, particularly on correctness, completeness, and reasoning quality. This gap highlights the effectiveness of our difficulty calibration and underscores the critical role of the SWE-QA-Pro agent in enabling high-quality, repository-grounded reasoning.

Overall Performance. Among all evaluated mod-

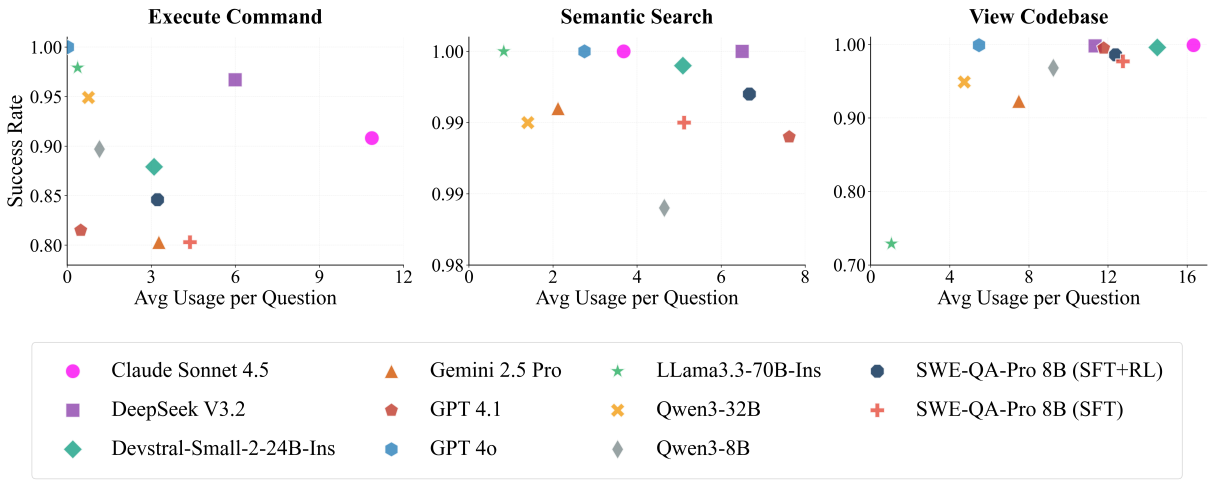


Figure 4: Tool usage behavior across models on SWE-QA-Pro.

els, Claude Sonnet 4.5 achieves the highest overall score, reflecting strong repository-level code understanding and effective tool use. Despite its smaller scale and limited training data, SWE-QA-Pro 8B outperforms many open-source baselines as well as GPT-4o with case stduy in Appendix H.2, and performs competitively with larger agentic models such as Devstral Small 2 24B Instruct. These results highlight that explicitly training agentic capabilities for repository-level QA can be more impactful than scaling model size alone.

Breakdown Results Analysis. Appendix G details model performance across repositories, semantic clusters, and question types. First, analyzing question types (Table 11, Table 12) reveals that localization-oriented questions yield consistently high scores with low variance, as they focus on identifying specific files, identifiers, or execution points. Conversely, causal and explanatory questions are significantly more challenging, particularly those involving design rationale, trade-offs, or implicit dependencies, which require multi-file evidence integration and global semantic reasoning. Procedural questions fall in between: concrete implementation tasks are tractable, whereas system-level inquiries remain difficult due to their reliance on holistic understanding.

Regarding repositories and semantic clusters in Appendix G.2 and G.2, configuration and workflow management areas involving dependency injection, CLI argument handling, and packaging prove consistently difficult. These clusters characterize configuration-driven repositories like jsonargparse, checkov, and yt-dlp, where answering requires reasoning over implicit control flow, cross-file propagation, and runtime behavior not localized to single files. In contrast, clusters with explicitly encoded, localized logic, such as Unicode/data parsing, protocol/API compatibility, filesystem config, and visualization, are easier. Consequently, repositories concentrating on these areas (docker-py, pint, mkdocs, seaborn) achieve higher, stable performance, benefiting from structurally explicit and locally grounded code.

4.3 Tool Usage Analysis

Figure 4 correlates tool proficiency with repository-level QA performance. Models with lower scores, such as LLaMA-3.3-70B-Instruct and GPT-4o, suffer from weak tool usage that limits context retrieval and global understanding. Conversely, Claude Sonnet 4.5 excels by leveraging the highest volume of tool calls, translating robust exploration into superior answer quality. Gemini 2.5 Pro, however, remains competitive with fewer calls, indicating that internal reasoning enables efficient, selective tool use; this underscores that reasoning is vital alongside tool capacity. Additionally, post-RL improvements in SWE-QA-Pro 8B demonstrate that RL fosters effective, judicious execution rather than merely inflating tool-call frequency.

4.4 Training Strategy Analysis

Figure 5 compares different training strategies under the same model backbone. SFT-1000 and SFT-1464 denote supervised fine-tuning on 1,000 and 1,464 tool-call trajectories, respectively, while SFT-1000 + RL-464 represents a two-stage setting that initializes the model with 1,000 SFT trajectories and then applies reinforcement learning on an additional 464 QA pairs. Increasing the amount of supervised fine-tuning data from 1,000 to 1,464 trajectories yields consistent but modest improvements across most evaluation dimensions. In contrast, in-

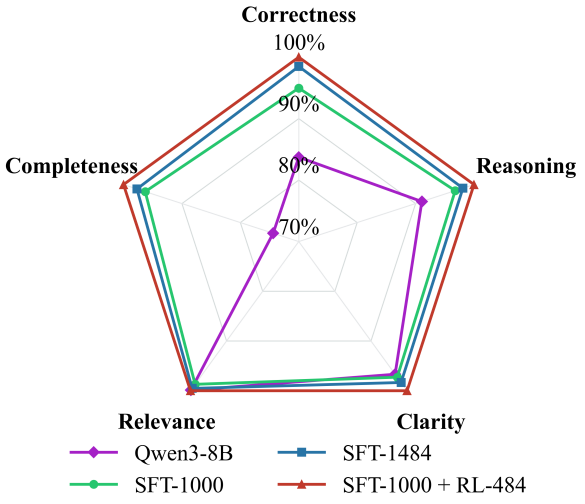


Figure 5: Effect of Training Strategy. We compare SFT at different scales and a two-stage SFT→RLAIF setting.

troducing reinforcement learning after SFT leads to a more pronounced performance gain. In particular, SFT-1000 + RL-464 achieves substantially higher scores in both Correctness and Completeness than SFT-only variants, including SFT-1464. This indicates that reinforcement learning does not simply replicate the effect of scaling supervised data, but instead introduces a qualitatively different optimization signal that further unlocks the potential of the SFT-initialized model, encouraging more accurate and more comprehensive answers. Overall, these results suggest that RL provides complementary supervision beyond SFT, especially effective at refining factual precision and answer coverage.

5 Related Work

Code-centric and Repository-level QA Benchmarks. Existing code and repository QA benchmarks focus on localized or context-limited settings, where questions can be answered from snippets, APIs, or documentation. Representative datasets such as CodeQueries, CS1QA, CoSQA, CoSQA+, and CodeSearchNet emphasize element-level reasoning or retrieval over individual functions, deliberately avoiding cross-file dependencies and repository structure (Sahu et al., 2024; Lee et al., 2022; Huang et al., 2021; Gong et al., 2024; Husain et al., 2019). InfiBench extends evaluation to free-form coding-related questions across languages, but remains knowledge- and snippet-centric rather than repository-grounded (Li et al., 2024).

More recent efforts move toward repository-scale evaluation. LongCodeBench relies on long context windows to ingest large codebases (Rando et al., 2025), while RepoChat uses offline indexing for structured retrieval (Abedu et al., 2025). SWE-QA formulates repository understanding as a QA task (Peng et al., 2025), but does not explicitly separate cases solvable by standard retrieval. In contrast, SWE-QA-Pro targets long-tail, executable repositories and filters out retrieval-solvable queries, isolating scenarios that require interactive code exploration.

Repository-level Agents. Agents in software engineering largely target generative tasks, including issue resolution, program repair, and code generation (Jimenez et al., 2023; Yang et al., 2024; Zhang et al., 2024; Da et al., 2025; Li et al., 2025; Bi et al., 2024; Bairi et al., 2024). In these domains, exploration is implicitly shaped by generation objectives rather than comprehension. Conversely, repository-level QA demands strict code navigation and understanding. Prior approaches, such as SWE-QA-Agent, rely on inference-time heuristics for tool use, often underperform retrieval-augmented generation (RAG) baselines due to unoptimized navigation (Peng et al., 2025). We address this limitation by explicitly training repository exploration policies, bridging the gap between passive retrieval and active agentic navigation.

6 Conclusion

In this work, we address the challenge of evaluating and training Large Language Models for repository-level code understanding, where reliance on memorized knowledge often masks deficits in genuine exploration capabilities. By introducing SWE-QA-Pro, we establish a rigorous testbed that enforces semantic diversity through long-tail repositories and systematically filters out questions solvable by direct answering. The substantial performance gap observed between direct and agentic baselines on this benchmark confirms that our design successfully isolates tasks requiring authentic codebase navigation and evidence grounding.

Beyond evaluation, we show that agentic capabilities can be learned using a scalable framework. We propose a two-stage SFT→RLAIF training recipe enabled by a synthetic data pipeline. A Qwen3-8B model trained with this recipe surpasses GPT-4o on SWE-QA-Pro and substantially narrows the gap to state-of-the-art proprietary models. We hope SWE-QA-Pro catalyzes future research toward active, grounded repository reasoning.

Limitations

The objective of SWE-QA-Pro is to provide a challenging benchmark and practical training recipe, yet we identify three limitations. First, despite employing semantic embedding clustering to maximize topical coverage, the benchmark is constrained to 260 questions from 26 repositories due to the high cost of expert human verification; this scale may not fully capture the extreme long-tail diversity of the software ecosystem, and the embedding models used for clustering could inadvertently introduce latent biases into the taxonomy. Second, the benchmark is currently restricted to the Python ecosystem due to the strict requirement for executable sandboxes to verify agent actions, though our data synthesis and agentic training pipeline is inherently language-agnostic and can be readily extended to other languages with compatible runtime environments. Third, our RLAIF training objectives share a similar distribution with evaluation metrics as both rely on LLM-as-a-Judge frameworks, creating a potential risk of reward hacking where models optimize for judge preferences rather than objective correctness. While preliminary case studies did not reveal significant gaming behaviors, this proximity suggests that future research should prioritize investigating more robust training methodologies—such as process supervision or diverse reward modeling—to mitigate such alignment risks.

References

Samuel Abedu, Laurine Menneron, SayedHassan Khatoonabadi, and Emad Shihab. 2025. Repochat: An llm-powered chatbot for github repository question-answering. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 255–259. IEEE.

Anthropic. 2025. Claude sonnet 4.5 system card. https://www.anthropic.com/news/claude-sonnet-4-5. Accessed: 2025-10-27.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. *arXiv preprint arXiv:2403.16792*.

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*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. 2025. Agent-rlvr: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jing Gong, Yanghui Wu, Linxi Liang, Yanlin Wang, Jiachi Chen, Mingwei Liu, and Zibin Zheng. 2024. Cosqa+: Pioneering the multi-choice code search benchmark with test-driven agents. *arXiv preprint arXiv:2406.11589*.

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. *arXiv preprint arXiv:2105.13239*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, and 1 others. 2025. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. Cs1qa: A dataset for assisting code-based question answering in an introductory programming course. *arXiv preprint arXiv:2210.14494*.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv e-prints*.

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

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

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Chenxiao Liu and Xiaojun Wan. 2021. Codeqa: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*.

OpenAI. 2025. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/. Accessed: 2026-01-06.

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

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

Abhinav Rastogi, Adam Yang, Albert Q Jiang, Alexander H Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, and 1 others. 2025. Devstral: Fine-tuning language models for coding agent applications. *arXiv preprint arXiv:2509.25193*.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, and 1 others. 2025. Swift: a scalable lightweight infrastructure for fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 29733–29735.

Table of Contents in Appendix

A Cluster Coverage of SWE-QA Bench

Cluster ID	Cluster Name	Original Ratio (%)	Test Ratio (%)	Coverage (×)	Origin Count	Test Count
0.0	Unicode / formatting / parsing / data validation	2.04	0.00	0.00	17,185	**0**
0.1	SQL / structured grammar / templating Engine	2.67	1.89	0.71	22,489	10
0.2	CLI args / syntax / regex / command completion	2.08	3.22	1.55	17,518	17
0.3	file import/export / metadata / sorting / recurrence	1.89	0.19	0.10	15,940	**1**
1.0	data formats / FITS / Astropy / units / WCS	0.93	8.14	8.78	7,825	43
1.1	pandas / parquet / datetime / Dask / table schema	1.77	0.57	0.32	14,917	**3**
1.2	numpy / dask arrays / dtype / array serialization / parallel	1.96	8.33	4.24	16,571	44
1.3	coordinate transform / image processing / IO / numeric precision	3.99	0.00	0.00	33,660	**0**
2.0	protocol serialization / encoding / headers / API compatibility	1.25	0.00	0.00	10,573	**0**
2.1	dependency injection / config / attrs / API design	2.27	7.20	3.17	19,168	38
2.2	dataclass / schema validation / enums / OpenAPI	1.68	0.76	0.45	14,190	4
2.3	type and attribute errors / initialization / CLI workflow	2.00	0.19	0.09	16,837	**1**
2.4	type hints / mypy / typing system / code generation	1.71	1.52	0.89	14,436	8
3.0	Python version / imports / deprecation / conflicts	1.93	0.00	0.00	16,300	**0**
3.1	install / virtual env / OS / hardware requirements / cloud deploy	2.14	0.00	0.00	18,035	**0**
3.2	artifacts / distribution format / repository management / post-install state	2.09	0.57	0.27	17,677	3
3.3	extensibility / configuration framework / plugin architecture	2.96	0.00	0.00	24,998	**0**
4.0	version control / Docker / build cache	1.81	0.00	0.00	15,274	**0**
4.1	release management / changelog / license / community	1.98	0.00	0.00	16,692	**0**
4.2	documentation / MkDocs / user tutorials	3.11	5.30	1.70	26,274	28
4.3	async refactor / migration / logging infra / i18n	2.37	0.38	0.16	19,972	**2**
4.4	CI pipelines / coverage / lint / GitHub Actions / security checks	1.30	0.19	0.15	11,011	**1**

Table 3: Cluster Coverage (1.0 - 4.4): SWE-Rebench (Original) vs. SWE-QA (Test).
[Back to Appendix Contents]

Cluster ID	Cluster Name	Original Ratio (%)	Test Ratio (%)	Coverage (×)	Origin Count	Test Count
5.0	asyncio / async context / resource cleanup	2.11	0.19	0.09	17,811	**1**
5.1	multiprocessing / advanced runtime / concurrency / heterogeneous compute	1.07	0.00	0.00	9,034	**0**
5.2	runtime error handling / DB transactions / retry / logging system	2.48	0.76	0.31	20,891	4
5.3	threading / execution limits / scheduling / memory / timeout	2.17	0.76	0.35	18,278	4
5.4	connection lifecycle / protocol handling / low-level failures	2.08	0.19	0.09	17,569	**1**
5.5	parallel execution / distributed frameworks / task graphs	1.72	0.00	0.00	14,511	**0**
6.0	file paths / filesystem permissions / symlinks / env config / cache system	2.13	0.76	0.36	17,982	4
6.1	unit testing / mocking / test automation	2.05	8.90	4.34	17,323	47
6.2	build pipeline / doc building / Sphinx / cloud provisioning	3.30	0.00	0.00	27,858	**0**
6.3	compiler toolchain / cross-compile / env vars / code quality analysis	1.42	0.00	0.00	11,984	**0**
7.0	API integration / sync / performance / DB / SDK	2.62	0.95	0.36	22,067	5
7.1	media download / playlist / metadata / client-side proxy config	1.13	0.00	0.00	9,532	**0**
7.2	auth systems / deployment / extension plugins / cloud services	2.21	2.46	1.11	18,668	13
7.3	AWS / Azure / K8s / container security / IAM policy	2.20	0.00	0.00	18,531	**0**
7.4	reverse proxy / URL routing / websocket / CDN / streaming	1.42	15.34	10.79	12,000	81
7.5	OAuth / JWT / SSL / access control / user sessions/ token lifecycle	4.00	1.33	0.33	33,762	7
8.0	tensors / training / GPU / ML experiment logging / tuning	2.20	2.08	0.95	18,525	11
8.1	ML analytical visualization / Fourier / ML animation / calibration	2.76	1.33	0.48	23,278	7
8.2	time series / feature engineering / explainability methods / behavioral analysis / computational semantics	0.90	2.46	2.73	7,606	13
8.3	data parallel / compression / ML plugin / indexing	2.39	2.65	1.11	20,182	14
8.4	bayesian models / MCMC / statistics / reproducibility	1.67	1.14	0.68	14,066	6
8.5	ML APIs / decorators / metrics / optimization strategies	2.15	10.61	4.94	18,116	56
9.0	UI layout / CSS / markdown / table extraction / frontend security	2.30	0.38	0.16	19,450	2
9.1	plotting systems / widgets / maps / UI animation / usability	1.78	4.92	2.77	14,995	26
9.2	runtime UI config / UI permission management / upload handling / customization / user-facing runtime extensibility	2.09	0.19	0.09	17,643	**1**
9.3	3D rendering / legends / color mapping / visualization formatting	1.73	4.17	2.41	14,615	22

Table 4: Cluster Coverage (5.0 - 9.3): SWE-Rebench (Original) vs. SWE-QA (Test).
[Back to Appendix Contents]

B SWE-QA-Pro Agent Algorithm

Algorithm 1 SWE-QA-Pro Agent

Require: User query Q, repository R
Ensure: Final answer A

/* Phase 1: Initialization */
$context \leftarrow [\,]$
$thought \leftarrow Analyze(Q)$
$context \leftarrow context \cup \{SemanticSearch(Q, R)\}$

/* Phase 2: Iterative ReAct Loop */
$max\_iterations \leftarrow N$
for $i \leftarrow 1$ **to** $max\_iterations$ **do**
    $thought \leftarrow Reason(context, Q)$
    $action \leftarrow SelectAction(thought)$
    if $action = SemanticSearch$ **then**
        $output \leftarrow Execute(SemanticSearch)$
    else if $action = ViewCodebase$ **then**
        $output \leftarrow Execute(ViewCodebase)$
    else if $action = ExecuteCommand$ **then**
        $output \leftarrow Execute(ExecuteCommand)$
    end if
    $context \leftarrow context \cup \{output\}$
    if $SufficientEvidence(context, Q)$ **or** $i = max\_iterations$ **then**
        break
    end if
end for

/* Phase 3: Finalization */
$A \leftarrow Synthesize(context, Q)$
return A

C Training Hyperparameters

Hyperparameter	Value
Precision	bfloat16
Max sequence length	32,768
Optimizer	AdamW
Learning rate	5×10^{-6}
Weight decay	0.05
LR scheduler	Cosine
Warmup ratio	0.05
Batch size (per device)	1
Gradient accumulation	2
Epochs	4
Agent template	Hermes

Table 5: Hyperparameters for SFT of SWE-QA-Pro 8B. [Back to Appendix Contents]

Hyperparameter	Value
Max turns	25
Max prompt length	2,048
Max response length	8,192
Max observation length	28,000
Temperature	1.0
Top-p	1.0
Number of rollouts (n)	8
KL loss coefficient	0.02
KL loss type	Low-variance KL
Entropy coefficient	0
Actor learning rate	1×10^{-6}
Batch size	8
PPO mini-batch size	8
Strategy	FSDP
Max model length	32,768

Table 6: Hyperparameters for RL of SWE-QA-Pro 8B. [Back to Appendix Contents]

D Prompts

Prompt Template for LLM-as-Judge

Model: GPT-5

You are a professional evaluator. Please rate the candidate answer against the reference answer based on five criteria. Evaluation Criteria and Scoring Guidelines (each scored 1 to 10):

1. Correctness:
10 — Completely correct; core points and details are accurate with no ambiguity.
8-9 — Mostly correct; only minor details are slightly inaccurate or loosely expressed.
6-7 — Partially correct; some errors or omissions, but main points are generally accurate.
4-5 — Several errors or ambiguities that affect understanding of the core information.
2-3 — Many errors; misleading or fails to convey key information.
1 — Serious errors; completely wrong or misleading.

2. Completeness:
10 — Covers all key points from the reference answer without omission.
8-9 — Covers most key points; only minor non-critical information missing.
6-7 — Missing several key points; content is somewhat incomplete.
4-5 — Important information largely missing; content is one-sided.
2-3 — Covers very little relevant information; seriously incomplete.
1 — Covers almost no relevant information; completely incomplete.

3. Relevance:
10 — Content fully focused on the question topic; no irrelevant information.
8-9 — Mostly focused; only minor irrelevant or peripheral information.
6-7 — Generally on topic; some off-topic content but still relevant overall.
4-5 — Topic not sufficiently focused; contains considerable off-topic content.
2-3 — Content deviates from topic; includes excessive irrelevant information.
1 — Majority of content irrelevant to the question.

4. Clarity:
10 — Fluent language; clear and precise expression; very easy to understand.
8-9 — Mostly fluent; clear expression with minor unclear points.
6-7 — Generally clear; some expressions slightly unclear or not concise.
4-5 — Expression somewhat awkward; some ambiguity or lack of fluency.
2-3 — Language obscure; sentences are not smooth; hinders understanding.
1 — Expression confusing; very difficult to understand.

5. Reasoning:
10 — Reasoning is clear, logical, and well-structured; argumentation is excellent.
8-9 — Reasoning is clear and logical; well-structured with solid argumentation.
6-7 — Reasoning generally reasonable; mostly clear logic; minor jumps.
4-5 — Reasoning is average; some logical jumps or organization issues.
2-3 — Reasoning unclear; lacks logical order; difficult to follow.
1 — 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 [1,10], corresponding to the evaluation scores:

```
{{
"correctness": <1-10>,
"completeness": <1-10>,
"relevance": <1-10>,
"clarity": <1-10>,
"reasoning": <1-10>
}}
```

REQUIREMENT:
You should assume that a score of 5 represents an average but imperfect answer. Scores above 7 should be reserved for answers that are clearly strong. Do not infer or assume missing information. Score strictly based on what is explicitly stated. No explanation, no extra text, no formatting other than valid JSON

Prompt Template for Generating Answer

Model: All Evaluated Model

System Prompt:

You are a codebase analysis agent operating in a strictly read-only environment.
Your task is to answer SWE-related questions by analyzing source code, configuration, documentation, and tests. You must prioritize correctness, completeness, clarity, relevance and evidence-based reasoning when answering given questions within 25 max turns.

`PROCESS PROTOCOL (MANDATORY)`
For every question, you MUST follow this process:
1. Planning
Before calling any tools, you MUST output a short planning explanation at each turn.
* Explain step by step what you have found so far from the current context, and what you will inspect next and why.
* This reasoning MUST be explicit and visible.
2. Investigation
* Call one or more read-only tools to gather evidence.
* Multiple tool calls in one turn are allowed.
3. Synthesis
* Combine evidence acoss multiple files or components.
* Do NOT rely on a single file unless clearly justified.
4. Finalization
* Produce a final answer following the OUTPUT PROTOCOL.

`TOOL USAGE RULES`
Available tools:
* semantic_search: find relevant files, symbols, or modules.
* viewcodebase: inspect structure or specific file sections.
* Prefer ‘concise=True‘ first; use ‘viewrange‘ when needed. Prefer using viewcodebase; avoid using ls -l or ls -R whenever possible. Don’t use tree without -L.
* executereadonlycommand: small, focused inspection tasks that require raw command output (Avoid using command-line operations that produce excessive and uncontrollable output, DON’t use ls -R path and ls -lR path as a command).
You may call one or more functions to assist with the user query.
You are provided with function signatures within <tools></tools> XML tags:
<tools> `{tools}` </tools>

`OUTPUT PROTOCOL (STRICT)`
You MUST follow this output structure at each assitant turn:
1. Reasoning
* Before any tool call, output only your step by step planning explanation
2. Final Answer
* Output **exactly one** block in this format without any tool calls:
<finish> `{Final answer's content}` </finish>
Rules for ‘<finish>‘ block:
* Must appear exactly once.
* Must contain only the final answer’s content.
* NO code blocks or copied code such as “‘python ...““.
* Cite evidence only using file paths relative to repo_path, in the format <relative_path>: line <start>-<end> (do not use absolute paths) e.g. responses/init.py: line 1-10.
Any violation of this protocol makes the answer invalid.
The working directory (where the code is executed) is /data/songcheng/SWE-QA-Pro-dev/eval. Now the code repo at repopath. Please use absolute paths in all tools.

User Prompt:

Repository Path: repopath
Question:question
Instructions:
- Please analyze the codebase to answer this question.
- Provide a step-by-step explanation before calling any tools.
- Follow this workflow:
1) Inspect the repository structure
2) Search for relevant files and symbols
3) Examine specific implementations
4) Cross-validate your findings
5) Provide a complete answer with evidence inside a <finish> block

Prompt Template for Generating Query Candidates

Model: Claude Code

You are a repository-aware planning agent. Your job is NOT to modify code, but to:
(1) lightly explore the local repository,
(2) generate ONE high-quality developer Query tailored to this repo,
(3) classify it with: cluster, task_type, clarity (0–5), context (0–5), difficulty (0–6),
(4) provide evidence (paths, line ranges for each paths, signals) and a concise rationale summary,
(5) produce a detailed NEXT_STEPS plan that explains how to solve the generated Query using your general software knowledge and the repo evidence. Each step must reference specific file paths and line ranges from the evidence.
Keep internal reasoning private. Output must follow the strict JSON schema at the end.

[SUGGESTED — OPTIONAL, USER-CUSTOMIZABLE]
- Goal bias: prefer qa_verifiable tasks that ask for structural, architectural, or algorithmic explanations with standard answers
- Risk preference: avoid trivial or opinion-based questions; push toward mid/high-level system questions (e.g., how modules interact, why design chosen)
- Domain preference: explanations tied to repo internals, standardized APIs, algorithmic design, or widely accepted conventions
- Complexity target: cover difficulty 2–5 (multi-step reasoning about design/structure, not trivial lookups)
- Output style: queries must include a ground_truth_answer field that captures a definitive, evidence-based explanation
[/SUGGESTED]

BUCKET / CLUSTER INDEX (L1 → L2)

Choose the tightest cluster (prefer L2; if uncertain, use L1). The “cluster” field will include [id, name].
cluster_taxonomy

QA_VERIFIABLE TAXONOMY
---
qa_type_taxonomy

LABELING DEFINITIONS

Clarity (0-5):
0 extremely vague; 1 very vague; 2 vague; 3 workable with small additions; 4 clear (acceptance feasible); 5 very clear (explicit acceptance/tests).

Context (0-5):
0 no repo/env needed; 1 light reference; 2 local file/API awareness; 3 multi-file/module; 4 system-level; 5 deep env/data/service coupling.

Difficulty (0-6):
0 trivial QA; 1 simple single-point; 2 routine; 3 moderately complex (multi-step or multi-file); 4 advanced (design/concurrency/test-heavy); 5 high complexity (tradeoffs, cross-domain, higher risk); 6 extreme system-level with high uncertainty.

WORKFLOW STEPS
---
1) Light repo scan: identify system-level modules, architecture diagrams, abstract base classes, registries, or pipelines.
2) Extract the repository info from the input and output it in the format: ‘"repo": ["owner/repo_name"]‘.
3) Map to cluster aligned with conceptual/system knowledge (algorithms, coordinate systems, unit registries, API design).
4) Generate the developer Query:
- must be a factual “what/how/why” about repo structures or algorithms, not trivial docstring repeats;
- answer must be checkable from code, docs, or API standards.
5) Score clarity, context, difficulty based on how well-defined and system-level the question is.
6) Evidence: point to modules, classes, or specs that define the authoritative structure.
7) NEXT_STEPS: describe how to cross-check the ground truth with codebase or documentation.

Prompt Template for Generating Query Candidates

————————-
QUALITY RULES
————————-
- All queries must require an evidence-backed, canonical answer (e.g., architecture, pipeline design, algorithm complexity).
- The queries should encourage advanced analysis, integration of multiple concept, or insight beyond surface-level information.
- All queries must contain exactly one of the following words: "What", "Why", "Where", or "How". The query must not contain more than one of these words or multiple sub-questions. For example, 'What is the architecture of chartpress's configuration system and how does it coordinate between chartpress.yaml parsing, image building workflows, and values.yaml modification?' is invalid:
- The evidence must explicitly include the relevant line numbers or line ranges for each repo_path.
- Must include a "ground_truth_answer" string in the JSON output, summarizing the verified explanation.
- Reject trivial "what is the type of X" unless it connects to a bigger design concept.
- Prioritize non-trivial, yet verifiable knowledge that reflects the repo's system design or standards compliance.

————————-
USER PREFERENCE
————————-
We HIGHLY suggest you prioritize selecting problems from Cluster swe_issue_qa_1_0, since issues in this repo often fall into this domain. Since the clusters were generated through unsupervised clustering and the labels were assigned based on random sampling and manual annotation within each cluster, there may be inherent bias. If the assigned cluster label conflicts with the reference issues, always treat the reference issues as the source of truth.

To help you understand the typical patterns, here are some example issues from this repo. These are provided only as *reference context* to inform your reasoning.

- If an issue has already been fixed, do not reuse it.
- If an issue is still relevant, you may paraphrase it into a fresh query.
- Ideally, you should write your own problem statement, using the examples only as background knowledge.

It is acceptable to generate a problem outside the recommended cluster if necessary — the examples are guidance, not a restriction.

Repo name:
{repo_name}

Reference issues:
{reference_issues}

Prompt Template for Generating Reference Answer

Model: Claude Code
You are a repository-aware QA answer agent. Your job is NOT to modify code, but to:
(1) Lightly explore the local repository at the given commit using available tools.
(2) Understand and answer the given generated_query based on the actual codebase.
(3) Produce a high-quality, evidence-backed gold-standard answer (refined_ground_truth) that satisfies the five dimensions: correctness, completeness, relevance, clarity, and reasoning quality.
(4) Optionally use reference_answer only as a weak hint, never as ground truth.

Inputs You Will Receive
* ‘repo_name‘
* ‘commit_id‘
* ‘generated_query‘ (the question you must answer)
* ‘reference_answer‘ (may be partially correct, incomplete, or wrong)

Your Objectives
You must produce a final answer that is:
* Fully verified against the repository’s source code.
* Structurally complete, covering all parts required by the query.
* Clear and technically correct, written for developers unfamiliar with the repo.
* Evidence-based, with file paths and line ranges supporting your claims.
* Free from speculation.

Suggestions
The following sections guide your behavior during exploration and answer construction.

1. How to Use ‘reference_answer‘
* Treat ‘reference_answer‘ as an **optional and unreliable hint**.
* It may point to relevant files or concepts, but you must verify everything independently using the actual code.
* You must not summarize, lightly edit, or trust the reference answer.
* If there is any contradiction between code and reference_answer, follow the code.
* The correct mental model is ’reference_answer is a hypothesis; the repository is the truth.’

2. Exploration & Evidence Collection
You must actively explore the repo:
* Navigate to relevant modules, subpackages, core classes, registries, and any architecture files.
* Inspect the implementations, comments, and interfaces relevant to the query.
* Track everything you depend on in ‘evidence.repo_paths‘, using exact format: ‘"path/to/file.py: line X-Y"‘
* Collect signals, which are short text markers such as: class names, method names, configuration patterns, key comments and helper functions
* Every important claim in your ‘refined_ground_truth‘ must be traceable to your collected evidence.

3. Answer Style and Constraints
Your ‘refined_ground_truth‘ must obey:
* No direct code quotations.
* You may name classes/functions/variables but do not copy their bodies.
* Format the answer as coherent paragraphs, not bullet points.
* The answer must be: concise but complete, technically precise, Clear for developers and Grounded in the repository.
* Every major claim must be supported by file paths: line ranges you listed in ‘evidence‘.

Required Output JSON Format
You must output **only** this JSON object: {JSON_EXAMPLE}

Working Procedure (Mental Checklist)
1. Read the ‘generated_query‘ and identify scope (architecture? registration? flow? algorithm?).
2. Lightly scan the repo structure to locate relevant modules.
3. Open related files and gather evidence.
4. Build an internal understanding of the underlying architecture or behavior.
5. Compare your understanding with ‘reference_answer‘: keep what matches the repo, correct what is wrong and add missing key pieces.
6. Write a clean, well-organized gold answer in the ‘refined_ground_truth‘.
7. Fill in ‘evidence‘, ‘rationale_summary‘, and ‘next_steps‘.
8. Output the JSON object.

Input
Repo Name: {repo_name}
Commit ID: {commit_id}
Generated Query: {generated_query}
Reference Answer: {reference_answer}

E Cluster and QA Type Taxonomy

Cluster	Subcluster	Description
Input / Parsing / Data Conversion	Unicode / formatting / parsing / data validation	Character encoding, string normalization, and file format sanitization; focuses on correctness of raw text and low-level input structure.
	SQL / structured grammar / templating engine	SQL syntax, AST grammars, and templating systems; covers grammar rules and templated string generation.
	CLI args / syntax / regex / command completion	Argument parsing, Bash/Zsh completion, and regex issues; applies when user input must be parsed or matched interactively.
	File import/export / metadata / sorting / recurrence	File loading, metadata extraction, sorting logic, and recurrence handling for structured data transfer.
Data / Array / Image / Coordinate	Data formats / FITS / Astropy / units / WCS	Scientific data formats and astronomy-specific coordinate systems, including units and world coordinate systems.
	Pandas / parquet / datetime / Dask / table schema	Tabular data manipulation, schema handling, and time-indexed datasets.
	NumPy / Dask arrays / dtype / serialization / parallel	Numerical array operations, data types, chunking strategies, and array serialization.
	Coordinate transform / image processing / IO / precision	Geometric transformations, image IO, and precision-sensitive numerical processing.
Schema / Types / Validation / Static Analysis	Protocol serialization / encoding / headers / API compatibility	Structured message formats and wire-level compatibility for APIs and protocols.
	Dependency injection / config / attrs / API design	Software design patterns controlling configuration, object construction, and API structure.
	Dataclass / schema validation / enums / OpenAPI	Structured field validation, enum constraints, and OpenAPI specifications.
	Type and attribute errors / initialization / CLI workflow	Runtime failures due to incorrect initialization, attribute access, or object lifecycle misuse.
	Type hints / mypy / typing system / code generation	Static typing, type checking, and auto-generated type stubs.
Packaging / Dependency / Build	Python version / imports / deprecation / conflicts	Import errors, deprecated APIs, and Python version compatibility issues.

Continued on next page

Cluster	Subcluster	Description
	Install / virtual env / OS / hardware / cloud deploy	Environment setup, package installation, OS and hardware requirements, and deployment.
	Artifacts / distribution format / repository management	Wheels, source distributions, repository layout, and post-install package state.
	Extensibility / configuration framework / plugin architecture	Plugin discovery, extension mechanisms, and dynamic component loading.
Docs / CI / Release / Workflow	Version control / Docker / build cache	Git workflows, containerization strategies, and build cache management.
	Release management / changelog / license / community	Release cycles, licensing policies, and community governance.
	Documentation / MkDocs / user tutorials	Systems for generating and maintaining user-facing documentation.
	Async refactor / migration / logging / i18n	Large-scale refactoring, logging infrastructure, and internationalization.
	CI pipelines / coverage / lint / GitHub Actions	Automated testing, linting, security checks, and CI execution.
Runtime / Async / Errors / Resources	Asyncio / async context / resource cleanup	Coroutine scheduling, event loops, async contexts, and cooperative concurrency.
	Multiprocessing / advanced runtime / heterogeneous compute	Process pools, CPU/GPU scheduling, and multi-backend execution.
	Runtime error handling / transactions / retry / logging	Exception handling, rollback mechanisms, and retry strategies.
	Threading / execution limits / scheduling / memory	OS-level threading, memory constraints, and timeout behavior.
	Connection lifecycle / protocol handling / failures	Socket errors, TLS issues, and low-level network failures.
	Parallel execution / distributed frameworks / task graphs	Distributed execution models such as Ray or Dask.
Build Env / Testing / Toolchain	File paths / filesystem / permissions / env config	OS-level filesystem configuration and environment-dependent behavior.
	Unit testing / mocking / test automation	Test frameworks, mocks, and automated verification pipelines.
	Build pipeline / doc building / Sphinx / provisioning	Automated build systems, documentation compilation, and provisioning.
	Compiler toolchain / cross-compile / static analysis	Compiler behavior, environment variables, and code quality analysis.
API / Cloud / Auth / Network	API integration / SDK / performance / DB	External API usage, SDK integration, and performance considerations.
	Media download / playlist / metadata / proxy	Media fetching, metadata extraction, and proxy configuration.

Continued on next page

Cluster	Subcluster	Description
	Auth systems / deployment / cloud plugins	Authentication services and cloud runtime behavior.
	AWS / Azure / Kubernetes / IAM	Infrastructure orchestration and cloud security policies.
	Reverse proxy / routing / websocket / CDN	URL routing, real-time transport, and CDN integration.
	OAuth / JWT / SSL / access control	Token-based authentication, certificates, and session lifecycle.
ML / Algorithms / Performance	Tensors / training / GPU / experiment logging	Model training, GPU execution, and ML experiment management.
	ML visualization / Fourier / calibration	Analytical visualization and mathematical interpretation of models.
	Time series / feature engineering / explainability	Feature extraction, behavioral analysis, and computational semantics.
	Data parallel / compression / ML plugins	Distributed training and compressed data or model representations.
	Bayesian models / MCMC / statistics	Probabilistic modeling and uncertainty-aware inference.
	ML APIs / metrics / optimization strategies	Model interfaces, evaluation metrics, and optimization behavior.
Visualization / UI / Rendering	UI layout / CSS / markdown / frontend security	Layout, formatting, and security of user-facing content.
	Plotting systems / widgets / maps / animation	Charts, interactive widgets, and UI animations.
	Runtime UI config / permissions / uploads	UI customization, permission control, and file upload handling.
	3D rendering / legends / color mapping	Rendering pipelines, color schemes, and legend formatting.

Table 7: Question Cluster taxonomy used in SWE-QA-Pro [Back to Appendix Contents]

Type	Intention	Definition
What	Architecture exploration	Identify components or structural organization of the system.
	Concept / Definition	Understand the meaning or semantics of code elements.
	Dependency tracing	Identify relationships or dependencies among code elements.
Why	Design rationale	Explain why certain design decisions are made.
	Purpose exploration	Understand the intended purpose of a function or component.
	Performance	Understand performance considerations or trade-offs.
Where	Data / Control-flow	Localize variables, data flow, or control statements.
	Feature location	Identify where a specific feature is implemented.
	Identifier location	Find where an identifier is defined or referenced.
How	System design	Explain overall system behavior or execution workflow.
	Algorithm implementation	Understand algorithmic steps or logic implemented in code.
	API / Framework support	Show how APIs or frameworks are used within the system.

Table 8: Taxonomy of Repository-Level Question Intentions

[Back to Appendix Contents]

F Statistics of SWE-QA-Pro

Cluster ID	Cluster Name	Count
0.0	Unicode / formatting / parsing / data validation	5
0.1	SQL / structured grammar / templating Engine	6
0.2	CLI args / syntax / regex / command completion	8
0.3	file import/export / metadata / sorting / recurrence	6
1.0	data formats / FITS / Astropy / units / WCS	4
1.1	pandas / parquet / datetime / Dask / table schema	5
1.2	numpy / dask arrays / dtype / array serialization / parallel	6
1.3	coordinate transform / image processing / IO / numeric precision	4
2.0	protocol serialization / encoding / headers / API compatibility	4
2.1	dependency injection / config / attrs / API design	9
2.2	dataclass / schema validation / enums / OpenAPI	7
2.3	type and attribute errors / initialization / CLI workflow	7
2.4	type hints / mypy / typing system / code generation	7
3.0	Python version / imports / deprecation / conflicts	5
3.1	install / virtual env / OS / hardware requirements / cloud deploy	5
3.2	artifacts / distribution format / repository management / post-install state	4
3.3	extensibility / configuration framework / plugin architecture	9
4.0	version control / Docker / build cache	4
4.1	release management / changelog / license / community	4
4.2	documentation / MkDocs / user tutorials	4
4.3	async refactor / migration / logging infra / i18n	5
4.4	CI pipelines / coverage / lint / GitHub Actions / security checks	8
5.0	asyncio / async context / resource cleanup	7
5.1	multiprocessing / advanced runtime / concurrency / heterogeneous compute	7
5.2	runtime error handling / DB transactions / retry / logging system	8
5.3	threading / execution limits / scheduling / memory / timeout	5
5.4	connection lifecycle / protocol handling / low-level failures	4
5.5	parallel execution / distributed frameworks / task graphs	4
6.0	file paths / filesystem permissions / symlinks / env config / cache system	8
6.1	unit testing / mocking / test automation	5
6.2	build pipeline / doc building / Sphinx / cloud provisioning	5
6.3	compiler toolchain / cross-compile / env vars / code quality analysis	6
7.0	API integration / sync / performance / DB / SDK	7
7.1	media download / playlist / metadata / client-side proxy config	5
7.2	auth systems / deployment / extension plugins / cloud services	5
7.3	AWS / Azure / K8s / container security / IAM policy	4
7.4	reverse proxy / URL routing / websocket / CDN / streaming	5
7.5	OAuth / JWT / SSL / access control / user sessions/ token lifecycle	4
8.0	tensors / training / GPU / ML experiment logging / tuning	4
8.1	ML analytical visualization / Fourier / ML animation / calibration	4
8.2	time series / feature engineering / explainability methods / behavioral analysis / computational semantics	6
8.3	data parallel / compression / ML plugin / indexing	4
8.4	bayesian models / MCMC / statistics / reproducibility	5
8.5	ML APIs / decorators / metrics / optimization strategies	5
9.0	UI layout / CSS / markdown / table extraction / frontend security	4
9.1	plotting systems / widgets / maps / UI animation / usability	5
9.2	runtime UI config / UI permission management / upload handling / customization / user-facing runtime extensibility	4
9.3	3D rendering / legends / color mapping / visualization formatting	4

Table 9: Cluster Statistics (Counts per Question Cluster) [Back to Appendix Contents]

Class Name	Sub-class Name	Count	Total Num
Why (Causal Queries)	Performance & Scalability	33	65
	Design Rationale	24	
	Purpose & Role	8	
What (Factual Queries)	Architecture & Components	20	51
	Dependency & Inheritance	17	
	Concepts & Definitions	14	
How (Procedural Queries)	System Design & Patterns	30	67
	Algorithm Implementation	23	
	API & Framework Support	14	
Where (Localization Queries)	Identifier Location	32	77
	Feature Location	30	
	Data & Control Flow	15	

Table 10: QA Type Statistics [Back to Appendix Contents]

G Breakdown Results in SWE-QA-Pro

G.1 Breakdown Results By QA Type

QA Type Name	Claude Sonnet 4.5	Gemini 2.5 Pro	GPT 4.1	GPT-4o	DeepSeek V3.2	Avg.
Why (Causal Queries)	38.72	38.18	37.65	32.50	36.87	36.78
Performance & Scalability	37.27	37.32	37.69	29.34	35.30	35.39
Design Rationale	41.38	38.49	37.99	35.76	38.68	38.46
Purpose & Role	36.48	40.83	36.46	35.72	37.92	37.48
What (Factual Queries)	39.27	38.59	37.66	30.93	37.95	36.88
Architecture & Components	38.38	38.52	37.82	30.41	37.48	36.52
Dependency & Inheritance	40.57	36.73	37.65	31.78	37.04	36.75
Concepts & Definitions	38.92	40.98	37.45	30.64	39.71	37.54
How (Procedural Queries)	40.86	39.17	38.44	32.81	39.10	38.08
System Design & Patterns	39.07	38.17	37.19	32.94	37.60	36.99
Algorithm Implementation	42.85	40.42	40.45	34.00	40.62	39.67
API & Framework Support	41.64	39.29	37.83	30.62	39.81	37.84
Where (Localization Queries)	42.84	41.36	39.73	35.56	40.36	39.97
Identifier Location	45.32	42.46	41.44	37.59	41.85	41.73
Feature Location	41.90	41.79	39.20	36.71	40.40	40.00
Data & Control Flow	39.58	38.18	37.16	28.95	37.11	36.19

Table 11: Results across Different Question Types by SWE-QA-Pro
[Back to Appendix Contents]

QA Type Name	Qwen3 8B	Qwen3 32B	Devstral Small-2-24B-Ins	Llama 3.3-70B-Ins	SWE-QA-Pro 8B (SFT)	SWE-QA-Pro 8B (SFT+RL)	Avg.
Why (Causal Queries)	31.52	32.83	35.52	26.66	32.35	33.54	32.07
Performance & Scalability	29.55	30.95	33.49	24.64	30.40	30.62	29.94
Design Rationale	33.74	34.68	38.56	28.39	34.66	37.42	34.57
Purpose & Role	33.04	35.00	34.75	29.71	33.46	33.96	33.32
What (Factual Queries)	29.28	30.11	36.50	21.69	32.95	34.17	30.78
Architecture & Components	28.33	29.52	36.53	20.78	32.97	34.87	30.50
Dependency & Inheritance	31.65	30.49	37.08	23.05	31.51	32.10	30.98
Concepts & Definitions	27.76	30.50	35.74	21.35	34.69	35.69	30.96
How (Procedural Queries)	28.43	31.57	37.60	22.43	35.30	35.84	31.86
System Design & Patterns	28.36	32.07	36.19	22.75	34.99	34.92	31.55
Algorithm Implementation	29.12	31.91	39.49	21.83	35.91	37.49	32.63
API & Framework Support	27.48	29.93	37.52	22.74	34.98	35.10	31.29
Where (Localization Queries)	30.67	33.21	39.09	23.67	36.10	37.36	33.35
Identifier Location	32.27	34.67	39.43	24.11	37.36	37.71	34.26
Feature Location	29.29	33.16	39.83	24.50	36.7	38.61	33.68
Data & Control Flow	30.02	30.20	36.89	21.08	32.18	34.13	30.75

Table 12: Results across Different Question Types (Open Models and SWE-QA-Pro Variants)
[Back to Appendix Contents]

G.2 Breakdown Results By Repository Name

Repo Name	Claude Sonnet 4.5	Gemini 2.5 Pro	GPT 4.1	GPT-4o	DeepSeek V3.2	Avg.
PSyclone	38.37	37.97	34.53	25.68	35.20	34.35
Pillow	39.90	37.33	39.33	33.73	37.93	37.64
cekit	42.47	39.77	39.37	32.80	38.47	38.58
checkov	37.10	36.90	34.30	28.40	36.27	34.59
docker-py	44.50	44.23	41.43	39.17	43.13	42.49
dwave-cloud-client	39.87	36.47	36.03	30.25	36.87	35.90
fitbenchmarking	41.47	39.63	40.33	37.83	38.63	39.58
frictionless-py	37.83	40.50	37.70	32.27	38.03	37.27
geopandas	39.80	40.70	38.83	35.42	38.63	38.68
hy	41.57	40.23	41.80	34.10	39.23	39.39
jsonargparse	37.33	33.10	33.13	31.70	34.73	34.00
mkdocs	43.20	40.97	38.83	35.12	41.97	40.02
numba	38.97	38.23	39.00	32.05	38.30	37.31
pennylane	38.37	41.47	37.60	30.48	38.30	37.24
pint	42.73	40.73	39.77	36.33	38.13	39.54
pybryt	40.33	41.77	38.97	35.80	40.43	39.46
qibo	43.27	38.90	39.53	33.48	40.50	39.14
responses	41.10	37.77	38.63	34.50	39.37	38.27
sanic	39.00	41.13	37.43	34.33	37.90	37.96
seaborn	42.97	41.73	39.07	32.95	39.87	39.32
sphinx	41.07	40.20	41.9	34.58	38.60	39.27
sqlfluff	41.60	39.17	38.07	34.90	39.13	38.57
tox	41.07	40.67	40.00	35.40	39.03	39.23
web3.py	40.70	38.90	39.90	33.33	41.03	38.77
xarray	41.00	38.70	37.07	31.25	38.53	37.31
yt-dlp	41.97	38.83	37.70	26.24	37.73	36.49

Table 13: Results across Different Repositories by SWE-QA-Pro

[Back to Appendix Contents]

Repo Name	Qwen3 8B	Qwen3 32B	Devstral Small-2-24B-Ins	Llama 3.3-70B-Ins	SWE-QA-Pro 8B (SFT)	SWE-QA-Pro 8B (SFT+RL)	Avg.
PSyclone	27.20	26.10	37.47	23.86	31.73	33.87	30.04
Pillow	29.60	31.80	36.27	24.50	35.37	32.70	31.71
cekit	28.07	33.27	39.57	23.07	35.87	33.83	32.28
checkov	27.47	29.20	34.43	22.68	33.62	32.73	30.02
docker-py	36.03	40.00	41.47	27.59	40.20	39.70	37.50
dwave-cloud-client	32.37	30.73	33.80	24.52	35.27	36.13	32.14
fitbenchmarking	32.83	34.50	38.43	22.00	38.00	38.90	34.11
frictionless-py	28.03	30.93	36.60	22.40	34.00	35.20	31.19
geopandas	29.43	33.27	37.73	25.90	34.57	35.43	32.72
hy	32.37	31.40	39.00	26.70	36.20	36.17	33.64
jsonargparse	31.17	31.10	29.20	21.35	28.93	32.87	29.10
mkdocs	32.40	33.63	41.43	23.80	36.23	38.33	34.30
numba	29.90	32.10	37.23	24.75	31.69	34.43	31.68
pennylane	28.47	32.23	33.80	18.03	33.13	32.27	29.66
pint	32.10	32.67	39.50	22.10	37.73	40.00	34.02
pybryt	31.37	34.43	39.00	23.65	35.70	37.63	33.63
qibo	26.30	30.10	37.10	22.10	35.70	37.10	31.40
responses	28.87	34.20	36.53	24.00	30.33	34.80	31.46
sanic	30.03	32.60	37.69	25.63	33.70	35.83	32.58
seaborn	30.67	33.00	37.90	24.90	34.13	35.73	32.72
sphinx	30.87	30.70	38.23	24.85	33.70	36.17	32.42
sqlfluff	28.90	30.87	36.30	22.52	33.67	36.00	31.38
tox	29.37	28.83	38.13	24.87	35.30	34.50	31.83
web3.py	28.87	33.17	38.43	23.95	33.93	33.03	31.90
xarray	29.83	31.93	38.33	22.25	34.20	34.13	31.78
yt-dlp	28.40	31.37	36.27	25.30	29.90	32.60	30.64

Table 14: Results across Different Repositories (Open Models and SWE-QA-Pro Variants)
[Back to Appendix Contents]

G.3 Breakdown Results By Cluster

Cluster ID	Name	Claude Sonnet 4.5	Gemini 2.5 Pro	GPT 4.1	GPT-4o	DeepSeek V3.2	Avg.
0.0	Unicode / formatting / parsing / data validation	42.73	41.07	38.07	34.20	38.00	38.81
0.1	SQL / structured grammar / templating Engine	43.06	42.28	38.11	38.00	39.22	40.13
0.2	CLI args / syntax / regex / command completion	40.67	37.21	35.17	33.88	39.08	37.20
0.3	file import/export / metadata / sorting / recurrence	36.20	37.17	34.06	29.75	39.17	35.27
1.0	data formats / FITS / Astropy / units / WCS	44.58	40.25	36.00	37.00	40.58	39.68
1.1	pandas / parquet / datetime / Dask / table schema	39.08	42.27	37.67	30.25	40.93	38.04
1.2	numpy / dask arrays / dtype / array serialization / parallel	39.67	39.11	38.33	32.92	40.56	38.12
1.3	coordinate transform / image processing / IO / numeric precision	40.67	40.58	37.08	32.12	37.17	37.52
2.0	protocol serialization / encoding / headers / API compatibility	39.67	42.75	40.92	36.75	41.42	40.30
2.1	dependency injection / config / attrs / API design	42.42	40.19	38.37	33.86	40.19	39.00
2.2	dataclass / schema validation / enums / OpenAPI	40.95	34.10	33.67	27.96	37.29	34.79
2.3	type and attribute errors / initialization / CLI workflow	44.14	41.90	42.81	35.54	41.33	41.15
2.4	type hints / mypy / typing system / code generation	42.19	40.43	40.86	35.93	39.62	39.80
3.0	Python version / imports / deprecation / conflicts	42.87	39.27	37.87	36.95	38.93	39.18
3.1	install / virtual env / OS / hardware requirements / cloud deploy	39.80	35.67	38.93	29.75	36.27	36.08
3.2	artifacts / distribution format / repository management / post-install state	32.17	32.92	36.42	27.06	30.08	31.73
3.3	extensibility / configuration framework / plugin architecture	38.00	39.37	38.67	34.11	38.63	37.76
4.0	version control / Docker / build cache	41.67	42.50	40.58	36.25	39.33	40.07
4.1	release management / changelog / license / community	37.17	41.83	39.17	34.81	35.67	37.73
4.2	documentation / MkDocs / user tutorials	40.42	39.17	37.50	29.00	39.25	37.07
4.3	async refactor / migration / logging infra / i18n	42.40	38.27	39.87	36.15	40.13	39.36
4.4	CI pipelines / coverage / lint / GitHub Actions / security checks	41.88	40.92	41.12	32.38	40.88	39.43
5.0	asyncio / async context / resource cleanup	40.10	36.38	39.95	31.07	36.48	36.80
5.1	multiprocessing / advanced runtime / concurrency / heterogeneous compute	33.24	37.71	31.43	28.68	33.29	32.87
5.2	runtime error handling / DB transactions / retry / logging system	38.94	39.38	40.71	31.19	39.29	37.90
5.3	threading / execution limits / scheduling / memory / timeout	39.33	32.33	34.27	25.20	32.40	32.71
5.4	connection lifecycle / protocol handling / low-level failures	44.50	43.58	41.67	41.31	41.50	42.51
5.5	parallel execution / distributed frameworks / task graphs	40.50	38.50	39.67	28.50	37.83	37.00
6.0	file paths / filesystem permissions / symlinks / env config / cache system	42.50	40.83	41.50	38.03	42.83	41.14
6.1	unit testing / mocking / test automation	43.60	38.13	39.87	34.75	40.60	39.39
6.2	build pipeline / doc building / Sphinx / cloud provisioning	44.13	43.67	41.53	34.25	35.67	39.85

Continued on next page

Cluster ID	Name	Claude Sonnet 4.5	Gemini 2.5 Pro	GPT 4.1	GPT-4o	DeepSeek V3.2	Avg.
6.3	compiler toolchain / cross-compile / env vars / code quality analysis	42.11	39.28	40.56	36.96	41.17	40.01
7.0	API integration / sync / performance / DB / SDK	42.62	40.67	39.57	34.46	41.43	39.75
7.1	media download / playlist / metadata / client-side proxy config	38.83	38.53	33.93	18.83	35.47	33.12
7.2	auth systems / deployment / extension plug-ins / cloud services	44.47	44.07	43.00	39.15	44.27	42.99
7.3	AWS / Azure / K8s / container security / IAM policy	32.75	37.42	32.67	29.56	33.17	33.11
7.4	reverse proxy / URL routing / websocket / CDN / streaming	44.07	43.60	40.67	39.05	43.20	42.12
7.5	OAuth / JWT / SSL / access control / user sessions / token lifecycle	33.25	33.00	36.42	27.50	36.25	33.28
8.0	tensors / training / GPU / ML experiment logging / tuning	32.56	40.00	32.42	28.44	34.00	33.48
8.1	ML analytical visualization / Fourier / ML animation / calibration	38.25	39.92	38.08	30.38	40.83	37.49
8.2	time series / feature engineering / explainability methods / behavioral analysis / computational semantics	42.06	41.39	38.17	33.62	37.89	38.62
8.3	data parallel / compression / ML plugin / indexing	37.17	34.58	38.83	31.50	32.83	34.98
8.4	bayesian models / MCMC / statistics / reproducibility	42.93	38.13	38.53	30.20	38.13	37.59
8.5	ML APIs / decorators / metrics / optimization strategies	40.00	38.47	38.33	36.50	38.40	38.34
9.0	UI layout / CSS / markdown / table extraction / frontend security	41.33	38.92	39.67	34.56	39.83	38.86
9.1	plotting systems / widgets / maps / UI animation / usability	43.60	44.27	42.80	40.40	42.00	42.61
9.2	runtime UI config / UI permission management / upload handling / customization / user-facing runtime extensibility	31.44	39.00	33.75	26.40	33.08	32.74
9.3	3D rendering / legends / color mapping / visualization formatting	45.92	43.83	44.50	41.44	41.58	43.45

Table 15: Results across Question Clusters (Closed Models)

[Back to Appendix Contents]

Cluster ID	Name	Qwen3 8B	Qwen3 32B	Devstral Small-2-24B-Ins	Llama 3.3-70B-Ins	SWE-QA-Pro 8B (SFT)	SWE-QA-Pro 8B (SFT+RL)	Avg.
0.0	Unicode / formatting / parsing / data validation	31.47	33.40	38.87	24.17	39.53	38.40	34.31
0.1	SQL / structured grammar / templating Engine	30.00	32.78	39.44	21.76	35.33	38.89	33.03
0.2	CLI args / syntax / regex / command completion	29.79	30.29	31.12	22.72	33.54	35.54	30.50
0.3	file import/export / metadata / sorting / recurrence	31.33	35.61	36.28	23.67	34.00	33.00	32.31
1.0	data formats / FITS / Astropy / units / WCS	30.50	28.42	38.42	20.38	34.17	42.83	32.45
1.1	pandas / parquet / datetime / Dask / table schema	25.67	27.20	40.07	24.45	35.80	35.87	31.51
1.2	numpy / dask arrays / dtype / array serialization / parallel	30.00	34.33	38.94	24.38	37.22	34.33	33.20
1.3	coordinate transform / image processing / IO / numeric precision	28.25	31.33	36.83	20.88	34.58	33.17	30.84
2.0	protocol serialization / encoding / headers / API compatibility	30.58	36.17	40.67	20.10	36.42	35.25	33.20
2.1	dependency injection / config / attrs / API design	31.00	33.22	38.70	23.23	35.56	36.30	33.00
2.2	dataclass / schema validation / enums / OpenAPI	25.76	29.48	35.81	23.81	30.57	31.90	29.56
2.3	type and attribute errors / initialization / CLI workflow	33.57	26.38	39.10	26.45	33.86	36.62	32.66
2.4	type hints / mypy / typing system / code generation	36.19	32.95	40.19	27.53	35.05	35.62	34.59
3.0	Python version / imports / deprecation / conflicts	37.07	35.27	41.67	21.00	34.60	33.20	33.80
3.1	install / virtual env / OS / hardware requirements / cloud deploy	26.40	28.80	39.60	24.36	30.80	30.33	30.05
3.2	artifacts / distribution format / repository management / post-install state	25.92	27.92	28.50	20.50	25.00	27.50	25.89
3.3	extensibility / configuration framework / plugin architecture	29.48	31.89	36.00	25.61	32.81	35.96	31.96
4.0	version control / Docker / build cache	31.00	27.08	41.33	25.67	38.08	40.33	33.92
4.1	release management / changelog / license / community	30.75	30.83	38.92	22.33	32.17	29.08	30.68
4.2	documentation / MkDocs / user tutorials	28.33	30.83	37.67	24.20	29.25	31.00	30.21
4.3	async refactor / migration / logging infra / i18n	31.87	34.07	40.60	23.20	36.67	39.67	34.34
4.4	CI pipelines / coverage / lint / GitHub Actions / security checks	27.88	31.75	37.62	22.94	39.96	38.04	33.03
5.0	asyncio / async context / resource cleanup	27.05	27.48	37.52	24.00	31.71	31.48	29.87
5.1	multiprocessing / advanced runtime / concurrency / heterogeneous compute	29.10	30.71	30.81	23.47	28.29	27.57	28.32
5.2	runtime error handling / DB transactions / retry / logging system	31.21	32.00	37.42	22.61	33.29	37.54	32.34
5.3	threading / execution limits / scheduling / memory / timeout	30.07	32.33	29.67	24.38	28.53	28.47	28.91
5.4	connection lifecycle / protocol handling / low-level failures	34.83	41.92	42.91	25.00	36.75	39.08	36.75
5.5	parallel execution / distributed frameworks / task graphs	27.75	32.58	38.17	22.82	32.55	36.25	31.69

Continued on next page

Cluster ID	Name	Qwen3 8B	Qwen3 32B	Devstral Small-2-24B-Ins	Llama 3.3-70B-Ins	SWE-QA-Pro 8B (SFT)	SWE-QA-Pro 8B (SFT+RL)	Avg.
6.0	file paths / filesystem permissions / symlinks / env config / cache system	33.46	32.88	40.79	25.21	37.96	40.00	35.05
6.1	unit testing / mocking / test automation	31.67	34.40	37.87	20.17	41.47	40.27	34.31
6.2	build pipeline / doc building / Sphinx / cloud provisioning	31.80	31.33	39.13	25.50	35.20	40.87	33.97
6.3	compiler toolchain / cross-compile / env vars / code quality analysis	31.67	35.78	34.44	24.38	35.41	35.89	32.93
7.0	API integration / sync / performance / DB / SDK	28.00	35.00	38.05	24.33	40.00	39.24	34.10
7.1	media download / playlist / metadata / client-side proxy config	23.60	28.60	32.67	24.64	24.53	28.07	27.02
7.2	auth systems / deployment / extension plugins / cloud services	35.20	41.27	42.80	28.29	44.13	37.73	38.24
7.3	AWS / Azure / K8s / container security / IAM policy	27.08	26.67	33.83	20.22	33.08	32.00	28.81
7.4	reverse proxy / URL routing / websocket / CDN / streaming	37.27	38.60	40.53	26.00	38.07	42.27	37.12
7.5	OAuth / JWT / SSL / access control / user sessions / token lifecycle	25.42	30.08	28.83	24.80	25.58	29.50	27.37
8.0	tensors / training / GPU / ML experiment logging / tuning	28.67	27.58	28.33	17.64	28.75	33.58	27.43
8.1	ML analytical visualization / Fourier / ML animation / calibration	25.50	23.17	34.08	22.25	29.58	32.50	27.85
8.2	time series / feature engineering / explainability methods / behavioral analysis / computational semantics	29.50	32.44	40.61	23.50	37.22	39.39	33.78
8.3	data parallel / compression / ML plugin / indexing	28.67	30.58	34.50	22.10	31.83	26.67	29.06
8.4	bayesian models / MCMC / statistics / reproducibility	29.47	32.00	31.87	22.82	31.60	31.20	29.83
8.5	ML APIs / decorators / metrics / optimization strategies	29.33	34.53	39.93	21.33	37.13	38.07	33.39
9.0	UI layout / CSS / markdown / table extraction / frontend security	26.50	30.58	38.33	25.56	35.17	37.58	32.29
9.1	plotting systems / widgets / maps / UI animation / usability	35.80	40.73	42.60	29.54	37.53	39.87	37.68
9.2	runtime UI config / UI permission management / upload handling / customization / user-facing runtime extensibility	26.83	33.17	33.33	22.38	31.75	33.67	30.19
9.3	3D rendering / legends / color mapping / visualization formatting	25.92	30.50	43.92	24.09	36.83	38.58	33.31

Table 16: Results across Question Clusters (Open Models and SWE-QA-Pro Variants)
[Back to Appendix Contents]

H Case Study

H.1 Cases of Model Comparison

Comparison between GPT-4o and SWE-QA-8B (SFT+RL)

Repo Name: dwavesystems/dwave-cloud-client
Generated Query: Where does coverage data flow from test execution to the external reporting service in the current CI pipeline?

Ground Truth:
During test execution, coverage data is generated via the 'coverage run -m unittest discover' command (.travis.yml: line 46), which instruments the test runs to collect code coverage metrics. The coverage collection is configured by .coveragerc, which specifies that coverage should track the 'dwave.cloud' source package (line 2) while omitting virtual environments, test files, performance tests, and the package_info.py file (lines 3-6). The necessary dependencies for this process, including both 'coverage' and 'coveralls' packages, are installed during the CI setup phase as specified in tests/requirements.txt and installed via .travis.yml. After successful test completion, the collected coverage data is transmitted to the external Coveralls reporting service through the 'coveralls' command executed in the after_success phase (.travis.yml: line 49).

Answer from GPT-4o (abridged):
In the current CI pipeline … coverage data flows … as follows: …
The appveyor.yml file installs … including coverage and coveralls. However, it does not explicitly invoke coveralls, suggesting that the upload process might be handled implicitly … …
No explicit script or command for invoking coveralls was found in the repository …
Score: total_score: 22, correctness: 2, completeness: 2, relevance: 6, clarity: 8, reasoning: 4

Answer from SWE-QA-8B (SFT+RL) (abridged):
Coverage data flows … through the following path … …
The CI pipeline runs tests using … coverage run -m unittest discover … in the .travis.yml file (line 46). …
After successful test execution, the coveralls command (line 49 in .travis.yml) sends the coverage data to Coveralls.io …
The flow is: Tests → … → Coveralls CLI → Coveralls.io Service
Score: total_score: 50, correctness: 10, completeness: 10, relevance: 10, clarity: 10, reasoning: 10

Scoring Rationale :
SWE-QA-8B (SFT+RL) explicitly matches the ground-truth flow (coverage run -m unittest discover in .travis.yml, then coveralls in after_success). GPT-4o is not just less aligned—it is wrong: it claims "No explicit … command for invoking coveralls was found" and suggests an "implicit" upload, even though .travis.yml explicitly runs coveralls. It also mis-centers appveyor.yml as the relevant CI path for uploading coverage, which contradicts the ground truth.

Figure 6: Example of a comparison of GPT-4o and SWE-QA-8B (SFT+RL) answers against the ground-truth.
[Back to Appendix Contents]

Comparison between SWE-QA-8B (SFT) and SWE-QA-8B (SFT+RL)

Repo Name: pydata/xarray
Generated Query: Where are the CRS format conversion methods (PROJ.4 vs WKT string) defined and implemented in xarray's rasterio backend?

Ground Truth:
The CRS format conversion logic is implemented in the open_rasterio function in xarray/backends/rasterio_.py. When constructing the output attributes, the code checks whether the rasterio dataset has a CRS and serializes it to a string by first attempting to call riods.crs.to_proj4() to obtain a PROJ.4 representation, and if this method is unavailable (raising AttributeError), falling back to riods.crs.to_string(), which returns a WKT representation; the resulting CRS string is stored in the returned DataArray's attributes under the crs key (xarray/backends/rasterio_.py: line 284–291, 341–342). A second CRS serialization occurs when the input is a WarpedVRT, in which case vrt.crs.to_string() is used when building the VRT parameters so that the virtual raster's CRS is preserved during reopening (xarray/backends/rasterio_.py: line 220–222).

Answer from SWE-QA-8B (SFT) (abridged):
The CRS format conversion methods … are defined and implemented in … xarray/backends/rasterio_.py. …
The conversion methods are: … to_proj4() … to_string() …
These methods are used in two key locations: …
… WarpedVRT … the CRS is converted to a string (PROJ.4 format) using vrt.crs.to_string() …
… the CRS is converted to both PROJ.4 and WKT formats using riods.crs.to_proj4() and riods.crs.to_string(), and stored as attributes …
The implementation delegates to rasterio's built-in CRS methods …
Score: total_score: 29, correctness: 3, completeness: 4, relevance: 9, clarity: 8, reasoning: 5

Answer from SWE-QA-8B (SFT+RL) (abridged):
The CRS format conversion methods … are … in xarray/backends/rasterio_.py. …
to_proj4() … used … to convert … to a PROJ.4 string …
to_string() … used … to convert … to a WKT string …
… when creating a DataArray … uses riods.crs.to_proj4() …
… if to_proj4() raises an AttributeError, it falls back to riods.crs.to_string() for WKT format …
… when creating a WarpedVRT … uses vrt.crs.to_string() …
… conversions are not implemented by xarray itself but delegate to … rasterio …
Score: total_score: 46, correctness: 9, completeness: 9, relevance: 10, clarity: 9, reasoning: 9

Scoring Rationale :
SWE-QA-8B (SFT+RL) matches the ground-truth control flow: it states the try to_proj4() then fallback to to_string() on AttributeError, and it places vrt.crs.to_string() in the WarpedVRT path. SWE-QA-8B (SFT) is weaker against the ground truth and also internally inconsistent: it labels vrt.crs.to_string() as "PROJ.4 format," even though to_string() is the WKT-style serialization path in the ground truth. It also claims the CRS is converted to "both PROJ.4 and WKT … and stored as attributes," which conflicts with the ground-truth logic that stores one CRS string (PROJ.4 if available, otherwise WKT) under a single crs attribute.

Figure 7: Example of Comparison between Reference Answer and Human Annotated Answer (web3.py).
[Back to Appendix Contents]

H.2 Case Studies of Human and Model Reference Answers

Session ID: ethereum__web3.py-3027-0773ab80-e07c-4730-a349-6f9eef2f0c76

Generated Query: Where is the AF_UNIX socket path validation logic implemented that generates 'AF_UNIX path too long' errors in the IPC provider system?
Generated Answer: The AF_UNIX socket path validation that generates 'AF_UNIX path too long' errors is not explicitly implemented in web3.py code, but rather occurs at the operating system level during socket connection. The error originates in the get_ipc_socket() function (web3/providers/ipc.py: line 35-47) where socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) creates a Unix domain socket and sock.connect(ipc_path) attempts the connection at line 45. When the path exceeds the system limit (typically 108 characters on Unix systems), the OS raises an OSError with the message 'AF_UNIX path too long'. The IPC path itself is constructed by get_default_ipc_path() (web3/providers/ipc.py: line 89-114) using platform-specific logic for macOS, Linux, FreeBSD, and Windows. This function is invoked during IPCProvider initialization (web3/providers/ipc.py: line 149-170) when no explicit path is provided. The AutoProvider class (web3/providers/auto.py: line 63-69) includes IPCProvider in its default_providers list and attempts initialization through _get_active_provider() (web3/providers/auto.py: line 117-122). The RequestManager (web3/manager.py: line 108-111) uses AutoProvider as the default when no provider is specified. When the path length error occurs, it manifests as an OSError that gets caught by the JSONBaseProvider.is_connected() method (web3/providers/base.py: line 120-130), which handles OSErrors during connection attempts by either returning False or raising a ProviderConnectionError with traceback details.

Ground Truth: The AF_UNIX socket path validation that generates 'AF_UNIX path too long' errors is not explicitly implemented in web3.py code, but rather occurs at the operating system level during socket connection. The error originates in the get_ipc_socket() function (web3/providers/ipc.py: line 35–47) where socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) creates a Unix domain socket and sock.connect(ipc_path) attempts the connection at line 45. When the path exceeds the system limit (typically 108 characters on Unix systems), the OS raises an OSError with the message 'AF_UNIX path too long'. The IPC path itself is constructed by get_default_ipc_path() (web3/providers/ipc.py: line 89–114) using platform-specific logic for macOS, Linux, FreeBSD, and Windows. This function is invoked during IPCProvider initialization (web3/providers/ipc.py: line 149–170) when no explicit path is provided. The AutoProvider class (web3/providers/auto.py: line 63–69) includes IPCProvider in its default_providers list and attempts initialization through _get_active_provider() (web3/providers/auto.py: line 113–123). The RequestManager (web3/manager.py: line 108–111) uses AutoProvider as the default when no provider is specified. When the path length error occurs, it manifests as an OSError that gets caught by the JSONBaseProvider.is_connected() method (web3/providers/base.py: line 111–133), which handles OSErrors during connection attempts by either returning False or raising a ProviderConnectionError with traceback details

Error Type: Factual Error
Error Reason: The original answer factually misidentified the error handling mechanism by citing lines 120–130 in JSONBaseProvider.is_connected. This code block is responsible for parsing application-level JSON-RPC errors (checking for an 'error' key in the response payload), not low-level connection failures. It failed to locate the actual try/except block (lines 112–115) that explicitly captures the OS-level OSError triggered by the socket path limit, effectively conflating protocol payload validation with transport exception handling.

Figure 8: Example of a comparison of SWE-QA-8B (SFT) and SWE-QA-8B (SFT+RL) answers.
[Back to Appendix Contents]

Session ID: frictionlessdata__frictionless-py-1156-b1134404-d537-4caa-999d-88f3b0969356

Generated Query: How does the schema detection algorithm implement field type inference and scoring for tabular data in the Detector class?
Generated Answer: The schema detection algorithm implements field type inference through a sophisticated multi-candidate scoring system orchestrated by the Detector class. The process begins by creating field type candidates from DEFAULT_CANDIDATES, a prioritized list containing 14 types ordered from most specific to most general: yearmonth, geopoint, duration, geojson, object, array, datetime, time, date, integer, number, boolean, year, and string (/testbed/frictionless/settings.py: line 71-86). For each data column, the algorithm initializes runners containing all candidate field types (/testbed/frictionless/detector/detector.py: line 487-500). During inference, the algorithm processes each row of the sample data and evaluates every candidate against each cell using the Field.read_cell() method (/testbed/frictionless/field.py: line 323-362). The scoring mechanism awards +1 point for successful cell parsing (when read_cell returns no error notes) and -1 point for parsing failures (/testbed/frictionless/detector/detector.py: line 519). The algorithm implements early elimination optimization by calculating a threshold value as len(fragment) * (field_confidence - 1) and skipping candidates whose scores fall below this threshold (/testbed/frictionless/detector/detector.py: line 505, 515). A candidate wins and becomes the inferred field type when its score reaches or exceeds max_possible_score * field_confidence, where max_possible_score accounts for missing values in the data (/testbed/frictionless/detector/detector.py: line 520-527). The default field_confidence is 0.9, meaning a candidate must successfully parse 90% of valid (non-missing) cells to be selected. This approach ensures robust type inference while maintaining performance through early elimination of poorly-performing candidates.

Ground Truth: The schema detection algorithm implements field type inference through a sophisticated multi-candidate scoring system orchestrated by the "detect_schema" method in the Detector class. The process begins by creating field type candidates from "DEFAULT_CANDIDATES", a prioritized list containing 14 types ordered from most specific to most general (e.g., yearmonth, geopoint... string) (frictionless/settings.py: line 71-86). For each data column, the algorithm initializes "runners" which track the state of every candidate field type (frictionless/detector/detector.py: line 485-500). During inference, the algorithm processes each row of the data fragment. It evaluates every candidate against each cell using the **"Field.read_cell()"** method, which parses the value and returns any type discrepancies (frictionless/field.py: line 350-361). The scoring mechanism awards +1 point for successful cell parsing and -1 point for parsing failures (frictionless/detector/detector.py: line 519).To optimize performance, the algorithm implements early elimination by calculating a "threshold" value based on the fragment length and confidence level; candidates dropping below this threshold are skipped in subsequent iterations (frictionless/detector/detector.py: line 505, 515). Finally, a candidate wins and is assigned as the inferred field type when its score meets the confidence requirement (calculated against a "max_score" that accounts for missing values) (frictionless/detector/detector.py: line 512-527).

Error Type: Localization Hallucination
Error Reason: The original answer accurately described the multi-candidate scoring logic but suffered from severe localization hallucination, citing the runners initialization at lines 346-362 based on an outdated file version. It failed to verify the code against the current commit, overlooking significant docstring additions that shifted the actual implementation to lines 485-500, resulting in a ~140-line discrepancy that rendered the supporting evidence invalid.

Figure 9: Example of Comparison between Reference Answer and Human Annotated Answer (fritionless-py).
[Back to Appendix Contents]

Session ID: hylang__hy-2514-955a46b9-91d6-46ec-98cb-2a41fbff46bd

Generated Query: Why does Hy's conditional compilation system generate redundant anonymous variable assignments instead of optimizing nested conditionals into efficient elif chains?
Generated Answer: Hy's conditional compilation system generates redundant anonymous variable assignments due to a fundamental architectural decision that prioritizes semantic correctness over performance optimization. The redundancy stems from two key design patterns. First, the cond macro recursively expands into deeply nested if expressions rather than a flat elif-like structure. The _cond function in hy/core/macros.hy: line 36-41 demonstrates this recursive expansion where each condition-result pair generates a nested if form that calls _cond again for remaining conditions. This creates a tree of nested conditionals rather than a linear chain. Second, the compile_if function in hy/core/result_macros.py: line 643-707 handles the fundamental mismatch between Hy's everything-is-an-expression philosophy and Python's separation of statements and expressions. When conditional bodies contain statements rather than pure expressions, the compiler creates anonymous variables using get_anon_var() (line 681) and generates assignment statements (lines 685-689) to store intermediate results. Each nested conditional receives its own temporary variable and assignment rather than being optimized into a single elif chain because the compiler must ensure that each branch can be used in both expression and statement contexts. The Result class architecture in hy/compiler.py: line 144-309 enforces this separation by maintaining both a statements list and an expression context, requiring assignments to bridge between them. This design choice ensures that complex nested conditionals with mixed statements and expressions maintain correct evaluation semantics, but at the cost of generating additional temporary variables and assignments that could theoretically be optimized away in simpler cases.

Ground Truth: Hy's conditional compilation system generates redundant anonymous variable assignments primarily because the "cond" macro recursively expands into nested "if" expressions (hy/core/macros.hy: line 36-41) that do not utilize the compiler's internal "if*" optimization symbol. While the "compile_if" function (hy/core/result_macros.py: line 643-707) includes logic to reuse temporary variables for nested conditionals (specifically checking for "if*" at line 652), the standard "cond" macro emits standard "if" forms, causing this optimization to be skipped. Consequently, "compile_if" treats each nesting level as a separate statement-to-expression conversion context. To bridge Hy's expression-oriented semantics with Python's statement-oriented architecture, the compiler is forced to create a new temporary variable via "get_anon_var" (line 681) and generate explicit assignment statements (lines 685-689) for each level. This behavior is structurally enforced by the "Result" class (hy/compiler.py: line 144-309), which separates statements from expression contexts, necessitating these assignments to maintain correct evaluation semantics across nested statement blocks.

Error Type: Reasoning Depth
Error Reason: The original answer superficially blamed design philosophy, overlooking the compiler's internal if* optimization mechanism. It failed to identify the root cause: the cond macro expands into standard if forms, inadvertently bypassing the built-in optimization path designed to prevent redundancy.

Figure 10: Example of Comparison between Reference Answer and Human Annotated Answer (hy).
[Back to Appendix Contents]

I Ethics and Reproducibility Statements

I.1 Potential Risks

This work focuses on training and evaluating large language models for repository-level code understanding and question answering, using real-world open-source project code. While all questions are grounded in executable repositories and paired with validated reference answers, models may still produce incomplete or misleading responses when codebase exploration or tool usage fails, particularly in cases of implicit reasoning errors or silent failures. This work involves only publicly available open-source repositories and does not include any personal, sensitive, or user-generated content.

I.2 Discuss the License for Artifacts

All released artifacts are provided under permissive licenses suitable for academic research. License terms permit use, modification, and redistribution in accordance with each license's conditions.

I.3 Artifact Use Consistent With Intended Use

All external datasets and software components were used in accordance with their original license agreements and intended purposes. Derived artifacts are intended solely for research and educational use, and are not authorized for commercial deployment or redistribution.

I.4 Data Contains Personally Identifying Info or Offensive Content

All data was either synthetically generated or obtained from public sources. Automated filters and manual review were applied to ensure that no samples contain personally identifying information or offensive content. All instructions and tables are free of references to real individuals, groups, or sensitive contexts.

I.5 Documentation of Artifacts

All released artifacts are accompanied by documentation describing their structure, content format, intended use, and evaluation methodology. Sufficient metadata and usage instructions are provided to support inspection, reproduction, and downstream research use.

I.6 Parameters for Packages

All external packages used during training and evaluation were applied in accordance with standard practices. Default parameters were used unless otherwise specified. Any deviations from default settings are documented in the accompanying implementation materials.

I.7 Data Consent

This work exclusively uses data derived from publicly available open-source software repositories and their associated issue trackers. All data was collected in accordance with the repositories' publicly stated licenses and terms of use, which permit research, analysis, and redistribution. As no private, restricted, or user-submitted personal data is included, explicit individual consent was not required. The data collection and curation process does not involve interaction with repository contributors, nor does it introduce new uses beyond the original public and open-source context.

I.8 AI Assistants in Research or Writing

Used ChatGPT to capture grammar errors in the manuscript.