Hindsight: A Git Intelligence Layer

Vanshika Agrawal
vanagrawal@umass.edu

Mustafa Ali
mustafaali@umass.edu

Priyamvadha Balakrishnan
pbalakrishna@umass.edu

Sheng-Kai Wen
shengkaiwen@umass.edu

Abstract

Large software repositories contain complex dependencies across files and modules, making code understanding difficult for both developers and AI systems. Existing code assistants primarily rely on text similarity and often overlook structural relationships within repositories. We propose **Hindsight**, a Git intelligence layer that enhances retrieval-augmented generation (RAG) with a knowledge graph representation of repository structure. The system combines BM25 lexical retrieval with graph-based context expansion, using function-level call graphs to surface cross-file dependencies that keyword search cannot reach. We evaluate multiple graph construction strategies including AST-based call graphs, RepoGraph (Xiao et al., 2024), DKB, and CGM (Tao et al., 2025) and analyze their effect on retrieval quality and answer generation for multi-file code understanding tasks. We evaluate the approach on DeepCodeBench (Qodo, 2025), and LocBench, using FactRecall@k and Acc@k to measure improvements in retrieval and localization.

1 Introduction

Problem. Modern software repositories are complex systems where functionality is distributed across multiple files, modules, and dependencies. Developers often need to answer questions that require reasoning across these components, such as understanding why a test fails, how a feature is implemented, or what might break after a code change. As codebases grow in size and complexity, enabling effective repository-level understanding becomes a key challenge for both developers and AI-assisted tools.

Recent approaches based on large language models and retrieval-augmented generation have shown promise for code understanding tasks. However, most existing systems rely primarily on textual similarity or embedding-based retrieval, treating code as independent chunks. This limits their ability to capture structural relationships such as function calls, dependencies, and cross-file interactions. As a result, they often retrieve incomplete or misleading context for queries that require multi-file reasoning. Benchmarks such as DeepCodeBench (Qodo, 2025) and LocBench (Chen et al., 2025b) highlight that retrieval quality is a major bottleneck for repository-level tasks.

Motivation. In practice, developers encounter different types of queries when working with large codebases. **Leading indicators** involve anticipating the impact of changes (e.g., which modules depend on a function), while **lagging indicators** arise during debugging, requiring tracing failures across multiple components. **Exploratory queries** focus on understanding unfamiliar parts of a system. All these scenarios require navigating relationships across files rather than inspecting isolated snippets. A standard retriever such as BM25 may identify an initial file relevant to the query, but the true answer often depends on related files connected through repository structure (e.g., dependencies, imports, or utility functions). Without incorporating these relationships, retrieval remains incomplete.

Our approach. We propose a hybrid retrieval framework that combines lexical retrieval with structure-aware graph expansion. As shown in Figure 1, we start from initial candidates, such as BM25 results, and traverse a repository knowledge graph to retrieve related files, creating richer multi-file context for downstream reasoning.

Our early results suggest that structural signals are most useful when questions require cross-file reasoning. On DeepCodeBench, HindSight v1 is competitive with BM25 and CodeRankEmbed overall, while outperforming both methods on broad-scope questions where evidence is distributed across multiple files. On LocBench and DeepCodeBench lo-

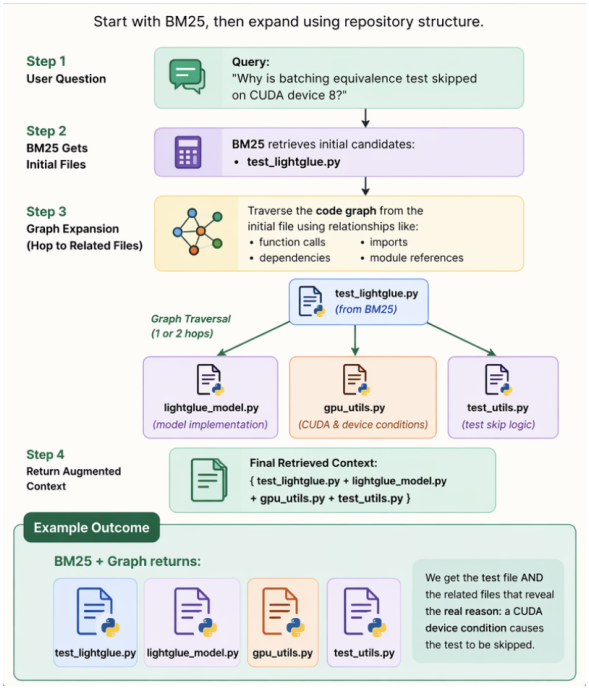


Figure 1: Overview of our hybrid retrieval approach. BM25 retrieves initial candidate files, which are then expanded through repository graph traversal over calls, imports, and dependencies to build richer multi-file context for answer generation.

calization, graph-guided methods improve top-1 file localization, although BM25 remains stronger at finer module and function granularity. Overall, these findings suggest that retrieval quality and context ranking, not model capability alone, are key bottlenecks in repository-level understanding.

2 Related Work

Code question answering. Early benchmarks such as CodeQA focus on QA over individual functions or small snippets (Zhang et al., 2019), while CodeRepoQA extends this setting to repository-level questions across multiple files (Peng et al., 2021). More recent benchmarks, including DeepCodeBench, SpiderCodeQA, and CoReQA, emphasize realistic multi-file reasoning grounded in PRs, repository semantics, metadata, and GitHub issues (Qodo, 2025; Anonymous, 2024b; Chen et al., 2025a). These benchmarks show that repository QA requires retrieving and reasoning over related files, but retrieval is often treated as a black box rather than explicitly modeling structural code relations.

Code retrieval and repository understanding. CoIR highlights the limits of purely lexical or embedding-based retrieval for developer queries that depend on repository structure (Wang et al., 2023). DependEval similarly evaluates whether models can identify dependency relationships across files (Du et al., 2025). Together, these works motivate structure-aware retrieval, since many pipelines still operate over flat text chunks and rely mainly on similarity rather than explicit code relationships.

Repository-level evaluation benchmarks. DeepCodeBench evaluates multi-file QA with grounded supporting facts (Qodo, 2025), SWE-bench studies issue resolution and patch generation (Jimenez et al., 2023), and LocBench evaluates localization of relevant files or functions from issue descriptions (Anonymous, 2024a). These benchmarks show that accurate localization is a key precursor to repository-level reasoning. In this work, we focus on improving retrieval and localization quality rather than full end-to-end issue solving.

Graph-based localization and structure-aware methods. LocAgent is closest to our work, using repository structure for graph-guided localization on LocBench (Chen et al., 2025b). Other work represents repositories through dependency graphs, AST-based structures, RepoGraph (Xiao et al., 2024), or Code Graph Models (Tao et al., 2025). Our approach integrates structural signals directly into retrieval through intent-guided graph traversal, combining first-stage retrieval such as BM25 with lightweight graph expansion for multi-file code understanding.

3 Method

3.1 Baseline Analysis

We first analyze a set of baseline methods under a unified evaluation setting. Our baselines include minimal-retrieval controls, a sparse retriever (BM25), a dense retriever (CodeRankEmbed) (Suresh et al., 2024), and LocAgent (Chen et al., 2025b) as a graph-guided baseline. All methods use the same repository snapshots, chunking strategy, prompt template, context budget, and evaluation splits, so observed differences can be attributed primarily to retrieval quality rather than to preprocessing or prompting. We categorize codebase queries into two types that impose distinct retrieval demands.

Leading-indicator query. A query asked before an observed failure, usually while planning or making a change. It anticipates impact, traces downstream consequences, or helps design an implementation. Examples include asking what depends on an API, what happens if a code path returns a certain value, or what must change to add a feature.

Lagging-indicator query. A query asked after an unexpected bug, failure, or regression has already appeared. It is diagnostic: the developer is trying to explain what broke, why incorrect behavior occurred, or what code path caused the symptom.

Exploratory query. A query asked to understand the codebase without planning a change or debugging an active failure. It asks factual or structural questions such as what a component does, where logic is implemented, how a mechanism works, or which code was changed for a past issue.

We study baseline performance along two dimensions. First, we compare results on *leading-indicator*, *lagging-indicator* and *exploratory* queries. Second, we separate single-file and multi-file questions to test whether structural retrieval is especially beneficial when evidence is distributed across the repository. In addition to aggregate metrics, we perform qualitative error analysis to identify whether failures arise from weak lexical matching, insufficient semantic retrieval, or inability to recover cross-file relationships.

3.2 Proposed Method

Our proposed method, **HindSight**, builds on the code graph introduced in LocAgent (Chen et al., 2025b) and extends it for repository-level question answering. LocAgent constructs a heterogeneous directed graph $G(V, E, A, R)$, where V is the set of code entities, $E \subseteq V \times V$ is the edge set, $A = \{\texttt{directory}, \texttt{file}, \texttt{class}, \texttt{function}\}$ is the set of node types, and $R = \{\texttt{contain}, \texttt{import}, \texttt{invoke}, \texttt{inherit}\}$ is the set of relation types.
Given a query q, retrieval proceeds through the LocAgent multi-turn loop, which operates over a NetworkX heterogeneous dependency graph and a BM25 term index built offline per repository. The agent iteratively calls three tools, `search_code_snippets` (BM25 + graph traversal), `explore_tree_structure` (directory listing), and `get_entity_contents` (source retrieval), until it converges on a candidate set of localized code entities. HindSight introduces two modifications to this baseline loop: a query decomposition step and an intent-aware query classification step that together shape how the agent allocates its tool calls and traverses the combined index.

Query decomposition. Complex natural-language questions about repository internals often bundle multiple retrieval intents into a single query. For example, a question such as "Which helper function is used to convert object detection model outputs into filtered boxes, scores, and labels in the integration tests?" implicitly requires locating the integration test files, identifying the relevant model output processing logic, and linking the two. To address this, we augment the LocAgent loop with an LLM-based decomposition step that breaks the original query q into a set of focused subquestions $\{q_1, \ldots, q_n\}$ before retrieval begins. These subquestions guide the agent's tool calls within a single localization run, providing more precise lexical anchors for BM25 and more targeted entry points for graph traversal over both the AST and RepoGraph indexes.

Query classification. After decomposition, HindSight classifies each query along two axes using an LLM-based classifier (GPT-5.5 in reasoning mode). The first axis assigns an *intent label* $z \in \{\texttt{leading}, \texttt{lagging}, \texttt{exploratory}\}$, distinguishing planning-oriented queries about prospective changes from diagnostic queries about observed failures and from open-ended structural queries about the codebase. Queries labeled as exploratory are further classified along the DeepCodeBench taxonomy of *core vs. non-core*, *searchable vs. non-searchable*, and *broad vs. deep scope* (**?**). The resulting class signature is passed to the agent loop and shapes graph traversal: leading queries bias traversal toward `import` and `invoke` edges originating from the candidate files (to surface downstream dependencies of a planned change); lagging queries bias traversal toward incoming `invoke` and `contain` edges (to trace symptoms back to their root cause); and exploratory queries use the fine-grained DeepCodeBench labels to decide whether to expand broadly across files via the RepoGraph symbol index (*broad*, *non-core*) or to descend into a single file's class and function hierarchy via the AST index (*deep*, *core*). This

intent-conditioned traversal is the central design choice that distinguishes HindSight from the lexical-first retrieval baselines.
The final candidate set produced by the agent loop is passed directly to the answer-generation stage,

$$\hat{y} = \mathcal{M}(q, R(q)),$$

where $\mathcal{M}$ is the language model and $R(q)$ are the retrieved code snippets together with their minimal metadata. We discuss an additional test-linkage index and an intent-aware reranker as planned extensions in Section 4.5.5.

4 Experimental Setup

4.1 Datasets

We use two complementary public datasets for codebase understanding: **DeepCodeBench** for repository-level Q&A and **LocBench** for issue-to-code localization (Qodo, 2025; Chen et al., 2025b). Together, they evaluate two stages of code intelligence: locating the relevant code context and producing the final answer.

DeepCodeBench (Q&A). DeepCodeBench contains 1,144 examples for evaluating final answer quality (Qodo, 2025). It also supports analysis by **scope**, **type**, and **searchability**, capturing whether questions require local or repository-wide reasoning, target core or peripheral functionality, and can be answered through explicit identifiers or deeper inference. We evaluate on the official test split.

LocBench (localization). LocBench contains 560 examples for evaluating whether a system can localize the correct file or function from an issue description (Chen et al., 2025b). We use the complete dataset, which includes 242 bug reports, 150 feature requests, 139 performance issues, and 29 security vulnerabilities.

Intent categorization. We categorize samples by developer intent: **leading** questions support planning before a change or failure, **lagging** questions support debugging and root-cause analysis after a failure, and **exploratory** questions support general code understanding.
As shown in Figure 2, DeepCodeBench is primarily exploratory, while LocBench provides stronger coverage of leading and lagging workflows. This gives broader coverage of realistic codebase-understanding scenarios across the two datasets.

Data splits. We evaluate on the official DeepCodeBench test split and the complete LocBench dataset. Since we do not train supervised models on these benchmarks, no new train/validation/test splits are created. All baselines and method variants use the same fixed evaluation data.

Future dataset extension. As future work, we plan to add localization targets to DeepCodeBench by mapping each question to the files, functions, or code regions needed to answer it. This would allow localization and answer-generation evaluation within the same benchmark.

4.2 Evaluation Methods

We use fixed datasets and metrics across all methods for fair comparison. Q&A is evaluated on the official DeepCodeBench test split (Qodo, 2025), while localization is evaluated on the complete LocBench dataset (Chen et al., 2025b). Since codebase understanding requires both answer generation and code navigation, we use three complementary metrics: fact-level coverage, answer-level quality, and localization accuracy.

1. **Fact Recall (Q&A).** Fact Recall measures whether the generated answer covers the key reference facts. Given a set of gold facts, an LLM judge checks whether each fact is present and correct in the generated answer:

$$\text{Fact Recall} = \frac{\#\{\text{facts judged present}\}}{\#\{\text{total gold facts}\}} \quad (1)$$

   This metric focuses on factual coverage rather than surface-level similarity. It follows nugget-style factual evaluation used in prior QA settings (Voorhees, 2004) and aligns with the evidence-based evaluation design of DeepCodeBench (Qodo, 2025).

2. **Quality Evaluation (Q&A).** We also evaluate overall answer quality using an LLM-as-a-judge setup inspired by the CoReQA evaluator (Chen et al., 2025a). Each generated answer is compared against the reference answer and scored on a 1–10 scale across four dimensions: **accuracy**, which measures factual correctness; **completeness**, which measures coverage of important answer aspects; **relevance**, which measures whether the answer directly addresses the question; and **clarity**, which measures whether the response

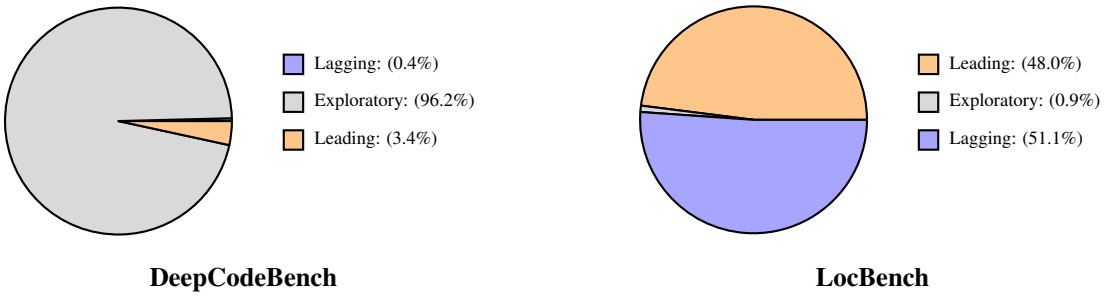


Figure 2: Intent distribution across DeepCodeBench and LocBench based on our categorization.

is organized and easy to follow. Following CoReQA's repeated-scoring setup, we run the quality evaluator twice for each answer and use the average score as the final quality score (Chen et al., 2025a).

3. **Acc@k (Localization).** For localization, we report Acc@k following LocAgent (Chen et al., 2025b). Each method outputs a ranked list of predicted code locations, and an example is correct if the gold location appears in the top-k predictions. We report Acc@k at both the **file level** and the **function level**, measuring whether the system retrieves the correct file and the specific relevant function or code region.

Together, these metrics separate factual coverage, overall answer quality, and code localization. This avoids collapsing distinct failure modes into a single score and ensures that comparisons reflect method behavior rather than changes in the evaluation protocol.

4.3 Baselines

We compare against baselines with increasing access to repository evidence. This progression helps separate the value of prior knowledge, problem-statement grounding, retrieved repository context, and structured graph-based exploration.

1. **Limited Context.** The Limited Context baseline receives only the repository name. It acts as a lower-bound control and tests prior-based answering without grounding in issue text or repository code.

2. **Full Context: Issue / PR.** The Full Context baseline receives the issue or PR title, body, and anchored code snippets, but no additional retrieved repository context. This tests how much can be answered from the problem statement alone.

3. **BM25.** BM25 is a sparse RAG baseline that retrieves top-k repository chunks using lexical matching. It tests whether keyword-based retrieval can provide enough grounding for answer generation.

4. **CodeRankEmbed.** CodeRankEmbed is a dense RAG baseline that retrieves top-k repository chunks using semantic similarity. It tests whether learned code embeddings improve retrieval over lexical matching. We use the available checkpoint and inference setup for this baseline.

5. **LocAgent.** LocAgent is a graph-guided localization baseline that performs structured exploration over the code graph to identify relevant files or functions from an issue description (Chen et al., 2025b). It is used for localization evaluation on LocBench.

4.4 Implementation Details

Our implementation uses public benchmarks, open-source retrieval tools, and both API-based and open-source language models to support reproducible evaluation across Q&A generation, retrieval, graph construction, and localization.

Off-the-shelf assets. We use DeepCodeBench for repository-level Q&A and LocBench for issue-to-code localization (Qodo, 2025; Chen et al., 2025b). For task models, we use GPT, Qwen3-8B, Qwen3-32B, and Llama-3.1-8B-Instruct. For judge-based evaluation, we use Qwen3-32B in thinking mode. For retrieval, BM25 is implemented directly as the sparse retriever, while CodeRankEmbed is used as the dense retrieval baseline. LocAgent is used as the graph-guided localization baseline, using available public checkpoints, code, or released implementation details where applicable (Suresh et al., 2024; Chen et al., 2025b).

Software stack. The implementation is written in Python. We use PyTorch, Hugging Face `transformers`, and `datasets` for model inference and dataset handling. For retrieval and indexing, we use `bm25s`, `rank_bm25`, and `faiss-cpu`. For orchestration, we use the OpenAI API and `llama-index`. For graph-based methods, we construct AST-based, dependency-based, structural, and knowledge-oriented graph views of the repository.

Computational resources. BM25 retrieval, indexing, and graph construction run on CPU. Open-source model inference for Q&A, localization, and judge-based evaluation requires GPU access, with Qwen3-32B requiring larger GPU resources than Qwen3-8B or Llama-3.1-8B-Instruct. GPT-based runs are executed through API calls.

Setup and reproducibility. We evaluate answer-generation baselines on the DeepCodeBench test split and localization baselines on the complete LocBench dataset (Qodo, 2025; Chen et al., 2025b). To ensure fair comparison, all applicable methods use fixed prompts, repository snapshots, chunking, retrieval settings, top-k values, and context-budget limits.

4.5 Initial Results and Analysis

Our initial results and analysis evaluate the performance of our pipeline in retrieving factual evidence, generating high-quality responses, and accurately localizing code regions within large repositories. We investigate whether graph-augmented representations can effectively bridge the gap between limited-context language models and the theoretical upper bound of full-context understanding. Our evaluation focuses on three core dimensions: fact recall on DeepCodeBench, multi-dimensional answer quality, and navigation accuracy on LocBench. The results demonstrate that BM25 remains a strong baseline, while graph-guided methods provide clear gains for top-ranked file localization and broad, cross-file queries.

4.5.1 Experimental Setup

To ensure a rigorous evaluation, we utilize two primary benchmarks: **DeepCodeBench** for repository-level question answering and **LocBench** for precise code region localization.

- **Models:** We evaluate a range of model scales, including **Qwen3 (8B and 32B)**, **Llama-3.1(8B)**, and **GPT-5.3-codex**, to observe how retrieval benefits scale with model capacity.
- **Baselines:** Performance is compared against **Limited Context** (providing an LLM only the repository name), and **Full Context** (the theoretical upper bound where the entire codebase is provided).
- **Metrics:** We employ **Fact Recall(%)** to measure information retrieval efficiency, **LLM-as-a-judge scores** (on a scale of 1-10) for Accuracy, Completeness, and Clarity, and **Acc@k** for localization precision.

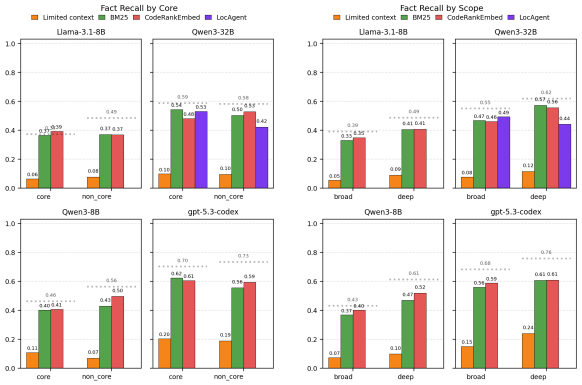
	Limited		BM25		CRE		HindSight v1		Full-PR	
Model	**Mic.**	**Mac.**	**Mic.**	**Mac.**	**Mic.**	**Mac.**	**Mic.**	**Mac.**	**Mic.**	**Mac.**
Llama-3.1-8B	.0679	.0722	.3639	.3689	.3751	.3787	n/a	n/a	.4107	.4388
Qwen3-8B	.0831	.0856	.3942	.4181	.3433	.4597	n/a	n/a	.4819	.5215
Qwen3-32B	.0903	.0969	.5049	.5203	.5023	.5094	.4820	n/a	.5597	.5848
GPT-5.3 codex	.1773	.1948	.5946	.5838	.6157	.5985	n/a	n/a	.7047	.7202

Table 1: Micro (Mic.) and Macro (Mac.) Fact Recall. **CRE**: CodeRankEmbed. HindSight v1 evaluated on Qwen3-32B only; n/a indicates not yet run.

4.5.2 Factual Recall Analysis

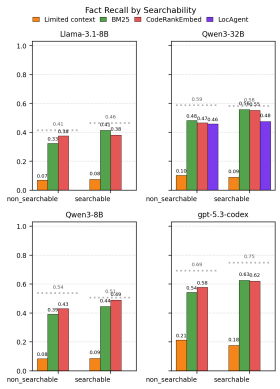
The first stage of our evaluation measures the system's ability to identify relevant code facts. Any form of retrieval offers a transformative improvement over the Limited Context baseline (0.195 for Qwen3-32B), confirming that retrieval is essential for multi-file repositories. We focus our analysis on Qwen3-32B to assess how different retrieval strategies perform with a competitive open-source backbone.

- **Overall retrieval performance:** As shown in fig. 3d, BM25 achieves the highest overall fact recall (0.520), followed by CodeRankEmbed (0.509) and HindSight v1 (0.482). HindSight v1 remains competitive with strong lexical and dense baselines despite using an open-source model for query decomposition.
- **Broad vs. deep questions:** HindSight v1 leads on broad-scope questions (0.493 vs. BM25 0.467 and CodeRankEmbed 0.462), supporting our hypothesis that graph-based retrieval helps when evidence is distributed across modules. On deep-scope questions, BM25 and CodeRankEmbed remain stronger because the required evidence is localized.
- **Core, non-core, and searchability:** HindSight v1 is competitive on core functionality

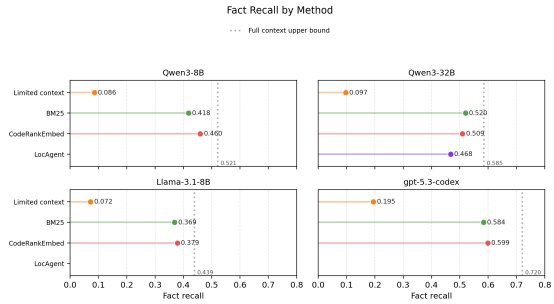


(a) By Core

(b) By Scope



(c) By Search



(d) Overall

Figure 3: Fact Recall breakdown across four dimensions. Detailed numerical results are provided in table 1.

but trails on non-core components, suggesting graph traversal helps most when queried entities are well-connected. It also remains close to the baselines on non-searchable questions, indicating that structural signals partially compensate when lexical cues are weak.

Overall, BM25 is the strongest aggregate baseline, but HindSight v1 shows a clear advantage on broad, cross-file questions where graph-guided retrieval adds structural context.

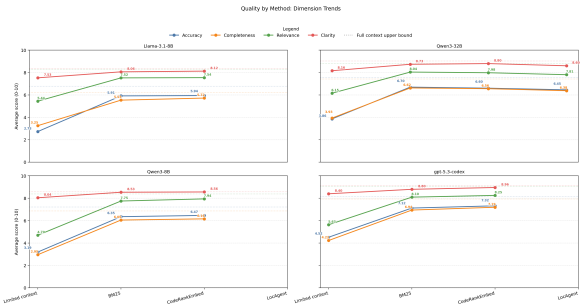


Figure 4: Quality dimension trends across methods for all evaluated models. While Clarity remains consistently high, Accuracy and Completeness show significant gains with improved retrieval.

4.5.3 Answer Quality Evaluation

Beyond raw recall, we evaluate answer utility across four dimensions: Accuracy, Completeness, Relevance, and Clarity. The results are summarized in table 2, and the trends are visualized in fig. 4.

- **Retrieval drives factual quality.** For Qwen3-32B, Accuracy rises from 3.860 (Limited) to 6.696 (BM25) and Completeness from 3.931 to 6.623, confirming that retrieved context is critical for grounded answers.
- **BM25 and CodeRankEmbed are closely matched.** The two retrievers perform similarly across dimensions. For GPT-5.3 Codex, CodeRankEmbed leads with 7.321 Accuracy and 7.194 Completeness; for Qwen3-32B, BM25 is marginally stronger on factual dimensions and CodeRankEmbed on Clarity.
- **HindSight v1 trails slightly on answer quality.** For Qwen3-32B, HindSight v1 reaches 6.455 Accuracy, 6.377 Completeness, 7.805 Relevance, and 8.602 Clarity — close to but consistently below BM25 and CodeRankEmbed. This suggests graph-based expansion retrieves useful context but needs better reranking before generation.
- **Clarity is least sensitive to retrieval.** For Qwen3-32B, Clarity rises only from 8.160 to 8.735 with BM25, while Accuracy and Completeness improve much more sharply, indicating that models produce well-written answers even when factual grounding is weak.
- **Full-PR remains the upper bound.** The gap between retrieval methods and Full-PR (e.g., 8.142 Accuracy for GPT-5.3 Codex) shows

Metric	Model	Limited	BM25	CRE	Hindsight v1	Full-PR
Accuracy	Llama-3.1-8B	2.728 ± .445	5.914	5.942 ± .460	N/A	6.778 ± .466
	Qwen3-8B	3.194 ± .323	6.353	6.466 ± .451	N/A	7.218 ± .570
	Qwen3-32B	3.860 ± .497	6.696	6.601 ± .399	6.455 ± .471	7.556 ± .457
	GPT-5.3 codex	4.517 ± .280	7.123	7.321 ± .381	N/A	8.142 ± .415
Completeness	Llama-3.1-8B	3.254 ± .567	5.535 ± .610	5.718 ± .594	N/A	6.231 ± .613
	Qwen3-8B	2.955 ± .290	6.052 ± .500	6.157 ± .479	N/A	6.862 ± .683
	Qwen3-32B	3.931 ± .476	6.623 ± .485	6.556 ± .488	6.377 ± .508	7.420 ± .485
	GPT-5.3 codex	4.226 ± .320	6.944 ± .536	7.194 ± .476	N/A	7.929 ± .460
Relevance	Llama-3.1-8B	5.442 ± .826	7.523 ± .494	7.543 ± .536	N/A	8.295 ± .448
	Qwen3-8B	4.696 ± .424	7.750 ± .427	7.940 ± .378	N/A	8.379 ± .451
	Qwen3-32B	6.153 ± .674	8.039 ± .402	7.981 ± .442	7.805 ± .416	8.791 ± .320
	GPT-5.3 codex	5.634 ± .244	8.097 ± .332	8.252 ± .271	N/A	9.041 ± .198
Clarity	Llama-3.1-8B	7.526 ± .549	8.060 ± .488	8.123 ± .387	N/A	8.364 ± .393
	Qwen3-8B	8.041 ± .911	8.532 ± .448	8.558 ± .393	N/A	8.573 ± .506
	Qwen3-32B	8.160 ± .524	8.735 ± .387	8.797 ± .390	8.602 ± .478	9.030 ± .317
	GPT-5.3 codex	8.397 ± 1.024	8.795 ± .454	8.957 ± .354	N/A	9.149 ± .351

Table 2: Answer quality comparison on DeepCodeBench. Metrics are grouped as primary rows to highlight performance differences between models across retrieval methods (Limited, BM25, CRE, Hindsight v1, and Full-PR). Hindsight v1 results are currently available only for Qwen3-32B.

Benchmark	Method	Model	File-level Acc			Module-level Acc		Function-level Acc	
			@1	@3	@5	@5	@10	@5	@10
LocBench	BM25-RAG	Llama-3.1-8B	.193	.173	.176	.062	.062	.057	.057
		Qwen-2.5-7B	.297	.290	.290	.090	.090	.087	.087
		Qwen-3-32B	.601	.584	.584	.273	.273	.265	.265
	HindSight v1	Llama-3.1-8B	.357	.333	.370	.069	.069	.037	.042
		Qwen-3-7B	.395	.435	.448	.211	.221	.120	.124
		Qwen-3-32B	.497	.556	.569	.390	.406	.301	.315
		GPT-5.3-Codex	**.798**	**.827**	**.864**	**.586**	**.655**	**.457**	**.532**
DeepCodeBench	BM25-RAG	Qwen-3-32B	.454	.533	**.582**	**.436**	**.476**	**.361**	**.410**
	LocAgent (adapted)	Qwen-3-32B	.423	.463	.467	.291	.308	.234	.251
	HindSight v1	Qwen-3-32B	**.535**	**.575**	.589	.363	.394	.283	.296

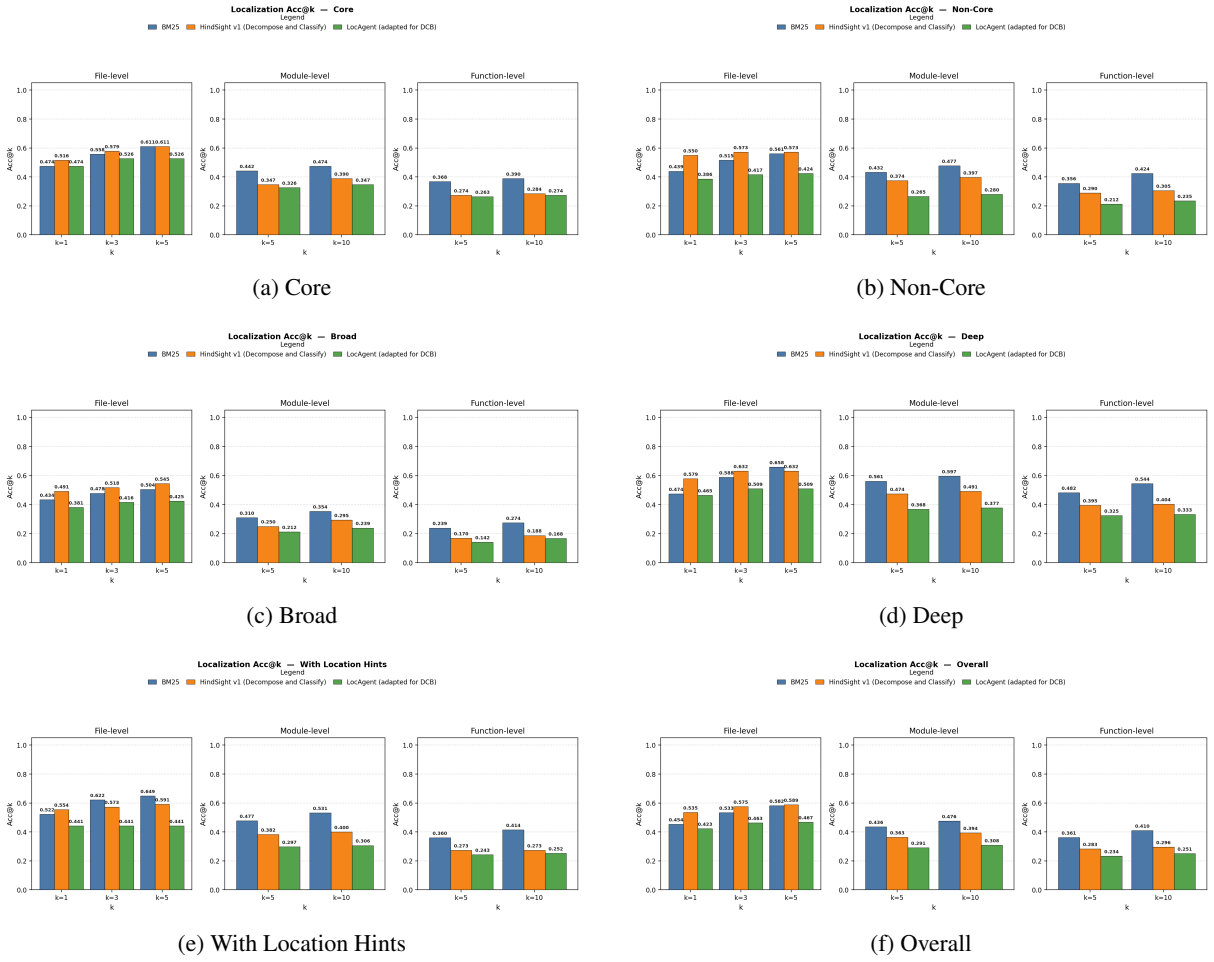
Table 3: Localization performance (Acc@k) across benchmarks and model scales. HindSight v1 outperforms BM25 on LocBench across granularities and achieves the best DeepCodeBench file-level Acc@1 with Qwen3-32B. BM25 remains stronger at module and function levels, motivating future graph-augmented reranking.

that context coverage is still the key bottleneck.

4.5.4 Code Localization Performance

To evaluate the effectiveness of our graph-based retrieval approach, we benchmark HindSight v1 (Decompose and Classify) against two representative baselines: BM25, a standard lexical search method, and LocAgent (adapted for DeepCodeBench), an agentic framework designed for codebase navigation. All three methods are evaluated on the DeepCodeBench localization subset using Qwen3-32B as the backbone, allowing a fair comparison of retrieval strategies under an identical open-source model. The aggregate results are shown in fig. 5f, and breakdowns by scope, core/non-core, location hints, and intent are shown in figs. 5a to 5e and 6.

- **Structural retrieval improves top-1 file localization.** HindSight v1 achieves the strongest file-level Acc@1 (**0.535**), outperforming BM25 (0.454) by **+8.1 points** and adapted LocAgent (0.423) by **+11.2 points** (fig. 5f). The gap narrows at Acc@5 (0.589 vs. 0.582), indicating that BM25 retrieves the correct files but ranks them less precisely.
- **Broad, cross-file queries benefit most from graph traversal.** On broad questions, HindSight v1 outperforms BM25 at every file-level k (Acc@1: 0.491 vs. 0.434; Acc@5: 0.545 vs. 0.504). On deep questions, HindSight v1 still leads at file Acc@1 (0.579 vs. 0.474), but BM25 wins at finer granularity.
- **HindSight v1 is robust without lexical an-**



(a) Core (b) Non-Core

(c) Broad (d) Deep

(e) With Location Hints (f) Overall

Figure 5: DeepCodeBench localization Acc@k breakdown across question categories. Results are shown at file, module, and function levels for BM25, HindSight v1, and LocAgent.

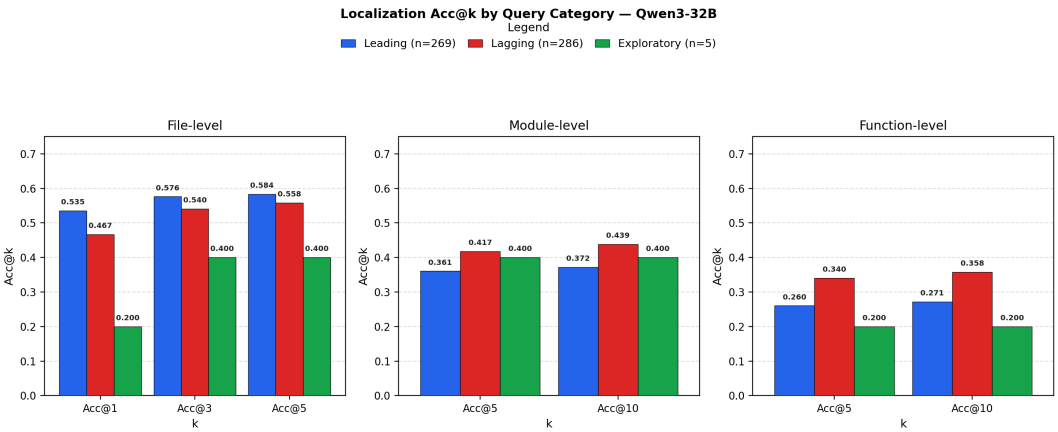


Figure 6: Localization Acc@k by query intent on DeepCodeBench using Qwen3-32B. Results are grouped into leading, lagging, and exploratory query categories and reported at file, module, and function levels.

chors. HindSight v1's largest advantage emerges when queries lack location hints: file-level Acc@1 of **0.517** vs. BM25's 0.388, a +12.9-point gain (fig. 5e). Query decomposition recovers latent retrieval anchors that lexical search misses. With hints, BM25 stays competitive since explicit identifiers favor keyword retrieval.

- **Graph-guided retrieval helps on non-core paths.** On non-core questions, HindSight v1 reaches file Acc@1 of **0.550** vs. BM25's

0.439 (+11.1 points). Non-core functionality is spread across utilities, tests, and peripheral modules — terrain where lexical retrieval alone struggles. On core questions, the two methods are closer.

- **BM25 remains stronger at finer granularity.** BM25 retains an advantage at module Acc@5 (0.436 vs. 0.363) and function Acc@5 (0.361 vs. 0.283), indicating that HindSight v1 surfaces the right file but does not yet rank entities within it as precisely.

- **Intent shapes the error pattern.** Leading queries achieve higher file-level accuracy than lagging queries (Acc@1: 0.535 vs. 0.467), since planning-oriented queries reference features directly. Lagging queries instead win at finer granularity (module Acc@5: 0.417 vs. 0.361; function Acc@5: 0.340 vs. 0.260), since bug reports describe symptoms that map onto specific methods.

- **Structural retrieval improves top-1 file localization.** HindSight v1 achieves the strongest file-level Acc@1 (**0.535**), outperforming BM25 (0.454) by **+8.1 points** and adapted LocAgent (0.423) by **+11.2 points** (fig. 5f). The gap narrows at Acc@5 (0.589 vs. 0.582), indicating that BM25 retrieves the correct files but ranks them less precisely.

- **Broad, cross-file queries benefit most from graph traversal.** On broad questions, HindSight v1 outperforms BM25 at every file-level k (Acc@1: 0.491 vs. 0.434; Acc@5: 0.545 vs. 0.504). On deep questions, HindSight v1 still leads at file Acc@1 (0.579 vs. 0.474), but BM25 wins at finer granularity.

- **HindSight v1 is robust without lexical anchors.** HindSight v1's largest advantage emerges when queries lack location hints: file-level Acc@1 of **0.517** vs. BM25's 0.388, a +12.9-point gain (fig. 5e). Query decomposition recovers latent retrieval anchors that lexical search misses. With hints, BM25 stays competitive since explicit identifiers favor keyword retrieval.

- **Graph-guided retrieval helps on non-core paths.** On non-core questions, HindSight v1 reaches file Acc@1 of **0.550** vs. BM25's

0.439 (+11.1 points). Non-core functionality is spread across utilities, tests, and peripheral modules — terrain where lexical retrieval alone struggles. On core questions, the two methods are closer.

- **BM25 remains stronger at finer granularity.** BM25 retains an advantage at module Acc@5 (0.436 vs. 0.363) and function Acc@5 (0.361 vs. 0.283), indicating that HindSight v1 surfaces the right file but does not yet rank entities within it as precisely.

- **Intent shapes the error pattern.** Leading queries achieve higher file-level accuracy than lagging queries (Acc@1: 0.535 vs. 0.467), since planning-oriented queries reference features directly. Lagging queries instead win at finer granularity (module Acc@5: 0.417 vs. 0.361; function Acc@5: 0.340 vs. 0.260), since bug reports describe symptoms that map onto specific methods.

4.5.5 Discussion and Preliminary Findings

Our midpoint results sharpen rather than overturn the hypothesis that structural knowledge is a force multiplier for repository-level RAG. Taken together, the fact-recall, quality, and localization experiments tell a consistent story: retrieval matters enormously, but *which* retriever wins depends on the character of the query.

The context gap is a completeness problem. Every retrieval method recovers most of the performance lost relative to Limited Context, but a substantial gap to Full Context remains. The LLM-judge dimensions localize this gap: Clarity and Relevance are near-saturated even without retrieval, while Accuracy and Completeness are the dimensions retrieval actually moves. Closing the gap is a problem of retrieval completeness, not of generation style.

Lexical retrieval is a stronger baseline than expected. BM25 and CodeRankEmbed produce nearly indistinguishable profiles, and on slices with strong surface signal (deep, core, searchable) both match or beat our agentic approach. A well-tuned sparse retriever captures most of what dense embeddings offer for repository QA, and agentic exploration is not automatically repaid by better aggregate numbers.

Agentic decomposition pays off where lexical anchors are weakest. HindSight v1's wins concentrate exactly where its design predicts: broad multi-file questions, non-searchable queries, and top-1 file localization. The non-searchable slice is most compelling — BM25 degrades sharply when keyword cues disappear while HindSight v1 preserves performance — showing that decompose-and-classify does work lexical retrieval cannot.

Granularity is a ranking problem, and intent shapes the errors. The same approach that wins at file Acc@1 loses at module and function Acc@k: HindSight v1 surfaces the right file but orders entities within it less precisely than BM25's term-level scoring. The leading/lagging crossover (planning queries are easier to localize at file level, debugging queries at function level) sharpens this into a ranking problem rather than a structural retrieval problem.

Future work. These findings shape the next phase. We plan to (i) add a test-linkage index alongside the existing AST and RepoGraph indexes (Xiao et al., 2024) to close the fact-recall gap on non-core and broad queries; (ii) introduce an intent-aware reranker adapted from Code Graph Model (Tao et al., 2025), conditioned on the query's intent label and decomposed subquestions, to close the module- and function-level gap without sacrificing the file Acc@1 advantage; and (iii) implement a per-intent traversal policy — file-level for leading queries, function-level for lagging queries — and refine the exploration policy for non-searchable and exploratory queries that remain furthest from the upper bound.

Acknowledgments

We thank our course staff and project mentors for their guidance and feedback. We also acknowledge the use of the Unity cluster for computational resources supporting our experiments.

References

Anonymous. 2024a. Locbench: Benchmarking code localization for repository-level tasks. *Computing Research Repository*. ArXiv preprint.

Anonymous. 2024b. Spidercodeqa: A benchmark for repository-level code question answering. *ACL Student Research Workshop*. ACL SRW.

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

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025b. Locagent: Graph-guided llm agents for code localization. *Computing Research Repository*, arXiv:2503.09089. ArXiv preprint.

Junjia Du, Yadi Liu, Hongcheng Guo, Jiawei Wang, Haojian Huang, Yunyi Ni, and Zhoujun Li. 2025. Dependeval: Benchmarking llms for repository dependency understanding. *Computing Research Repository*, arXiv:2503.06689. ArXiv preprint.

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? *Computing Research Repository*, arXiv:2310.06770. ArXiv preprint.

Hao Peng, Ge Li, Xin Xia, and Zhi Jin. 2021. Coderepoqa: A benchmark for repository-level code question answering. *Computing Research Repository*, arXiv:2109.12345.

Qodo. 2025. Deepcodebench: Real-world codebase understanding by q&a benchmarking. Blog post. Accessed 2026-03-03.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2024. Cornstack: High-quality contrastive data for better code retrieval and reranking. *arXiv preprint arXiv:2412.01007*.

Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. 2025. Code graph model (CGM): A graph-integrated large language model for repository-level software engineering tasks. *Preprint*, arXiv:2505.16901.

Ellen M. Voorhees. 2004. Overview of the TREC 2003 question answering track. In *Proceedings of the Twelfth Text REtrieval Conference (TREC 2003)*.

Jun Wang, Ming Zhou, and Yue Zhang. 2023. Coir: Code-oriented information retrieval for large-scale repositories. *Computing Research Repository*, arXiv:2306.00000.

Zilin Xiao, Wei Liu, Hanyang Guo, Quan Zhang, Guancheng Lin, Shijue Huang, Jieguang He, Zhengmao Li, and Weijiang Yu. 2024. RepoGraph: Enhancing AI software engineering with repository-level code graph. *Preprint*, arXiv:2410.14684.

Yun Zhang, David Lo, Xuan Wang, and Xiaoxing Ma. 2019. Codeqa: A question answering dataset for source code comprehension. In *Proceedings of the 2019 IEEE/ACM International Conference on Software Engineering*, pages 1–12. IEEE.

A Appendix

A.1 Prompt Templates

This appendix lists the prompt templates used for answer generation and evaluation. The answer-generation prompts differ only in the amount of context provided to the model: repository-only context, full issue/PR context, and retrieved repository context. The evaluation prompts are used for Fact Recall and quality evaluation.

A.1.1 Prompt for Answer Generation: Repository-Only

```
## Code Repository Question Answering

You are an AI assistant designed to answer
    questions about software repositories.
Given a repository name and a concise user
    question, produce a clear and concise
    answer.

Follow these steps:
1. Read the provided repository and question to
    understand the core problem or question.
2. Identify the minimal set of grounded facts
    needed to answer the question directly and
    accurately, including every part the
    question asks for, as far as the supplied
    text supports it.
3. Use only the supplied context: the
    repository name and the question. Do not
    rely on information about the codebase
    beyond that. Do not invent file paths,
    symbols, or behavior not grounded in what
    is supplied.
4. Answer directly; no broad background or
    unrelated classes.
5. Short code snippets of 2-6 lines are
    encouraged when the code itself is the
    clearest answer; avoid longer excerpts.
6. When the supplied context does not support
    any substantive answer, your entire reply
    must be exactly this one sentence and
    nothing else: "I am sorry, I could not find
    a solution for this question."
7. Privacy: do not include personal data,
    secrets, or credentials.
8. Do not add meta-commentary about the context
    itself.
9. Output format is strict: return only the
    answer text with no extra labels/headings,
    and no greetings, thanks, apologies except
    the exact fallback sentence above, or
    closing remarks.

## Question Context
### Repository Name
{repo}

### Question
{question}

### Answer:
```

A.1.2 Prompt for Answer Generation: Full Context

```
## Code Repository Question Answering

You are an AI assistant designed to answer
    questions about software repositories.
Use the provided code context blocks to produce
    a clear, concise, and useful answer.

Follow these steps:
1. Read the relevant code context to understand
    the core problem or question.
2. Identify the minimal set of grounded facts
    needed to answer the question directly and
    accurately, including every part the
    question asks for, as far as the supplied
    text supports it.
3. Use only the supplied context: the
    repository name, the question, and any code
    excerpts in the user message. Do not rely
    on information about the codebase beyond
    that. Do not invent file paths, symbols, or
    behavior not grounded in what is supplied.
4. Cite repo paths and symbols exactly as in
    NODE headers/snippets. Each NODE header has
    the form "SymbolName (path/to/file.py)".
    Whenever your answer identifies where a
    specific function, class, or method is
    defined or located, you MUST include the
    file path from the NODE header
    parenthetical. When the question asks about
    conditions, circumstances, or qualifiers,
    include those specific details if they
    appear in any NODE excerpts. If multiple
    NODEs describe similar-looking symbols but
    only one matches the specific condition
    stated in the question, answer based on
    that matching NODE and ignore the others.
    No unrelated code or filler.
5. Short code snippets of 2-6 lines are
    encouraged when the code itself is the
    clearest answer; avoid longer excerpts.
6. When the supplied context does not support
    any substantive answer, your entire reply
    must be exactly this one sentence and
    nothing else: "I am sorry, I could not find
    a solution for this question."
7. Privacy: do not include personal data,
    secrets, or credentials.
8. Do not add meta-commentary about the context
    itself.
9. Output format is strict: return only the
    answer text with no extra labels/headings,
    and no greetings, thanks, apologies except
    the exact fallback sentence above, or
    closing remarks.

## Code Context
### Repository Name
{repo}

### Relevant Code Context
{code_context}

### Question to be answered
{question}

### Answer:
```

A.1.3 Prompt for Answer Generation: Retrieved Context / RAG

```
## Code Repository Question Answering

You are an AI assistant designed to answer
    questions about software repositories.
Given a repository name, retrieved code
    context, and a user question, generate a
    concise answer.

Follow these steps:
1. Read the retrieved code context to
    understand the core problem or question.
2. Identify the minimal set of grounded facts
    needed to answer the question directly and
    accurately, including every part the
    question asks for, as far as the supplied
    text supports it.
3. Use only the supplied context: the
    repository name, the question, and any code
    excerpts in the user message. Do not rely
    on information about the codebase beyond
    that. Do not invent file paths, symbols, or
    behavior not grounded in what is supplied.
4. Cite repo paths and symbols exactly as in
    NODE headers/snippets. Each NODE header has
    the form "SymbolName (path/to/file.py)".
    Whenever your answer identifies where a
    specific function, class, or method is
    defined or located, you MUST include the
    file path from the NODE header
    parenthetical. When the question asks about
    conditions, circumstances, or qualifiers,
    include those specific details if they
    appear in any NODE excerpts. If multiple
    NODEs describe similar-looking symbols but
    only one matches the specific condition
    stated in the question, answer based on
    that matching NODE and ignore the others.
    No unrelated code or filler.
5. Short code snippets of 2-6 lines are
    encouraged when the code itself is the
    clearest answer; avoid longer excerpts.
6. When the supplied context does not support
    any substantive answer, your entire reply
    must be exactly this one sentence and
    nothing else: "I am sorry, I could not find
    a solution for this question."
7. Privacy: do not include personal data,
    secrets, or credentials.
8. Do not add meta-commentary about the context
    itself.
9. Output format is strict: return only the
    answer text with no extra labels/headings,
    and no greetings, thanks, apologies except
    the exact fallback sentence above, or
    closing remarks.

## Retrieved Code Context
### Repository Name
{repo}

### Relevant Information from Retrieval
{retrieved_code_context}

### Question to be answered
{question}

### Answer:
```

A.1.4 Prompt for Evaluation of Fact Recall

```
You are a precise fact-checking judge. Your
    sole task is to determine whether a given
    text mentions or supports a specific fact.
    Reply with only YES or NO - no explanation,
    no other text.

Does the following text mention or support this
    fact?

Text: {candidate_answer}

Fact: {fact}

Reply with only YES or NO.
```

A.1.5 Prompt for Quality Evaluation

```
You are an expert evaluator using
    chain-of-thought reasoning.

Question: {question}
Reference Answer: {reference_answer}
Generated Answer: {generated_answer}

Using the Reference Answer as the ground truth,
    evaluate the Generated Answer on the four
    dimensions below. You MUST follow this
    three-phase structure:

Phase 1 - Understand: Briefly summarize what
    the Reference Answer covers and what the
    Generated Answer claims.

Phase 2 - Reason: For each dimension, think
    step by step through the evidence before
    committing to a score. Write out your
    reasoning explicitly for each one.

Phase 3 - Score: After completing Phase 2,
    output ONLY the following JSON block with
    no other text after it:
{
  "accuracy_score": <int 1-10>,
  "completeness_score": <int 1-10>,
  "relevance_score": <int 1-10>,
  "clarity_score": <int 1-10>
}

Evaluation Dimensions:

1. Accuracy
Objective: Assess the factual correctness of
    the Generated Answer in comparison with the
    Reference Answer.
Evaluation Criteria: Verify the factual
    correctness of the Generated Answer by
    comparing it with the Reference Answer.
    Check the accuracy of any quoted sources.
Scoring Rules:
  - 1-2: The Generated Answer is mostly
    incorrect.
  - 3-4: The Generated Answer contains
    significant factual errors.
  - 5-6: The Generated Answer has some factual
    errors but is primarily accurate.
```

```
  - 7-8: The Generated Answer has minor
    inaccuracies but is overall correct.
  - 9-10: The Generated Answer is factually
    correct and has no errors.

2. Completeness
Objective: Evaluate whether the Generated
    Answer covers all aspects of the question.
Evaluation Criteria: Understand the key
    components of the Reference Answer and
    identify any critical points absent in the
    Generated Answer.
Scoring Rules:
  - 1-2: The Generated Answer covers almost
    none of the key aspects.
  - 3-4: The Generated Answer covers few key
    aspects with significant gaps.
  - 5-6: The Generated Answer covers some
    aspects but is missing important points.
  - 7-8: The Generated Answer covers most
    aspects with only minor omissions.
  - 9-10: The Generated Answer comprehensively
    covers all aspects of the question.

3. Relevance
Objective: Assess whether the Generated Answer
    addresses the core concern of the question.
Evaluation Criteria: Understand the core
    concern of the question, determine whether
    the Generated Answer directly addresses the
    question, and ensure the Generated Answer
    stays on-topic.
Scoring Rules:
  - 1-2: The Generated Answer is entirely
    off-topic or does not address the question.
  - 3-4: The Generated Answer mostly misses the
    core concern of the question.
  - 5-6: The Generated Answer partially
    addresses the core concern with notable
    digressions.
  - 7-8: The Generated Answer mostly addresses
    the core concern with minor off-topic
    content.
  - 9-10: The Generated Answer directly and
    fully addresses the core concern of the
    question.

4. Clarity
Objective: Determine whether the Generated
    Answer is easily understandable.
Evaluation Criteria: Evaluate the logical
    clarity and simplicity of the Generated
    Answer, ensuring it is easy to comprehend.
Scoring Rules:
  - 1-2: The Generated Answer is very difficult
    to understand.
  - 3-4: The Generated Answer is mostly unclear
    or poorly structured.
  - 5-6: The Generated Answer is somewhat clear
    but has notable clarity issues.
  - 7-8: The Generated Answer is mostly clear
    and easy to follow with minor issues.
  - 9-10: The Generated Answer is perfectly
    clear, logical, and easy to comprehend.

Now begin with Phase 1.
```