SpecAgent: A Speculative Retrieval and Forecasting Agent for Code Completion

George Ma[1*], Anurag Koul[2], Qi Chen[2], Yawen Wu[2], Sachit Kuhar[2], Yu Yu[2]
Aritra Sengupta[2†], Varun Kumar[2], Murali Krishna Ramanathan[2]
[1]UC Berkeley [2]Amazon Web Services

Abstract

Large Language Models (LLMs) excel at code-related tasks but often struggle in realistic software repositories, where project-specific APIs and cross-file dependencies are crucial. Retrieval-augmented methods mitigate this by injecting repository context at inference time. The low inference-time latency budget affects either retrieval quality or the added latency adversely impacts user experience. We address this limitation with *SpecAgent*, an agent that improves both latency and code-generation quality by proactively exploring repository files during indexing and constructing *speculative context* that anticipates future edits in each file. This indexing-time asynchrony allows thorough context computation, masking latency, and the speculative nature of the context improves code-generation quality. Additionally, we identify the problem of *future context leakage* in existing benchmarks, which can inflate reported performance. To address this, we construct a synthetic, leakage-free benchmark that enables a more realistic evaluation of our agent against baselines. Experiments show that SpecAgent consistently achieves absolute gains of 9–11% (48–58% relative) compared to the best-performing baselines, while significantly reducing inference latency.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in a wide range of software engineering tasks, including code completion (Izadi et al., 2024; Yang et al., 2025), code editing (Gupta et al., 2023), issue resolution (Jimenez et al., 2024), and automated test generation (Deng et al., 2023). These successes are largely the result of advances in large-scale model pre-training and fine-tuning techniques (Wang et al., 2021; Feng et al., 2020), fueled by massive, publicly available code-corpus datasets (Kocetkov et al., 2023). This has led to significant improvement over benchmarks (Chen et al., 2021; Austin et al., 2021; Ding et al., 2023) that test general programming knowledge , logical reasoning, and problem-solving skills.

However, the majority of these benchmarks evaluate models in isolated settings that do not reflect the complexity of real-world software development (Jimenez et al., 2024). In practice, software engineers often work within large, evolving, and context-rich codebases where understanding the local context, e.g., private dependencies, is essential to completing tasks correctly. Without access to such context, such as the definitions of project-specific APIs, cross-file dependencies, or coding conventions, LLMs can produce completions or edits that are syntactically correct but semantically inconsistent with the target repository. As a result, bridging the gap between the general capabilities of LLMs and the repository-specific requirements of real-world engineering tasks remains a critical challenge (Liang et al., 2025a).

One promising approach to this challenge has been to enhance LLMs with information retrieval (IR) mechanisms that incorporate repository-specific knowledge during inference. This includes techniques like retrieving similar code snippets using BM25 (Robertson et al., 2009), mining semantically related code via dense embeddings (Zhang et al., 2024, 2023), and extracting structural metadata from the repository (Ouyang et al., 2025). Integrating such retrieved information into the model's prompt has been shown to significantly improve performance (Wu et al., 2024) on tasks that require awareness of cross-file dependencies, custom function signatures, or domain-specific patterns. While effective in improving correctness, these retrieval-based strategies require querying large indexes or scanning complex repository structures during on-

[*]Work done as an intern at Amazon Web Services.
[†]Correspondence to: george_ma@berkeley.edu, aritras@amazon.com.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 17266–17327
July 2-7, 2026 ©2026 Association for Computational Linguistics

line inference, which can introduce substantial latency.

This work is in a low-latency domain of in-IDE code auto-completions—inline completions, typically offered by products such as Cursor (Team, 2024), Copilot (Inc., 2024), Amazon Q (Web Services, 2024), etc. Our aim is to enhance inline completions user experience by improving model performance through richer context, as well as eliminating inference-time retrieval latency. To achieve this, we develop *SpecAgent*, an agent that constructs a structured context for each file at repository indexing time rather than at inference time. By front-loading these costly operations, we streamline the online phase for faster responses. We leverage a novel insight where the agent *speculates* on future functionalities or issues within files and retrieves additional context that supports building or fixing these anticipated changes in the upcoming developer session.

Our key contributions include the design of a speculative context construction framework that anticipates developer needs by pre-computing both current and likely future code-related information. This context is retrieved asynchronously at indexing, allows thorough context computation saving on inference time latency. Additionally, we also identify a critical limitation in existing code-completion benchmarks, which suffer from *"future-context leakage"* where target function invocations across files inadvertently reveal information about future code, such as function signature, intended usage, etc. To address this issue, we introduce a new benchmark, explicitly crafted to eliminate such leakage and provide a more rigorous evaluation environment. Finally, we develop an oracle agent to establish upper-bound performance metrics on this benchmark and demonstrate that leveraging SpecAgent's context improves the Qwen3-8B (Yang et al., 2025) model's performance by approximately **10–11%** (58% relative), and the Qwen3-30B-A3B (Yang et al., 2025) model by **9–10%** (48% relative) compared to strong baselines, with no additional inference-time retrieval latency.

2 Related Works

Code Language Models. Code language models underpin a wide range of software engineering tasks, including code completion (Izadi et al., 2024), debugging (Chen et al., 2023), translation (Dhruv and Dubey, 2025), and issue resolution (Jimenez et al., 2024). Progress in code completion has spanned data filtering, data synthesis, and both pre-training and post-training strategies. Notable developments include StarCoder-v2 (Lozhkov et al., 2024), which scales diversified high-quality code data, and OpenCoder (Huang et al., 2024), which leverages code-related web data with synthetic fine-tuning. Seed-Coder (Seed et al., 2025) introduces Long Chain-of-Thought (LongCoT) reinforcement learning to improve reasoning, while Qwen3-Coder (Team, 2025) extends model context to 256K tokens.

Repository-Level Code Completion. Repository level completion is more challenging than function- or file-level tasks, as it requires holistic reasoning over large codebases. Benchmarks such as RepoBench (Liu et al., 2024) and CrossCodeEval (Ding et al., 2023) evaluate model performance in this setting. Recent methods enhance repository context during inference: RepoFuse (Liang et al., 2024) integrates dependency and similarity signals; R2C2-Coder (Deng et al., 2024) fine-tunes models with diverse context types; and Zhang et al. (2025) study context integration and pruning strategies.

Efficient Context Retrieval for Inline Completion. Code language models face efficiency constraints from limited context windows and strict latency budgets that preclude full-repository input. Retrieval-augmented generation (RAG) mitigates this by dynamically fetching relevant snippets (Lewis et al., 2020), though it introduces significant retrieval overhead (Wang et al., 2025). To reduce this cost, CoSHC (Gu et al., 2022) applies deep hashing for compact similarity search, while SECRET (Gu et al., 2025b) uses segmented hashing to accelerate lookups. Complementary work (Gu et al., 2025a) distills smaller retrievers to reduce embedding latency. Our approach is orthogonal to these optimizations: we shift retrieval computation from inference to indexing time, reframing the cost–latency trade-off in retrieval-augmented code completion.

3 Preliminaries

3.1 Inline Function Completion

We study the problem of *inline code completion* (Izadi et al., 2024), specifically *function completion*, in which the goal is to predict the body of a target function within a partially observed source file. At inference time, a code completion model is

provided with: *1) Left context:* the content appearing before the target function in the source file. *2) Right context:* the content appearing after the target function in the source file. *3) Prompt:* the signature and docstring of the target function. *4) Cross-file contexts:* additional code snippets retrieved from other files in the repository. The code completion model generates a prediction for the target function body, which is then evaluated against unit tests (when available) (Zhuo et al., 2025) or through static metrics such as exact match or edit similarity (Paul et al., 2024) .

3.2 Baseline Retrieval Methods for Cross-File Contexts

Cross-file contexts can be obtained using various retrieval methods. We summarize three representative approaches below. These are state-of-the-art baselines in inline low latency code autocompletions to the best of our knowledge.

Sparse retrieval. A classical approach that ranks candidate code chunks from other files by token-level similarity to the query, often using methods such as BM25 (Robertson et al., 2009). The query is typically constructed from the target function's signature, docstring, and potentially parts of its surrounding context.

Dense retrieval. A learned approach that encodes both the query and candidate code chunks into vector representations using a pre-trained embedding model such as UniXcoder (Guo et al., 2022) or CodeSage (Zhang et al., 2024). The candidates are ranked by a similarity measure (e.g., cosine similarity) between their embeddings.

RepoMap. A structural retrieval method that leverages the API structure of the repository. RepoMap (Aider, 2023) extracts and indexes information from all files, such as class names, member functions, function signatures, and variable names. For a given target file, RepoMap retrieves related entities from imported files based on the repository's dependency graph.

3.3 Indexing-Time vs. Inference-Time Retrieval

Retrieval methods for code completion differ fundamentally in *when* the relevant computations can be performed.

- *Inference-time retrieval*: Sparse or dense retrieval methods rely on queries constructed from information available when the developer begins authoring a target function, such as its signature, docstring, a program statement inside the function, or surrounding file context. Since this information is unavailable in advance, similarity scores and rankings must be computed at inference time, immediately before invoking code-completion model. This introduces additional latency, which can significantly degrade the responsiveness of interactive code completion systems (up to 9 secs in our experiments). Moreover, as we discuss in Section 5.1, existing benchmarks often allow such methods to exploit future context leakage, leading to inflated and unrealistic evaluations.

- *Indexing-time retrieval*: In contrast, indexing-time retrieval is performed proactively during repository analysis, before the developer has written the target function. For example, RepoMap (Aider, 2023) builds a repository-wide API map that is fully computable ahead of time, without access to the inference-time query. In this paradigm, agents operate on an *indexing-time repository state*, a snapshot that predates the target function and often its associated callers and tests. Because indexing-time analysis is decoupled from user interaction, agents can conduct extensive exploration, static analysis, and even speculative reasoning without affecting user-perceived latency. The contexts associated with a target file are then determined by these pre-computed structures and relations.

The distinction between indexing-time retrieval and inference-time retrieval is central to our work. Indexing-time retrieval enables the system to shift costly exploration and context construction away from the latency-critical inference path. This not only improves efficiency, but also allows for richer and more global forms of analysis that would be impractical to execute at inference time. In Section 6, we show that such indexing-time speculative context construction leads to substantial improvements in model accuracy while introducing no inference-time overhead.

4 Methodology

This section describes our agent-based approach for constructing cross-file context to support inline

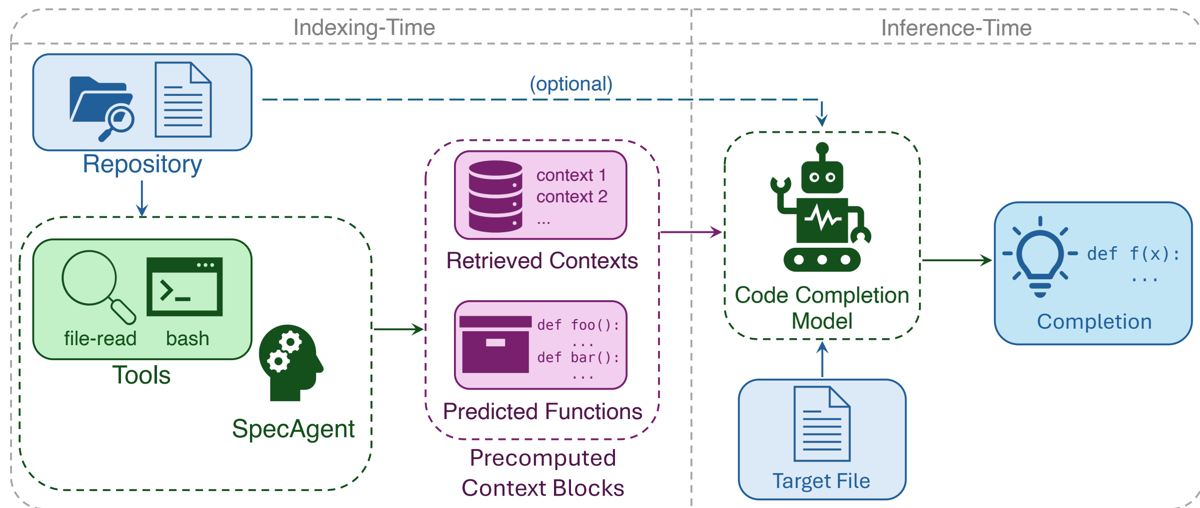


Figure 1: SpecAgent's workflow at indexing-time (left) involves retrieving relevant contexts and generating target function predictions asynchronously. These are then provided to the code completion model during inference (right).

code completion. The primary idea is to perform *indexing-time* (background) asynchronous exploration of a repository to produce structured context blocks that are later consumed by a code completion model at *inference time*. We formalize the indexing-time settings and describe the indexing-time agent family (including their interfaces, outputs, and operational constraints). We also propose an oracle agent that serves as a principled upper bound for context quality.

4.1 Indexing-Time Agents

We propose a family of agents that, at indexing time, proactively retrieve cross-file contexts and generate target function predictions to enhance code completion.

Indexing-time agent capabilities. The agent receives the path and content of the target file, along with full-repository access. It can read entire files or specific line ranges, perform keyword searches, and execute read-only shell commands. Prompted to analyze the target file and explore the repository, the agent returns a set of *context blocks*, each representing a specific type of potentially useful information, e.g., related code snippets, dependency structures, interface signatures, error-handling patterns, or speculative implementations. All retrieval is conducted solely on the indexing-time repository state, ensuring no leakage of the ground-truth target function.

In our experiments (Section 6.2), each agent produces 12 context blocks per target file. These blocks are later merged with local signals (left file context, right file context, and the prompt) before being passed to the completion model, introducing no additional inference-time latency compared to standard retrieval-based methods.

Agent variants. We evaluate three complementary variants, each adhering to a common interface and output format:

- *Retriever Agent*. It focuses exclusively on *retrieval*. Given the target file, it identifies likely dependencies and usage patterns, searches for relevant code (helpers, call patterns, test snippets), and returns ranked snippets and structural hints to guide the model at inference time. It generates 12 retrieval-based context blocks, due to context-size limitation of inline code-completion model.
- *Forecaster Agent*. It focuses exclusively on *prediction*. Without retrieving external snippets, it hypothesizes plausible functions a developer might add to the file, generating one or more candidate implementations with brief rationales. It then outputs 12 prediction-based context blocks. The Retriever Agent and Forecaster Agent differ only in their prompting, ensuring comparability across retrieval- and prediction-focused strategies.
- *Speculative Agent (SpecAgent)*. It combines retrieval and prediction into a single workflow. SpecAgent constructs a hybrid set of 12 context blocks by taking the top-ranked retrieval blocks from the Retriever Agent and the top-ranked prediction blocks from the Forecaster

Agent. This allows the composition of contexts to vary flexibly. In the main experiments (Section 6.2), we use 9 retrieval blocks and 3 prediction blocks, and we further study alternative compositions in ablation experiments (Section 6.3). By jointly selecting relevant contexts and synthesizing candidate functions, SpecAgent can directly supply accurate completions or high-quality drafts, while retaining the fallback benefits of retrieved evidence.

We summarize the indexing-time pipeline of SpecAgent in Algorithm 1. For clarity, let $\mathcal{C}(f)$ denote the ordered list of context blocks associated with a target file f, and let k denote the total number of blocks (fixed to 12 in our experiments).

Algorithm 1 SpecAgent Indexing-Time Pipeline

Require: Repository $\mathcal{R}$, target file f, block budget k, prediction budget k_p, retrieval budget k_r

Ensure: Ordered list of context blocks $\mathcal{C}(f)$ for file f

1: Initialize prompts P_{ret} (retrieval) and P_{pred} (prediction)

2: **Retriever Agent:**
3: Invoke agent with prompt P_{ret} and read-only tools over $\mathcal{R}$
4: Agent explores repository using arbitrary sequences of tool calls (e.g., search, file inspection)
5: $\mathcal{R}_f \leftarrow$ ordered list of up to k_r retrieval context blocks

6: **Forecaster Agent:**
7: Invoke agent with prompt P_{pred} and read-only tools over $\mathcal{R}$
8: Agent analyzes repository structure and synthesizes plausible future functions for f
9: $\mathcal{P}_f \leftarrow$ ordered list of up to k_p prediction context blocks

10: **Composition:**
11: $\mathcal{C}(f) \leftarrow$ concatenate $\mathcal{P}_f$ and $\mathcal{R}_f$ (preserving order)
12: **if** $|\mathcal{C}(f)| > k$ **then**
13: truncate $\mathcal{C}(f)$ to first k blocks
14: **end if**

15: Store $\mathcal{C}(f)$ in index for file f
16: **return** $\mathcal{C}(f)$

Integration with inference. Context blocks from indexing-time agents are stored and indexed per file. At inference time, the completion model receives: (i) left file context, (ii) right file context, (iii) the prompt, and (iv) the stored cross-file context blocks. Since all exploratory work is completed asynchronously before inference, this design supports richer, more diverse contexts without increasing inference latency. An example of the code completion model's prompt during inference is shown in Section E.

Rationale and benefits. Indexing-time exploration enables deep, repository-wide analysis under realistic operational constraints, amortizing computation across many future completions. The Retriever Agent supplies corroborating evidence that disambiguates interfaces; the Forecaster Agent provides fully formed drafts that may be directly adopted as target code; and SpecAgent combines both to maximize the probability of correct completion under limited attention budgets.

Re-indexing and deployment considerations. In a realistic IDE deployment, indexing-time context construction runs asynchronously and is not triggered by every keystroke or routine save. Instead, cached contexts are refreshed only when repository changes are large enough to materially alter the file's anticipated future functionality or the relevant cross-file contexts, such as substantial edits, addition or removal of major functionalities, or non-local dependency changes. This makes the preprocessing cost amortizable across many future completion requests. Although the one-time indexing cost per file is non-trivial, it is incurred infrequently and entirely off the latency-critical inference path, while the resulting contexts can be reused many times. In addition, indexing-time agents can be instantiated with lower-cost models while still preserving the main benefits of speculative context construction. Overall, this yields a favorable trade-off for inline code completion: moderate asynchronous preprocessing can improve completion quality without adding inference-time latency. Incremental and modular re-indexing remains an important direction for future work.

4.2 Oracle Agent

In code completion, many factors influence performance. Since our work focuses on cross-file context, we design an "oracle" retrieval agent to estimate the upper bound achievable by improv-

ing context quality alone. This agent operates in the full *inference-time state*, where both the target function and its callers are present. Unlike the *Retrieval Agent*, it also has access to the ground-truth implementation, enabling analysis of calls, dependencies, and error handling to select highly relevant cross-file contexts—though the model never sees the ground-truth directly. To ensure fairness, any block that copies or paraphrases the target function is filtered out. The oracle uses the same repository tools and formatting as the Retrieval agent, enabling direct comparison across strategies.

Why inference-time? The oracle upper bounds both indexing-time retrieval (like our agent variants) and inference-time methods (e.g., BM25, dense retrieval), which may surface usage patterns or tests of the target function. It thus simulates the best-case inference-time scenario for evaluating how close practical methods come to this ideal.

5 Benchmark

This section presents the benchmark setup used in our study. We begin by identifying a critical flaw in existing code completion benchmarks: *future context leakage*, which makes them incompatible with our experimental goals. We then describe the methodology for constructing a new benchmark specifically tailored for evaluating indexing-time context retrieval.

5.1 The Future Context Leakage Problem

Many existing code completion benchmarks (e.g., Liang et al. (2025b); Ding et al. (2023); Liu et al. (2024)) suffer from a critical flaw in their construction. Given a target function to be completed, these benchmarks typically remove its definition prior to context retrieval. However, this does not eliminate all forms of information leakage: other parts of the repository, such as test files or caller functions, may still reference or depend on the target function. In practice, we observe that retrieval methods can access these leaked code chunks, such as test cases, resulting in artificially inflated performance. This issue contradicts real-world development scenarios, where code that calls a target function would not exist prior to the function's implementation. We refer to this as the *future context leakage* problem, following the terminology introduced by Zheng et al. (2025).

We illustrate an example of this problem in Figure 2. As shown in Figure 2, the target function `save` is defined in `target_file.py` and invoked in `caller_file.py`. In typical benchmarks, only the definition of `save` is removed, while `caller_file.py` remains untouched. As a result, information about the target function can still be retrieved via its call sites, violating the assumption that the function does not yet exist during context retrieval.

Such leakage leads to misleading evaluations. For existing retrieval methods, including sparse retrieval based on lexical similarity and dense retrieval based on semantic similarity, this can result in retrieval of the very test cases used for evaluation, thereby inflating performance. In our indexing-time context retrieval setting, the problem becomes even more pronounced. Our retrieval agent is capable of analyzing these leaked call sites and accurately inferring the implementation of the target function, yielding unreasonably high performance that does not reflect realistic conditions.

To our knowledge, the only publicly available benchmark that explicitly addresses this issue is HumanEvo (Zheng et al., 2025). Unfortunately, we were unable to setup their containerization environment. As a result, we construct a synthetic benchmark specifically designed for indexing-time context retrieval without future context leakage. While building such a benchmark from real-world repositories is beyond the scope of this paper, we advocate for the development of more realistic code completion benchmarks in the research community.

5.2 Benchmark Construction

Given a target function and its associated inference-time repository, our goal is to synthesize a plausible state of the repository from an earlier point in time, before the target function has been implemented. This reflects a realistic scenario in which the user has not yet begun authoring the target function, and the retrieval agent operates asynchronously during indexing time. As illustrated in Figure 2, simply removing the function calls to `save` is insufficient. Residual information, such as import statements and surrounding logic (e.g., print statements), can still implicitly reveal the existence and semantics of the target function. In practice, such leakage can also persist in comments, docstrings, and the broader program structure. Therefore, simple rule-based filtering or static analysis alone is inadequate for eliminating all references to the target function. The desired indexing-time state is visualized in Figure 2.

target_file.py

```
import json

__all__ = ["compute", "save"]

def compute(x):
    return x * 2 + 1

def save(x):
    ...
```

caller_file.py

```
from target_file import compute, save

def run():
    y = compute(42)
    print(y)
    # saving to disk
    save(y)
    print("saved")
```

Inference-Sime State

target_file.py

```
import json

__all__ = ["compute", "save"]

def compute(x):
    return x * 2 + 1
```

caller_file.py

```
from target_file import compute

def run():
    y = compute(42)
    print(y)
    # saving to disk

    print("saved")
```

AST-Based Context Leakage Cleaning

target_file.py

```
import json

__all__ = ["compute"]

def compute(x):
    return x * 2 + 1
```

caller_file.py

```
from target_file import compute

def run():
    y = compute(42)
    print(y)
```

Agent-Based Context Cleaning

Figure 2: Illustration of future context leakage in existing benchmarks. Although the target function `save` is removed from `target_file.py`, relevant information remains in the target file and `caller_file.py`, allowing context retrieval methods to access information that would not exist in a real development scenario.

To construct a repository state that is both free of future context leakage and functionally intact, we introduce an automated agent, which we call the *function removal agent*. This agent is equipped with tools for executing shell commands, reading and writing files, and navigating the repository. It is guided by static analysis tools (Brunsfeld et al., 2019) that identify all call sites of the target function, and it is prompted to explore and edit the repository to produce a realistic indexing-time state: one in which not only the callers but also all semantically linked information has been removed. Crucially, the agent is expected to preserve the functional correctness of the remaining codebase.

All experiments involving our retrieval agent are conducted on the indexing-time state produced by the function removal agent. This ensures a fair evaluation setting where no future knowledge of the target function is leaked. A detailed description of the construction pipeline, along with our benchmark validation procedure, is provided in Section C. A real-world example of the synthetic indexing-time state constructed by the function removal agent is presented in Section D. This example demonstrates that simple AST-based cleaning is insufficient to eliminate future context leakage.

6 Experiments

6.1 Experimental Setup

We evaluate on the REPOCOD dataset (Liang et al., 2025b), a function completion benchmark with 980 problems drawn from 11 popular open-source projects, where over 58% of the problems require file- or repository-level context. For each problem, the completion model must generate the body of a target function given its signature and docstring, the left and right context, and optionally cross-file contexts.

We compare our indexing-time agents (including SpecAgent) and the oracle agent against the following baseline retrieval methods: (1) no cross-file context, (2) sparse retrieval (BM25), (3) RepoMap, (4) BM25 + RepoMap, and (5) dense retrieval using UniXcoder (Guo et al., 2022) and CodeSage v2 large (Zhang et al., 2024), the state-of-the-art low-latency retrievals for inline code completion.

A key distinction between baselines and our agents lies in the *timing of context construction*:

Baseline retrieval methods. Following the official REPOCOD setting, we remove the body of the target function at inference time, and use its signature and docstring as the query. Left and right file context as well as retrieved cross-file contexts are provided to the model. This setup enables comparability with prior work, but is known to suffer from *future context leakage*, since other files may indirectly reference the ground-truth completion (Section 5.1). As a result, baseline results likely *overestimate* real-world deployment performance.

Indexing-time agents (ours). Our agents operate on the synthetic indexing-time state of the repository (Section 5.2), before the target function has been written. They cannot rely on the ground-truth completion or future contexts, but instead must proactively predict or retrieve useful blocks based only on current repository contents. The resulting

Model	Method	Pass@1 (%)	Latency (s)	Pre-processing Time (s)
Qwen3-8B	None	16.22	4.03	-
Qwen3-8B	BM25	17.55	5.01	-
Qwen3-8B	RepoMap	16.73	4.06	74.45
Qwen3-8B	Dense (UniXcoder)	16.22	7.15	-
Qwen3-8B	Dense (CodeSage v2 large)	15.41	11.50	-
Qwen3-8B	BM25 + RepoMap	17.65	4.72	74.45
Qwen3-8B	Retriever Agent (Ours)	20.92	4.36	46.27
Qwen3-8B	Forecaster Agent (Ours)	22.35	4.55	49.08
Qwen3-8B	**SpecAgent (Ours)**	**27.86**	4.55	49.08
Qwen3-8B	*Oracle Agent*	*40.20*	4.53	45.19
Qwen3-30B-A3B	None	18.37	5.03	-
Qwen3-30B-A3B	BM25	18.88	6.03	-
Qwen3-30B-A3B	RepoMap	17.45	5.54	74.45
Qwen3-30B-A3B	Dense (UniXcoder)	18.78	8.49	-
Qwen3-30B-A3B	Dense (CodeSage v2 large)	16.94	14.84	-
Qwen3-30B-A3B	BM25 + RepoMap	18.16	5.88	74.45
Qwen3-30B-A3B	Retriever Agent (Ours)	23.27	5.75	46.27
Qwen3-30B-A3B	Forecaster Agent (Ours)	23.67	5.80	49.08
Qwen3-30B-A3B	**SpecAgent (Ours)**	**27.96**	5.89	49.08
Qwen3-30B-A3B	*Oracle Agent*	*40.10*	5.91	45.19

Table 1: Pass@1, inference-time latency, and indexing pre-processing time for all retrieval methods.

blocks are cached and later supplied to the completion model as cross-file context at inference. This setting avoids leakage and more accurately reflects realistic development scenarios, at the cost of being a harder task.

Oracle agent. The Oracle Agent is evaluated at inference time, with access to the ground-truth completion. It generates cross-file context blocks that are passed to the completion model, but cannot include the ground-truth completion itself. This represents an upper bound.

Implementation details. We evaluate with two code completion models: Qwen3-8B and Qwen3-30B-A3B (Yang et al., 2025). All agents use Claude 3.7 Sonnet (Anthropic, 2025) as the backbone. We also experiment with Qwen3-Coder as the agent in Section A. The left, right, and cross-file contexts are each capped at 10K tokens. We report pass@1 as the primary metric, along with inference-time latency and indexing pre-processing time (Table 1). We run experiments on a cluster with 8 A100 GPUs.

6.2 Main Results

Table 1 shows that SpecAgent achieves the highest pass@1 among retrieval-based methods, excluding the Oracle Agent upper bound. SpecAgent's context improves the Qwen3-8B (Yang et al., 2025) model's performance by approximately **10–11%** (58% relative), and the Qwen3-30B-A3B (Yang et al., 2025) model by **9–10%** (48% relative) compared to strong baselines. Importantly, SpecAgent maintains inference-time efficiency: unlike BM25 or dense retrievers, it does not require on-the-fly index lookups or similarity computations, instead incurring a one-time indexing cost of roughly 50 seconds (executed asynchronously). A comparison of contexts generated by different methods is shown in Section F.

6.3 Ablation Studies

Component ablation. We compare SpecAgent to its variants (Retriever Agent and Forecaster Agent, see Section 4.1). Removing either component lowers performance, highlighting the complementary roles of prediction and retrieval. Interestingly, the Forecaster Agent alone outperforms the Retriever Agent, underscoring the benefit of anticipating user intent.

Combined retrieval strategies. When SpecAgent's contexts are concatenated with those from BM25 or dense retrievers, performance decreases

(Table 2). This indicates that SpecAgent already selects high-quality contexts, and adding noisy blocks from other methods dilutes performance.

Method	Pass@1 (%)
SpecAgent	**27.86**
SpecAgent + BM25	25.10
SpecAgent + Dense (UniXcoder)	25.31

Table 2: Pass rates of combined retrieval strategies.

Inference-time ablation. For completeness, we also run SpecAgent under the same inference-time setup as baselines (target function removed, contexts retrieved at inference). This setting introduces leakage, enabling the agent to guess functionality from surrounding files. As shown in Table 3, SpecAgent achieves much higher pass@1 here, but we emphasize that these numbers are not realistic. Our main results adopt the stricter indexing-time async setting to better reflect real-world usage.

Model	Method	Pass@1 (%)
Qwen3-8B	SpecAgent (indexing)	27.86
Qwen3-8B	SpecAgent (inference)	**30.92**
Qwen3-30B-A3B	SpecAgent (indexing)	27.96
Qwen3-30B-A3B	SpecAgent (inference)	**34.29**

Table 3: Ablation on inference-time SpecAgent.

Composition of context blocks. We vary the ratio of prediction vs. retrieval blocks while fixing the total number. Performance peaks with three prediction blocks (Table 4), showing that both speculative predictions and retrieved contexts contribute to SpecAgent's success.

#Predictions	Pass@1 (%)
0	20.92
1	27.76
3	**27.86**
6	24.29
12	22.35

Table 4: Ablation on the composition of contexts.

7 Summary

We presented *SpecAgent*, a speculative context construction framework that shifts repository-specific retrieval from inference time to indexing time, enabling large language models to operate with richer, pre-computed context while maintaining interactive responsiveness. By anticipating likely future changes and pre-gathering relevant cross-file information, SpecAgent addresses both the latency bottleneck and the context insufficiency that limit retrieval-augmented methods in real-world repositories. To enable realistic evaluation, we introduced a benchmark free from future-context leakage, providing a fairer measure of code completion performance. Experiments on two strong LLMs show consistent absolute 9–11% accuracy gains over competitive baselines without additional inference-time cost, highlighting the promise of speculative context construction for scaling LLM-assisted software development to large, evolving codebases.

8 Limitations

The primary limitation of our work is the lack of publicly available benchmarks that eliminate *future-context leakage*. To evaluate SpecAgent in a realistic setting, we constructed a synthetic benchmark by modifying REPOCOD with a function removal agent to create synthetic indexing-time repository states. While this setup removes leakage, it is not derived from actual real-world repository histories, and thus the resulting performance may differ from what would be observed in production environments. Conversely, experiments under the original REPOCOD setting yield high pass rates, but these results are inflated by future-context leakage and likewise fail to reflect real-world performance.

Beyond benchmark availability, our work also has several other limitations. First, SpecAgent's speculative capabilities depend on the quality and breadth of its indexing-time exploration tools; in repositories with unconventional structures or sparse documentation, relevant context may still be missed. Second, although we demonstrate latency benefits at inference time, SpecAgent introduces additional computational overhead at indexing time, which may be non-trivial for extremely large or frequently changing repositories. Finally, our experiments are limited to two strong code completion models (Qwen3-8B and Qwen3-30B-A3B); it remains to be seen how well the approach generalizes to smaller models, multilingual codebases, or tasks beyond function completion, such as bug fixing or large-scale refactoring.

References

Aider. 2023. Repository map. Blog post.

Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. Blog post.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*.

Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Stafford Brunk, Niranjan Hasabnis, bfredl, Mingkai Dong, Vladimir Panteleev, ikrima, and 10 others. 2019. Tree-sitter.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*.

Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, and 1 others. 2024. R2C2-Coder: Enhancing and Benchmarking Real-world Repository-level Code Completion Abilities of Code Large Language Models. In *CoRR*.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 423–435.

Akash Dhruv and Anshu Dubey. 2025. Leveraging large language models for code translation and software development in scientific computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–9.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Wenchao Gu, Zongyi Lyu, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, and Michael R Lyu. 2025a. SPENCER: Self-Adaptive Model Distillation for Efficient Code Retrieval. *arXiv preprint arXiv:2508.00546*.

Wenchao Gu, Ensheng Shi, Yanlin Wang, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Michael R Lyu. 2025b. SECRET: Towards Scalable and Efficient Code Retrieval via Segmented Deep Hashing. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 2303–2315. IEEE.

Wenchao Gu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Michael Lyu. 2022. Accelerating Code Search with Deep Hashing and Code Classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 2534–2544.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225.

Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Language Models Meet Code Edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1495.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, and 1 others. 2024. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. In *CoRR*.

GitHub Inc. 2024. GitHub Copilot: Your AI Pair Programmer. Accessed: 2024-10-06.

Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Carlos Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*.

Denis Kocetkov, Raymond Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis,

Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and 1 others. 2023. The Stack: 3 TB of permissively licensed source code. In *Transactions on Machine Learning Research*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Wei Jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. In *CoRR*.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025a. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. *arXiv preprint arXiv:2506.12286*.

Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. 2025b. Can Language Models Replace Programmers? REPOCOD Says 'Not Yet'. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173*.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *The Thirteenth International Conference on Learning Representations*.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 87–94. IEEE.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-Coder: Let the Code Model Curate Data for Itself. *arXiv preprint arXiv:2506.03524*.

Cursor Team. 2024. Cursor. Accessed: 2024-10-06.

Qwen Team. 2025. Qwen3-Coder: Agentic Coding in the World. Blog post.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Zora Zhiruo Wang, Akari Asai, Frank F Xu, Yiqing Xie, Graham Neubig, Daniel Fried, and 1 others. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214.

Michael L. Waskom. 2021. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

Amazon Web Services. 2024. Amazon Q Developer: AI-Powered Coding Assistant. Accessed: 2024-10-06.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning*, pages 53270–53290.

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*.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code Representation Learning at Scale. In *The Twelfth International Conference on Learning Representations*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

Lei Zhang, Yunshui Li, Jiaming Li, Xiaobo Xia, Jiaxi Yang, Run Luo, Minzheng Wang, Longze Chen, Junhao Liu, Qiang Qu, and 1 others. 2025. Hierarchical Context Pruning: Optimizing Real-World Code Completion with Repository-Level Pretrained Code LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25886–25894.

Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2025. HumanEvo: An Evolution-Aware Benchmark for

More Realistic Evaluation of Repository-Level Code Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering*, pages 1372–1384. IEEE.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *The Thirteenth International Conference on Learning Representations*.

Appendix

Table of Contents

A Results with Qwen3-Coder as indexing-time agent **14**

B Agent implementation details **16**
- B.1 Retriever agent 16
- B.2 Forecaster agent 17
- B.3 SpecAgent 18
- B.4 Oracle agent 19
- B.5 Function removal agent 21
- B.6 Function removal scoring agent 22

C Benchmark construction details **25**
- C.1 Benchmark creation 25
- C.2 Benchmark validation 25

D Function removal example **26**
- D.1 Inference-time state 26
- D.2 Indexing-time state 27

E Code completion prompt example **30**

F Cross-file contexts examples **34**
- F.1 BM25 contexts 34
- F.2 RepoMap context 35
- F.3 Retriever agent contexts 36
- F.4 Forecaster agent 38
- F.5 SpecAgent contexts 39
- F.6 Oracle agent contexts 40

G Retrieval methods cross-comparisons **43**
- G.1 Evaluation setup 43
- G.2 BM25 versus oracle agent 44
- G.3 RepoMap versus oracle agent 51

H Hyperparameter settings **60**

I Potential risks **61**

J Licenses and responsible use **62**

A Results with Qwen3-Coder as indexing-time agent

In Section 6, we evaluated our indexing-time framework using Claude 3.7 Sonnet (Anthropic, 2025) as the agent and observed substantial gains over retrieval-based baselines. To assess generality across different agent backbones, we repeat the same experiments using Qwen3-Coder (Team, 2025) as the indexing-time agent. Across all configurations, our agentic framework continues to outperform baseline retrieval methods, confirming its robustness and architecture-agnostic benefits.

Model	Method	Pass@1 (%)	Latency (s)	Pre-processing Time (s)
Qwen3-8B	None	16.22	4.03	-
Qwen3-8B	BM25	17.55	5.01	-
Qwen3-8B	RepoMap	16.73	4.06	74.45
Qwen3-8B	Dense (UniXcoder)	16.22	7.15	-
Qwen3-8B	Dense (CodeSage v2 large)	15.41	11.50	-
Qwen3-8B	BM25 + RepoMap	17.65	4.72	74.45
Qwen3-8B	Retriever Agent (Ours)	17.76	4.26	41.45
Qwen3-8B	Forecaster Agent (Ours)	19.80	3.95	40.72
Qwen3-8B	**SpecAgent (Ours)**	**20.20**	3.80	41.45
Qwen3-8B	*Oracle Agent*	*27.04*	4.05	42.29
Qwen3-30B-A3B	None	18.37	5.03	-
Qwen3-30B-A3B	BM25	18.88	6.03	-
Qwen3-30B-A3B	RepoMap	17.45	5.54	74.45
Qwen3-30B-A3B	Dense (UniXcoder)	18.78	8.49	-
Qwen3-30B-A3B	Dense (CodeSage v2 large)	16.94	14.84	-
Qwen3-30B-A3B	BM25 + RepoMap	18.16	5.88	74.45
Qwen3-30B-A3B	Retriever Agent (Ours)	20.20	5.28	41.45
Qwen3-30B-A3B	Forecaster Agent (Ours)	21.73	5.72	40.72
Qwen3-30B-A3B	**SpecAgent (Ours)**	**24.69**	5.39	41.45
Qwen3-30B-A3B	*Oracle Agent*	*33.47*	5.39	42.29

Table 5: Main results using Qwen3-Coder as the indexing-time agent. SpecAgent achieves the highest pass@1 while maintaining inference-time latency comparable to baselines, demonstrating the benefits of combining retrieval and prediction at indexing time.

Main results. Table 5 reports the main results. Although Qwen3-Coder achieves slightly lower absolute scores than Claude 3.7 Sonnet (on average $\sim$5% lower), the same trends hold: both the *Retriever Agent* (retrieval only) and the *Forecaster Agent* (prediction only) outperform all standard retrieval baselines, and the full *SpecAgent*—combining retrieval and prediction—achieves the highest overall performance without introducing any inference-time latency. These results further validate that indexing-time synthesis and retrieval complement each other effectively.

Combined retrieval strategies. As in Section 6.3, we evaluate hybrid configurations combining SpecAgent's contexts with additional retrieved snippets from BM25 and dense retrievers. Table 6 shows a modest improvement from adding dense retrieval (UniXcoder), indicating that SpecAgent's precomputed contexts already capture most relevant information.

Inference-time ablation. We also test SpecAgent under an inference-time setup (retrieving contexts after removing the target function). This setting, while unrealistic due to information leakage, establishes an upper bound on achievable performance. As shown in Table 7, SpecAgent achieves higher pass@1 under this condition, consistent with trends reported in Section 6.3.

Composition ablation. Finally, we vary the ratio of retrieved versus predicted context blocks while fixing the total number (12). As shown in Table 8, performance peaks when using one or three prediction

Method	Pass@1 (%)
SpecAgent	20.20
SpecAgent + BM25	20.71
SpecAgent + Dense (UniXcoder)	**20.97**

Table 6: Effect of combining SpecAgent contexts with traditional retrievals. Qwen3-Coder is used as the indexing-time agent.

Model	Method	Pass@1 (%)
Qwen3-8B	SpecAgent (indexing)	20.20
Qwen3-8B	SpecAgent (inference)	**24.08**
Qwen3-30B-A3B	SpecAgent (indexing)	24.69
Qwen3-30B-A3B	SpecAgent (inference)	**27.14**

Table 7: Inference-time ablation for SpecAgent using Qwen3-Coder. Higher results stem from leakage, not from improved generalization.

blocks—mirroring results from Section 6.3. This reinforces that both speculative prediction and repository retrieval are essential for SpecAgent's success.

#Predictions	Pass@1 (%)
0	17.76
1	**21.84**
3	21.63
6	20.51
12	19.80

Table 8: Ablation on the composition of SpecAgent's retrieved and predicted context blocks. Qwen3-Coder serves as the indexing-time agent.

B Agent implementation details

This section documents the implementations of the agents introduced in Sections 4.1, 4.2 and 5.2. We focus on the operational details needed to reproduce our pipeline: tool access, execution workflow, and prompt templates. In the experiments reported in Section 6.1, we instantiate these agents with the backbone models described in the main text.

B.1 Retriever agent

The retriever agent is the retrieval-only indexing-time agent introduced in Section 4.1. It operates on the synthetic indexing-time repository state, after the target function has been removed, and produces a set of cross-file context blocks that are later cached and passed to the completion model at inference time.

Tools. The agent is given two exploration tools: a file-reading tool for inspecting repository contents and a shell tool for lightweight repository navigation. Before retrieval begins, it invokes the function removal agent described in Section B.5 to construct or reuse the leakage-free repository snapshot associated with the current REPOCOD instance.

Workflow. For each benchmark instance, the system first locates a persistent function-removed repository snapshot. If such a snapshot is not already available, it copies the repository, removes the target function together with its callers, and stores the resulting snapshot for reuse. The retriever agent then reads the target file after removal, explores related files and import chains, and emits a fixed number of context blocks. Each block contains a short natural-language explanation and one focused piece of evidence, such as a code snippet, dependency pattern, validation idiom, or API usage example. These blocks are parsed into structured records and cached for later inference-time use.

Prompt. The implementation uses a fixed system prompt together with an instance-specific task prompt. The output is parsed by looking for `<CONTEXT_START>` and `<CONTEXT_END>` delimiters.

```
[System prompt]
You are a code analysis agent that extracts relevant context for code completion tasks.

You have access to:
- The target file path: <TARGET_FILE_PATH>
- A repository where the target function has been removed
- All other files in the repository

Your task is to analyze the repository and extract contextual information that would help a code
↪  completion model when adding a new function to the target file. The target function has been cleanly
↪  removed from the repository, so you need to find relevant patterns, utilities, and examples from the
↪  remaining codebase.

RESPONSE FORMAT - Use this EXACT format:

<CONTEXT_START>
[Brief explanation of why this context is relevant]
[Your context content - can be code snippets, explanations, patterns, etc.]
<CONTEXT_END>

You can have multiple CONTEXT_START/CONTEXT_END blocks. Each block should contain ONE piece of relevant
↪  context.

ANALYSIS STRATEGY:
1. Start by examining the target file to understand its current structure and purpose
2. Analyze what kinds of features are missing from the target file, and what contexts would be helpful to
↪  add them
3. Look at import statements to understand dependencies
4. Find files in the same package/directory as the target file
5. Follow import chains to find relevant utility functions, classes, and patterns
6. Identify common patterns used throughout the codebase
7. Look for similar files or functions that might provide relevant context
8. Extract error handling, validation, and design patterns used in the codebase
9. Find configuration, constants, and data structures that might be relevant

CONTEXT TYPES YOU CAN PROVIDE:
- Code snippets from **other** files in the repository (with file paths)
- Import statements and dependency information
- Implementations and usages of imported functions and classes
- Class/function signatures and their usage patterns
- Error handling and validation patterns
- API usage examples
- Data structure definitions
- Constants and configuration values
```

````
- Test examples showing usage patterns
- Documentation and architectural insights
- Design patterns used in the codebase

GUIDELINES:
- Always include relative file paths when providing code snippets
- Do not include code snippets from the target file - they are already available to the code completion
↪  model
- Focus on context that would be helpful for someone adding a function to the target file
- Prioritize content from related files (same package, imported modules, etc.)
- Look for reusable patterns and utilities
- Each context should be focused and explain its relevance clearly
- Provide the most valuable context rather than quantity

TOOLS AVAILABLE:
- file_read: Read contents of specific files
- execute_bash: Run bash commands to explore repository structure

Remember: You are analyzing a repository where the target function has been cleanly removed. Your goal is
↪  to gather helpful context that would assist with adding a new function to the target file. The target
↪  file contents is already available to the code completion model, so please don't include them again in
↪  your contexts.

[Instance prompt]
TASK: Extract relevant context for adding a new function to a target file.

TARGET FILE: <TARGET_FILE_PATH>
REPOSITORY PATH: <TEMP_REPO_PATH>

TARGET FILE CONTENT (after function removal):
```python
<TARGET_FILE_CONTENT>
```

INSTRUCTIONS:
1. Analyze the repository structure to understand the codebase
2. Examine the target file to understand its current purpose and structure
3. Follow import statements to find relevant dependencies and utilities
4. Look for files in the same package/directory as the target file
5. Find similar patterns, utilities, and examples throughout the codebase
6. Focus on the <MAX_CONTEXT_FILES> most relevant files for context
7. Provide <MAX_CONTEXTS> context items using the required format

Since the target function has been removed, focus on:
- Understanding what the target file is for based on its current content
- Finding relevant utilities, patterns, and examples from the broader codebase
- Identifying common practices and conventions used in this repository
- Providing context that would be helpful for adding functionality to this type of file

Use the CONTEXT_START/CONTEXT_END format for each piece of context.
Provide a list of <MAX_CONTEXTS> context blocks in the CONTEXT_START/CONTEXT_END format, each containing
↪  one piece of relevant context.
Always include relative file paths when providing code snippets.
Use the file_read tool and execute_bash tool to explore the repository first before providing context.
````

B.2 Forecaster agent

The forecaster agent is the prediction-only indexing-time agent introduced in Section 4.1. Rather than retrieving repository evidence as the final output, it produces multiple plausible implementations of the missing function and uses these predicted implementations themselves as speculative context blocks.

Tools. The forecaster agent uses the same repository exploration tools as the retriever agent: file reading and shell-based repository navigation. It also relies on the function removal agent to ensure that forecasting is performed on the synthetic indexing-time repository state rather than on the leakage-prone inference-time repository.

Workflow. For each instance, the system first prepares or reuses the function-removed repository snapshot. The agent then inspects the target file after removal, studies repository structure and coding conventions, and proposes multiple candidate implementations for the missing function. Each candidate is intended to be a complete, standalone function body or function definition, representing a different plausible hypothesis about the future edit. These predicted implementations are stored as structured context blocks and cached for downstream composition in SpecAgent or for standalone ablations of the forecasting component.

Prompt. As with the retriever agent, the implementation uses a fixed system prompt and an instance-specific task prompt. Predicted implementations are parsed from `<PREDICTION_START>` and

<PREDICTION_END> blocks.

````
[System prompt]
You are a code prediction agent that generates multiple implementations of a missing function.

You have access to:
- The target file path: <TARGET_FILE_PATH>
- A repository where the target function has been removed
- All other files in the repository

Your task is to analyze the repository structure and generate MULTIPLE POSSIBLE IMPLEMENTATIONS of the
↪  missing function. You should NOT retrieve or quote existing code from other files - your job is purely
↪  to PREDICT what the missing function should look like.

RESPONSE FORMAT - Use this EXACT format:

<PREDICTION_START>
[Brief explanation of what this implementation does and why it might be correct]
```python
[Your predicted function implementation - complete Python code]
```
<PREDICTION_END>

You can have multiple PREDICTION_START/PREDICTION_END blocks. Each block should contain ONE complete
↪  function implementation.

PREDICTION STRATEGY:
1. Examine the target file to understand its structure, imports, and existing functionality
2. Analyze the file's context to understand what type of function might be missing
3. Look at the broader repository to understand coding patterns and conventions
4. Based on common patterns, generate multiple plausible implementations
5. Consider different complexity levels (simple, moderate, advanced implementations)
6. Think about edge cases and different approaches to the same problem

GUIDELINES:
- Generate ONLY function implementations - no other code snippets
- Each prediction should be a complete, standalone function
- Provide different variations/approaches for the same function
- Consider different levels of complexity and sophistication
- Include proper error handling, validation, and edge cases where appropriate
- Follow the coding style and patterns observed in the repository
- Make educated guesses based on function names, file context, and repository patterns
- DO NOT include code from other files - focus purely on prediction

TOOLS AVAILABLE:
- file_read: Read contents of specific files to understand context
- execute_bash: Run bash commands to explore repository structure

Remember: Your goal is to predict what the missing function should look like, not to retrieve existing
↪  code. Generate multiple creative and plausible implementations.

[Instance prompt]
TASK: Generate multiple predicted implementations for a missing function.

TARGET FILE: <TARGET_FILE_PATH>
REPOSITORY PATH: <TEMP_REPO_PATH>

TARGET FILE CONTENT (after function removal):
```python
<TARGET_FILE_CONTENT>
```

INSTRUCTIONS:
1. Analyze the target file structure and understand its purpose
2. Examine imports, class structure, and existing methods to understand context
3. Explore the repository structure to understand coding patterns and conventions
4. Generate <MAX_CONTEXTS> different predicted implementations for the missing function
5. Each prediction should be a complete, standalone function implementation
6. Consider different approaches: simple, moderate, and sophisticated implementations
7. Include proper error handling and edge cases where appropriate
8. Follow the coding style and patterns observed in the repository

FOCUS ON PREDICTION ONLY:
- DO NOT retrieve or quote existing code from other files
- Generate ONLY function implementations
- Each prediction should be creative and plausible
- Consider different complexity levels and approaches
- Think about what this function might realistically do based on its name and context

Use the PREDICTION_START/PREDICTION_END format for each predicted implementation.
Use the file_read tool and execute_bash tool to explore the repository first before providing context.
````

B.3 SpecAgent

SpecAgent is the hybrid indexing-time method studied throughout the paper. Conceptually, it combines the retriever agent and the forecaster agent described above. In the implementation, this combination is

not realized as a separate interactive agent class. Instead, it is constructed deterministically by merging the cached outputs of the two component agents.

Tools. SpecAgent inherits its effective tool access from its two constituent agents. During the composition stage itself, no additional interactive tools are required; the system only reads the cached outputs generated by the retriever and forecaster agents and writes the merged context list to a retrieval-compatible cache directory.

Workflow. The implementation first runs the retriever agent and the forecaster agent independently over the evaluation set and caches their outputs. A composition script then loads both caches for each benchmark instance, keeps the first n_{pred} predicted implementations from the forecaster, and fills the remaining slots with the highest-ranked retrieval contexts from the retriever. The resulting ordered list is saved in the same format expected by the inference pipeline, so the completion model can consume SpecAgent contexts exactly as it would consume a standard agent cache. In the main experiments of Section 6.2, we use three prediction blocks and nine retrieval blocks; Section 6.3 varies this ratio.

Prompt. SpecAgent does not instantiate an additional LLM agent and therefore has no standalone prompt. The only prompt-bearing components are the retriever and forecaster agents in Sections B.1 and B.2. The composition stage itself is deterministic: it loads the two caches for the same benchmark instance, keeps the first n_{pred} forecast blocks, appends retriever blocks until the context budget is filled, and saves the merged list in the cache format consumed by the inference pipeline.

B.4 Oracle agent

The oracle agent provides the upper-bound context construction setting discussed in Section 4.2. Unlike the indexing-time agents, it runs with access to the inference-time repository, the target function signature, and the full target-file contents. In the implementation, the oracle path uses the unmodified repository with full content, so the target-file contents supplied to the agent include the ground-truth implementation of the target function. The oracle is therefore an intentionally privileged upper bound rather than a realistic deployment setting.

Tools. The oracle agent uses the same two repository exploration tools as the retriever agent: file reading and shell-based navigation. It does not call the function removal agent, because it is intentionally evaluated on the inference-time repository state.

Workflow. For each benchmark instance, the system first reads the target file from the original repository and passes the resulting full file contents, together with the target function signature, to the oracle prompt. It also creates a temporary copy of the repository for exploration. The agent then searches the repository and emits structured context blocks. After generation, the parser applies a leakage filter before returning the final contexts. This filter removes blocks that appear to implement the target function by matching the target signature or strong implementation-oriented textual indicators in the content or explanation. The metadata records how many raw blocks were generated and how many survived filtering. This post-processing step is essential because the oracle agent is given direct access to the ground-truth target-file contents.

Prompt. The implementation again uses a system prompt followed by an instance-specific task prompt. The text below matches the prompt strings in the codebase, with concrete paths and file contents replaced by placeholders.

```
[System prompt]
You are a specialized code analysis agent that extracts relevant context for code completion tasks.

Your task is to analyze a repository and extract contextual information that would help a code completion
↪  model complete a specific function. You must NOT provide the ground-truth implementation of the target
↪  function - only provide helpful context from OTHER files in the repository.

IMPORTANT: Use the following EXACT format for your responses:

<CONTEXT_START>
[Brief explanation of why this context is relevant]
```

```
[Your context content - can be code snippets, explanations, patterns, etc.]
<CONTEXT_END>

You can have multiple CONTEXT_START/CONTEXT_END blocks. Each block should contain ONE piece of relevant
↪  context.

CRITICAL CONSTRAINTS - CONTEXTS WILL BE AUTOMATICALLY REMOVED IF THEY VIOLATE THESE:
1. DO NOT provide any solution for the target function
2. DO NOT provide any code snippets from the target file: <TARGET_FILE_PATH>
3. ALWAYS include the relative file path when providing code snippets (e.g., "mypackage/utils/helpers.py")
4. Any context violating these rules will be automatically filtered out

ANALYSIS STRATEGY:
1. DO NOT provide any code snippets from the target file itself - that content is already available to the
↪  code completion model
2. The target file path is: <TARGET_FILE_PATH> - AVOID this file completely
3. Focus on analyzing OTHER files in the repository to find relevant context
4. Follow import statements to understand dependencies
5. Look for similar function patterns in related files (same package, similar names, etc.)
6. Identify utility functions, classes, and patterns that might be relevant
7. Extract common error handling and validation patterns used in this codebase
8. When providing code snippets, ALWAYS include the relative file path within the repository

CONTEXT TYPES YOU CAN PROVIDE (from OTHER files only):
- Code snippets from **other** files in the repository (with file paths)
- Import statements and dependency information from other files
- Class/function signatures and structures from related files
- Error handling patterns from other files
- Validation patterns from other files
- Similar function implementations from other files
- API usage patterns from other files
- Data structure definitions from other files
- Constants and configuration values from other files
- Test examples that show how similar functionality is used
- Documentation or comments that explain relevant concepts

ABSOLUTELY FORBIDDEN - THESE WILL BE AUTOMATICALLY REMOVED:
- Any completion of the target function
- Any code snippets from <TARGET_FILE_PATH>
- Any context without a clear relative file path for code snippets
- Any solutions or direct answers to what the target function should do
- Any code that looks like it could be the target function implementation

CONSTRAINTS:
- DO NOT provide any code snippets from the target file itself
- DO NOT provide solutions or implementations for the target function
- When providing code snippets, ALWAYS include the relative file path (e.g., "mypackage/utils/helpers.py")
- Prioritize quality over quantity - provide the most relevant context
- Each context block should be focused and specific
- Explain why each context is relevant in 1-2 sentences

TOOLS AVAILABLE:
- file_read: Read contents of specific files
- execute_bash: Run bash commands to explore repository structure

Remember: Your job is to provide helpful CONTEXT from OTHER files, not to solve the problem. The context
↪  will be provided as hints to help a code completion model. Any contexts that complete the target
↪  function, provide code from the target file, or lack proper file paths will be automatically removed.

[Instance prompt]
TASK: Extract relevant context for completing a Python function.

TARGET FILE: <TARGET_FILE_PATH>
REPOSITORY PATH: <TEMP_REPO_PATH>

FUNCTION TO COMPLETE:
<FUNCTION_SIGNATURE>

TARGET FILE CONTENT:
'''python
<TARGET_FILE_CONTENT>
'''

INSTRUCTIONS:
1. Explore the repository structure to understand the codebase
2. DO NOT provide any code snippets from the target file (<TARGET_FILE_PATH>) - it is already available to
↪  the code completion model
3. Find similar functions, utility classes, and relevant patterns in OTHER files
4. Extract context that would help complete the target function - but DO NOT implement the target function
5. Focus on the <MAX_CONTEXT_FILES> most relevant files (excluding the target file)
6. Provide up to <MAX_CONTEXTS> context items using the required format

CRITICAL WARNINGS - These contexts will be automatically REMOVED:
- Any implementation or completion of the target function
- Any code snippets from <TARGET_FILE_PATH>
- Any code snippets without relative file paths
- Any solutions that show how to implement the target function
```

```
Remember to use the CONTEXT_START/CONTEXT_END format for each piece of context you provide.
Do NOT provide the ground truth implementation of the target function - only provide helpful context from
↪  OTHER files.
Do NOT provide any code snippets from the target file itself.
Include relative file paths when providing code snippets from other files.
Focus on patterns, utilities, and examples from OTHER files that could inform the implementation.
Use the file_read tool and execute_bash tool to explore the repository first before providing context.
```

B.5 Function removal agent

The function removal agent constructs the synthetic indexing-time repository state used by the retriever and forecaster agents. Its purpose is to remove the target function and all semantically revealing callers while preserving syntactic validity and, as far as possible, the functional coherence of the remaining repository. This agent is therefore the core mechanism that eliminates future context leakage in the benchmark described in Sections C and 5.2.

Tools. This agent has a richer tool set than the context-construction agents. In addition to file reading and shell navigation, it is given repository editing tools together with tree-sitter-based function and class removal tools. The tree-sitter analysis identifies candidate references to the target function, including top-level functions, class methods, decorated functions, and qualified calls, and the agent then uses the editing tools to carry out repository-wide removal and cleanup.

Workflow. Given a repository snapshot and a target function, the system first enumerates all Python files and uses tree-sitter to detect candidate call sites and, when needed, the target function definition itself. It then summarizes these references for the LLM agent and asks the agent to remove the function definition, delete or rewrite all callers, clean up now-unused imports, and preserve syntactic validity. The resulting repository snapshot is stored persistently and reused across later retrieval and forecasting runs. In practice, this procedure removes not only direct calls but also many indirect clues, such as imports and local scaffolding that would otherwise reveal the target function's intended behavior.

Prompt. The removal pipeline again uses a reusable system prompt and an instance-specific task prompt. The second prompt contains a tree-sitter-derived summary of candidate references before the LLM edits the repository.

```
[System prompt]
You are a specialized code modification agent that removes specific functions and all their callers from
↪  Python files.

Your task is to analyze Python files and remove:
1. The target function definition
2. All calls to the target function throughout the repository
3. Any import statements that become unused after removal

While preserving:
1. File structure and syntax
2. Other functions and classes
3. Necessary import statements
4. Module-level code
5. Comments and docstrings (except those within removed code)

RESPONSE FORMAT - Use this EXACT format:

<RESULT>
[Brief explanation of what was done]
<RESULT_END>

ANALYSIS STRATEGY:
1. Read and understand the code structure
2. Identify the target function and all its call sites
3. Remove function calls intelligently:
   - If the call is a standalone statement, remove the entire statement
   - If the call is part of an expression, remove or replace appropriately
   - If the call is in a condition, handle the conditional logic
   - If the call is in an assignment, remove or modify the assignment
4. Remove the function definition itself
5. Clean up any unused imports
6. Ensure all modified files remain syntactically valid
7. Report comprehensive details of what was removed

TOOLS AVAILABLE:
- file_read: Read contents of specific files
```

```
- execute_bash: Run bash commands to explore repository structure
- editor: Edit files (view, str_replace, etc.)
- remove_class: Remove entire classes from Python files using tree-sitter
- remove_function: Remove specific functions from Python files using tree-sitter

CRITICAL CONSTRAINTS:
- Remove the target function AND all its callers
- Do not leave any comments about the removal
- Preserve proper indentation and formatting
- Ensure all files remain syntactically valid
- Handle edge cases like method calls, decorated functions, etc.
- Be intelligent about removing calls without breaking code flow
- Report if the target function cannot be found

Remember: Your goal is complete removal of the target function and all references to it while keeping the
↪  repository functional.

[Instance prompt]
TASK: Remove a specific function and ALL its callers from a Python repository.

REPOSITORY PATH: <TEMP_REPO_PATH>
TARGET FILE: <TARGET_FILE_PATH>
FUNCTION TO REMOVE: <FUNCTION_NAME>

FUNCTION REFERENCES FOUND:
<REFERENCES_SUMMARY>

INSTRUCTIONS:
1. Remove the function definition from <TARGET_FILE_PATH>
2. Remove ALL calls to this function from the repository
3. For each function call, handle removal intelligently:
   - If it's a standalone statement: Remove the entire statement
   - If it's part of an assignment: Remove or modify the assignment appropriately
   - If it's in a conditional: Handle the condition logic properly
   - If it's in an expression: Replace or remove the expression safely
   - If it's a method call: Remove the call and handle the result appropriately

4. Ensure all modified files remain syntactically valid
5. Clean up any imports that become unused after removing calls
6. Handle edge cases like:
   - Function calls in list comprehensions
   - Function calls as arguments to other functions
   - Function calls in return statements
   - Method calls on objects

CRITICAL REQUIREMENTS:
- The function <FUNCTION_NAME> should not exist ANYWHERE in the repository after removal
- Do not leave any comments about the removal
- All Python files should remain syntactically valid
- Preserve the overall structure and functionality of unrelated code
- Be intelligent about removing calls without breaking program flow

FILES TO POTENTIALLY MODIFY:
<PYTHON_FILE_LIST>

Please proceed systematically through each file and remove all references to <FUNCTION_NAME>.
Report comprehensively what was removed from each file.
```

B.6 Function removal scoring agent

The function removal scoring agent is used in the benchmark validation loop summarized in Section C. Its purpose is to assess whether the synthetic indexing-time repository state is sufficiently leakage-free to be retained in the benchmark. The agent does not modify the repository. Instead, it inspects the post-removal snapshot and assigns a structured quality score.

Tools. The scoring agent uses only repository exploration tools: file reading and shell-based navigation. It receives the target file path and target function name and is asked to reason about whether any remaining definitions or suspicious calls are likely to refer to the removed function.

Workflow. The validator first runs basic syntactic checks using tree-sitter and then invokes the scoring agent on the repository snapshot. The agent examines the target file for residual definitions, searches for potentially related calls across the repository, reasons about import structure and call context, and returns a score in the range 0–5 together with a confidence value and a free-form explanation. Repositories with scores below a threshold of 4 are treated as problematic and are either refined or regenerated, yielding the iterative validation loop described in Section C.

Prompt. The scoring agent also uses a fixed system prompt plus an instance-specific task prompt. The system prompt defines the response fields and the 0–5 scale used by the validator.

```
[System prompt]
You are a specialized code analysis agent that evaluates how well a specific function has been removed
↪  from a Python repository.

Your task is to analyze a Python repository and provide a detailed assessment of function removal quality.

ANALYSIS APPROACH:
1. Check if the target function definition still exists (critical factor)
2. Identify function calls that might be related to the target function
3. Use context clues to determine if remaining calls are likely to the target function:
   - Import patterns
   - Variable assignment patterns
   - Class instantiation patterns
   - Method chaining patterns
   - Documentation and comments

4. Assess the likelihood that remaining calls are actually calls to the target function
5. Consider edge cases like inheritance, polymorphism, and name collisions

SCORING CRITERIA (0-5 scale):
- 5: Perfect removal - function definition gone, no suspicious calls remain
- 4: Good removal - definition gone, only clearly unrelated calls with same name remain
- 3: Moderate removal - definition gone, some potentially related calls remain
- 2: Poor removal - definition gone, many likely related calls remain
- 1: Very poor removal - definition exists OR many obvious calls to target function remain
- 0: No removal detected - target function definition clearly exists and unchanged

RESPONSE FORMAT - Use this EXACT format:

<ANALYSIS>
[Detailed analysis of what you found]
<ANALYSIS_END>

<SCORE>
[Integer score 0-5]
<SCORE_END>

<CONFIDENCE>
[Float confidence 0.0-1.0]
<CONFIDENCE_END>

<EXPLANATION>
[Human-readable explanation of the score]
<EXPLANATION_END>

TOOLS AVAILABLE:
- file_read: Read contents of specific files
- execute_bash: Run bash commands to explore repository structure

CRITICAL CONSTRAINTS:
- Focus on the SPECIFIC target function, not all functions with the same name
- Consider context and import patterns to distinguish function calls
- Be conservative - if unsure whether a call is to the target function, assume it might be
- Provide detailed reasoning for your score
- Consider both false positives (unrelated functions) and false negatives (missed calls)

Remember: The goal is accurate assessment of removal quality for the SPECIFIC target function.

[Instance prompt]
TASK: Analyze and score how well a specific function has been removed from a Python repository.

REPOSITORY PATH: <TEMP_REPO_PATH>
TARGET FILE: <TARGET_FILE_PATH>
FUNCTION TO ASSESS: <FUNCTION_NAME>

ANALYSIS INSTRUCTIONS:
1. First, check if the target function definition still exists in <TARGET_FILE_PATH>
   - Look for the exact function: <FUNCTION_NAME>
   - Consider decorated functions, class methods, etc.

2. Search the entire repository for potential calls to this function
   - Look for direct calls: <FUNCTION_NAME>()
   - Look for qualified calls: obj.<SIMPLE_NAME>()
   - Look for import-based calls

3. For each potential call, assess the likelihood it's actually calling the target function:
   - Check import statements and module paths
   - Look at variable assignments and object instantiation
   - Consider class inheritance and method resolution
   - Analyze context clues in surrounding code

4. Pay special attention to:
   - Class methods vs. instance methods
```

```
   - Functions with common names that might exist in multiple classes
   - Method chaining and attribute access patterns
   - Test files that might import or reference the function

5. Provide a score (0-5) based on removal quality:
   - How thoroughly was the target function removed?
   - How many likely calls to the target function remain?
   - What's the confidence that remaining calls are NOT to the target function?

Please analyze the repository systematically and provide your assessment.
```

C Benchmark construction details

This appendix provides additional details on the construction of our benchmark for indexing-time context retrieval. We describe both the creation of indexing-time repository states and the validation process used to ensure their quality.

C.1 Benchmark creation

Given a target function and its associated inference-time repository, our goal is to synthesize a plausible state of the repository from an earlier point in time, before the target function has been implemented. This reflects a realistic scenario in which the user has not yet begun authoring the target function, and the retrieval agent operates asynchronously during indexing time. As discussed in Section 5.1, residual references to the target function can remain even after removing its body, including call sites, imports, and docstrings.

To construct a repository state that is both free of future context leakage and functionally intact, we introduce a *function removal agent*. This agent is guided by static analysis tools and is equipped with shell and file manipulation tools to explore the repository. It edits files to eliminate all explicit and implicit references to the target function while preserving the functional correctness of the remaining codebase. The indexing-time state produced by this process serves as the foundation for all experiments in the main paper.

C.2 Benchmark validation

To ensure the quality and reliability of our synthetic benchmark, we implement a validation procedure that evaluates and refines the constructed indexing-time repository states. Specifically, we introduce a *function removal scoring agent* that explores the repository and assigns a score to the quality of the function removal.

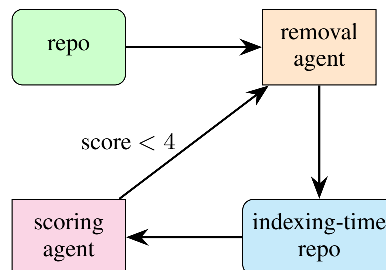


Figure 3: The benchmark validation loop.

The scoring agent is prompted to analyze whether all references to the target function have been removed and whether the repository remains functional. It produces a quality score from 0 to 5. If the score falls below 4, the function removal agent is reapplied to further refine the repository state. This loop is repeated until the repository receives a score of at least 4. We apply this iterative validation and refinement pipeline across all target functions in our dataset to ensure that future context leakage is effectively mitigated and that the benchmark reliably reflects indexing-time retrieval scenarios.

D Function removal example

In the main text, we introduced the issue of *future-context leakage* in existing code completion benchmarks and proposed a method to construct synthetic indexing-time repository states that avoid leaking information about the target function. We argued that relying solely on static analysis tools to remove callers of the target function is insufficient, since semantic and contextual clues may still enable an agent to infer the target function. To address this, we introduced a *function removal agent* that constructs the synthetic indexing-time state, as well as a *function removal scoring agent* that validates the quality of this process.

In this section, we provide a concrete example from the seaborn (Waskom, 2021) repository. We illustrate the states of both the target file (containing the target function) and a caller file (which imports and invokes the target function) under inference-time and indexing-time states. We also highlight what information is removed in the synthetic indexing-time state and explain why static analysis alone cannot achieve the same effect.

D.1 Inference-time state

We examine an example from the REPOCOD benchmark (Liang et al., 2025b), specifically repository ID 33 corresponding to the seaborn repository. The target function is `color_palette`, defined in `seaborn/palettes.py`. The inference-time state of the target file is shown below, with the definition and reference to the target function highlighted.

```
import colorsys
from itertools import cycle
import numpy as np
import matplotlib as mpl
from .external import husl
from .utils import desaturate, get_color_cycle
from .colors import xkcd_rgb, crayons
from ._compat import get_colormap


__all__ = ["color_palette", "hls_palette", "husl_palette", "mpl_palette",
           "dark_palette", "light_palette", "diverging_palette",
           "blend_palette", "xkcd_palette", "crayon_palette",
           "cubehelix_palette", "set_color_codes"]

...

class _ColorPalette(list):
    ...

def _patch_colormap_display():
    ...

def color_palette(palette=None, n_colors=None, desat=None, as_cmap=False):
    """Return a list of colors or continuous colormap defining a palette.

    ..."""
    if palette is None:
        palette = get_color_cycle()
        if n_colors is None:
            n_colors = len(palette)

    elif not isinstance(palette, str):
        palette = palette
        if n_colors is None:
            n_colors = len(palette)
    else:
        ...

    return palette

def hls_palette(n_colors=6, h=.01, l=.6, s=.65, as_cmap=False):  # noqa
    ...
```

As seen in the code, the target function `color_palette` is defined just before the `hls_palette` function and is also referenced in the `__all__` variable at the beginning of the file. The latter reference introduces future context leakage, since retrieval methods may leverage the `__all__` declaration to deduce the existence of the target function.

We also examine another file, `seaborn/axisgrid.py`, which imports and calls the target function. The inference-time state of this caller file is shown below, with references to the target function highlighted.

```python
from __future__ import annotations
from itertools import product
from inspect import signature
import warnings
from textwrap import dedent

...
from .palettes import color_palette, blend_palette
from ._docstrings import (
    DocstringComponents,
    _core_docs,
)

__all__ = ["FacetGrid", "PairGrid", "JointGrid", "pairplot", "jointplot"]

_param_docs = DocstringComponents.from_nested_components(
    core=_core_docs["params"],
)

...

class Grid(_BaseGrid):
    """A grid that can have multiple subplots and an external legend."""
    _margin_titles = False
    _legend_out = True

    def __init__(self):
        ...

    def _get_palette(self, data, hue, hue_order, palette):
        """Get a list of colors for the hue variable."""
        if hue is None:
            palette = color_palette(n_colors=1)

        else:
            hue_names = categorical_order(data[hue], hue_order)
            n_colors = len(hue_names)

            # By default use either the current color palette or HUSL
            if palette is None:
                current_palette = utils.get_color_cycle()
                if n_colors > len(current_palette):
                    colors = color_palette("husl", n_colors)
                else:
                    colors = color_palette(n_colors=n_colors)

            # Allow for palette to map from hue variable names
            elif isinstance(palette, dict):
                color_names = [palette[h] for h in hue_names]
                colors = color_palette(color_names, n_colors)

            # Otherwise act as if we just got a list of colors
            else:
                colors = color_palette(palette, n_colors)

            palette = color_palette(colors, n_colors)

        return palette
```

Here, the target function is imported at the top of the file and called multiple times inside the `_get_palette` function. This poses a strong leakage risk: for example, BM25 may retrieve code chunks containing these call sites, which would not exist in real-world settings where the user has not yet implemented the function. By analyzing such call sites, an indexing-time agent could infer the intended behavior of the missing function and even generate its complete implementation, artificially inflating benchmark performance.

Attempting to construct synthetic indexing-time states with static analysis tools—for example, by simply removing the lines that call the target function—proves inadequate. Such removal leaves behind empty or broken control structures, from which an intelligent agent can still deduce the intended role of the function. This motivates our use of an agent-based approach to reliably construct leakage-free states, as demonstrated below.

D.2 Indexing-time state

We now show the indexing-time states of the target and caller files produced by the function removal agent. For the target file, both the function definition and its reference in the `__all__` variable are removed, eliminating obvious leakage channels.

```python
import colorsys
from itertools import cycle

import numpy as np
import matplotlib as mpl

from .external import husl

from .utils import desaturate, get_color_cycle
from .colors import xkcd_rgb, crayons
from ._compat import get_colormap


__all__ = ["hls_palette", "husl_palette", "mpl_palette",
           "dark_palette", "light_palette", "diverging_palette",
           "blend_palette", "xkcd_palette", "crayon_palette",
           "cubehelix_palette", "set_color_codes"]

...

class _ColorPalette(list):
    ...


def _patch_colormap_display():
    ...


def hls_palette(n_colors=6, h=.01, l=.6, s=.65, as_cmap=False):  # noqa
    ...
```

For the caller file, the function removal agent eliminates both the import statement and all call sites of the target function. To preserve code functionality and avoid leaving broken logic, the agent further introduces a helper function, `_process_palette`, as a replacement for the removed function. This ensures that the caller file remains coherent and executable, while also preventing any information leakage that could enable an indexing-time agent or retrieval method to infer the target function. In this way, the synthetic indexing-time state avoids future context leakage and provides a more realistic evaluation environment.

```python
from __future__ import annotations
from itertools import product, cycle
from inspect import signature
import warnings
from textwrap import dedent

...
from .palettes import blend_palette, husl_palette, SEABORN_PALETTES
from ._docstrings import (
    DocstringComponents,
    _core_docs,
)

def _process_palette(palette=None, n_colors=None):
    """Internal palette processing function."""
    if palette is None:
        palette = get_color_cycle()
        if n_colors is None:
            n_colors = len(palette)
    elif not isinstance(palette, str):
        if n_colors is None:
            n_colors = len(palette)
    else:
        ...
    return palette

__all__ = ["FacetGrid", "PairGrid", "JointGrid", "pairplot", "jointplot"]

_param_docs = DocstringComponents.from_nested_components(
    core=_core_docs["params"],
)

...

class Grid(_BaseGrid):
    """A grid that can have multiple subplots and an external legend."""
    _margin_titles = False
    _legend_out = True

    def __init__(self):
        ...
```

```
    def _get_palette(self, data, hue, hue_order, palette):
        """Get a list of colors for the hue variable."""
        if hue is None:
            palette = _process_palette(n_colors=1)

        else:
            hue_names = categorical_order(data[hue], hue_order)
            n_colors = len(hue_names)

            # By default use either the current color palette or HUSL
            if palette is None:
                current_palette = utils.get_color_cycle()
                if n_colors > len(current_palette):
                    colors = husl_palette(n_colors)
                else:
                    colors = _process_palette(n_colors=n_colors)

            # Allow for palette to map from hue variable names
            elif isinstance(palette, dict):
                color_names = [palette[h] for h in hue_names]
                colors = _process_palette(color_names, n_colors)

            # Otherwise act as if we just got a list of colors
            else:
                colors = _process_palette(palette, n_colors)

            palette = _process_palette(colors, n_colors)

        return palette
```

E Code completion prompt example

The construction of prompts for the code completion model varies depending on the retrieval method. Each prompt is designed to provide the model with sufficient contextual information to complete the target function accurately. In particular, the prompt is structured as follows:

1. *Target file path*: The path to the source file containing the target function.
2. *Left context*: The content preceding the target function within the same file.
3. *Right context*: The content following the target function within the same file.
4. *Cross-file contexts* (optional): Additional relevant contexts retrieved from other files in the repository.
5. *Function signature and docstring*: The header and documentation string of the target function.

To illustrate, we provide below an example prompt constructed using the BM25 retrieval method. The example is drawn from the seaborn (Waskom, 2021) repository, corresponding to repository ID 0 in the REPOCOD benchmark. The full prompt presented to the code completion model is as follows:

````
This is the file that contains the target function to be generated.

## File path: seaborn/_core/scales.py

### Context before the target function
```python
from __future__ import annotations
import re
from copy import copy
from collections.abc import Sequence
from dataclasses import dataclass
from functools import partial
from typing import Any, Callable, Tuple, Optional, ClassVar
...
from matplotlib.axis import Axis
from matplotlib.scale import ScaleBase
from pandas import Series

from seaborn._core.rules import categorical_order
from seaborn._core.typing import Default, default

from typing import TYPE_CHECKING
...


@dataclass
class Continuous(ContinuousBase):
    """
    A numeric scale supporting norms and functional transforms.
    """
    values: tuple | str | None = None
    trans: str | TransFuncs | None = None

    # TODO Add this to deal with outliers?
    # outside: Literal["keep", "drop", "clip"] = "keep"

    _priority: ClassVar[int] = 1

    def tick(
        self,
        locator: Locator | None = None, *,
        at: Sequence[float] | None = None,
        upto: int | None = None,
        count: int | None = None,
        every: float | None = None,
        between: tuple[float, float] | None = None,
        minor: int | None = None,
    ) -> Continuous:
        """
        Configure the selection of ticks for the scale's axis or legend.

        ..."""
        # Input checks
        if locator is not None and not isinstance(locator, Locator):
            raise TypeError(
                f"Tick locator must be an instance of {Locator!r}, "
                f"not {type(locator)!r}."
            )
        log_base, symlog_thresh = self._parse_for_log_params(self.trans)
````
````

````
        if log_base or symlog_thresh:
            if count is not None and between is None:
                raise RuntimeError("`count` requires `between` with log transform.")
            if every is not None:
                raise RuntimeError("`every` not supported with log transform.")

        new = copy(self)
        new._tick_params = {
            "locator": locator,
            "at": at,
            "upto": upto,
            "count": count,
            "every": every,
            "between": between,
            "minor": minor,
        }
        return new
```

### Context after the target function
```python
    def _parse_for_log_params(
        self, trans: str | TransFuncs | None
    ) -> tuple[float | None, float | None]:
        ...

    def _get_locators(self, locator, at, upto, count, every, between, minor):
        ...

    def _get_formatter(self, locator, formatter, like, base, unit):
        ...


@dataclass
class Temporal(ContinuousBase):
    """
    A scale for date/time data.
    """
    ...

...
```

### Relevant context from other files of the repo
```python
# Code from: seaborn/categorical.py
    width=dedent("""\
    width : float
        Width allotted to each element on the orient axis. When `native_scale=True`,
        it is relative to the minimum distance between two values in the native scale.\
    """),
    dodge=dedent("""\
    dodge : "auto" or bool
        When hue mapping is used, whether elements should be narrowed and shifted along
        the orient axis to eliminate overlap. If `"auto"`, set to `True` when the
        orient variable is crossed with the categorical variable or `False` otherwise.
        .. versionchanged:: 0.13.0
            Added `"auto"` mode as a new default.\
    """),
    linewidth=dedent("""\
    linewidth : float
        Width of the lines that frame the plot elements.\
    """),
    linecolor=dedent("""\
    linecolor : color
        Color to use for line elements, when `fill` is True.
        .. versionadded:: v0.13.0\
    """),
    log_scale=dedent("""\
    log_scale : bool or number, or pair of bools or numbers
        Set axis scale(s) to log. A single value sets the data axis for any numeric
        axes in the plot. A pair of values sets each axis independently.
        Numeric values are interpreted as the desired base (default 10).
        When `None` or `False`, seaborn defers to the existing Axes scale.
        .. versionadded:: v0.13.0\
    """),
    native_scale=dedent("""\
    native_scale : bool
        When True, numeric or datetime values on the categorical axis will maintain
        their original scaling rather than being converted to fixed indices.
        .. versionadded:: v0.13.0\
    """),
    formatter=dedent("""\
    formatter : callable
        Function for converting categorical data into strings. Affects both grouping
        and tick labels.
        .. versionadded:: v0.13.0\
````

````
    """),
    legend=dedent("""\
    legend : "auto", "brief", "full", or False
        How to draw the legend. If "brief", numeric `hue` and `size`
        variables will be represented with a sample of evenly spaced values.
        If "full", every group will get an entry in the legend. If "auto",
        choose between brief or full representation based on number of levels.
        If `False`, no legend data is added and no legend is drawn.
        .. versionadded:: v0.13.0\
    """),

# Code from: seaborn/_core/plot.py
"variables": variables,
            "structure": structure,
            "wrap": wrap,
        }
        new = self._clone()
        new._facet_spec.update(spec)
        return new
    # TODO def twin()?
    def scale(self, **scales: Scale) -> Plot:
        """
        Specify mappings from data units to visual properties.
        Keywords correspond to variables defined in the plot, including coordinate
        variables (`x`, `y`) and semantic variables (`color`, `pointsize`, etc.).
        A number of "magic" arguments are accepted, including:
            - The name of a transform (e.g., `"log"`, `"sqrt"`)
            - The name of a palette (e.g., `"viridis"`, `"muted"`)
            - A tuple of values, defining the output range (e.g. `(1, 5)`)
            - A dict, implying a :class:`Nominal` scale (e.g. `{"a": .2, "b": .5}`)
            - A list of values, implying a :class:`Nominal` scale (e.g. `["b", "r"]`)
        For more explicit control, pass a scale spec object such as :class:`Continuous`
        or :class:`Nominal`. Or pass `None` to use an "identity" scale, which treats
        data values as literally encoding visual properties.
        Examples
        --------
        .. include:: ../docstrings/objects.Plot.scale.rst
        """
        new = self._clone()
        new._scales.update(scales)
        return new
    def share(self, **shares: bool | str) -> Plot:
        """
        Control sharing of axis limits and ticks across subplots.
        Keywords correspond to variables defined in the plot, and values can be
        boolean (to share across all subplots), or one of "row" or "col" (to share
        more selectively across one dimension of a grid).
        Behavior for non-coordinate variables is currently undefined.
        Examples
        --------
        .. include:: ../docstrings/objects.Plot.share.rst
        """
        new = self._clone()
        new._shares.update(shares)
        return new
    def limit(self, **limits: tuple[Any, Any]) -> Plot:
        """
        Control the range of visible data.
        Keywords correspond to variables defined in the plot, and values are a
        `(min, max)` tuple (where either can be `None` to leave unset).
        Limits apply only to the axis; data outside the visible range are
        still used for any stat transforms and added to the plot.
        """

...
```

### Target function to complete

```python
    def label(
        self,
        formatter: Formatter | None = None, *,
        like: str | Callable | None = None,
        base: int | None | Default = default,
        unit: str | None = None,
    ) -> Continuous:
        """
        Configure the appearance of tick labels for the scale's axis or legend.

        Parameters
        ----------
        formatter : :class:`matplotlib.ticker.Formatter` subclass
            Pre-configured formatter to use; other parameters will be ignored.
        like : str or callable
            Either a format pattern (e.g., `".2f"`), a format string with fields named
            `x` and/or `pos` (e.g., `"${x:.2f}"`), or a callable with a signature like
````

```
            `f(x: float, pos: int) -> str`. In the latter variants, `x` is passed as the
            tick value and `pos` is passed as the tick index.
        base : number
            Use log formatter (with scientific notation) having this value as the base.
            Set to `None` to override the default formatter with a log transform.
        unit : str or (str, str) tuple
            Use  SI prefixes with these units (e.g., with `unit="g"`, a tick value
            of 5000 will appear as `5 kg`). When a tuple, the first element gives the
            separator between the number and unit.

        Returns
        -------
        scale
            Copy of self with new label configuration.

        """

```

Please complete the target function and do not output anything else. Do not attach any docstrings.

F Cross-file contexts examples

In this section, we present an example from the seaborn repository (repository ID 0 in the REPOCOD benchmark) to illustrate the cross-file contexts retrieved by different methods: BM25, RepoMap, SpecAgent, and the Oracle Agent. The target function in this case is `Continuous.label`, whose ground-truth implementation is shown below:

```
def label(
    self,
    formatter: Formatter | None = None, *,
    like: str | Callable | None = None,
    base: int | None | Default = default,
    unit: str | None = None,
) -> Continuous:
    """
    Configure the appearance of tick labels for the scale's axis or legend.

    ..."""
    # Input checks
    if formatter is not None and not isinstance(formatter, Formatter):
        raise TypeError(
            f"Label formatter must be an instance of {Formatter!r}, "
            f"not {type(formatter)!r}"
        )
    if like is not None and not (isinstance(like, str) or callable(like)):
        msg = f"`like` must be a string or callable, not {type(like).__name__}."
        raise TypeError(msg)

    new = copy(self)
    new._label_params = {
        "formatter": formatter,
        "like": like,
        "base": base,
        "unit": unit,
    }
    return new
```

F.1 BM25 contexts

The cross-file contexts retrieved by BM25 are presented below, consisting of code chunks along with their respective relative file paths. There are a total of 12 context blocks, but only the initial two are displayed, and the remaining blocks are omitted due to their substantial length.

```
# Code from: seaborn/categorical.py
    width=dedent("""\
    width : float
        Width allotted to each element on the orient axis. When `native_scale=True`,
        it is relative to the minimum distance between two values in the native scale.\
    """),
    dodge=dedent("""\
    dodge : "auto" or bool
        When hue mapping is used, whether elements should be narrowed and shifted along
        the orient axis to eliminate overlap. If `"auto"`, set to `True` when the
        orient variable is crossed with the categorical variable or `False` otherwise.
        .. versionchanged:: 0.13.0
            Added `"auto"` mode as a new default.\
    """),
    linewidth=dedent("""\
    linewidth : float
        Width of the lines that frame the plot elements.\
    """),
    linecolor=dedent("""\
    linecolor : color
        Color to use for line elements, when `fill` is True.
        .. versionadded:: v0.13.0\
    """),
    log_scale=dedent("""\
    log_scale : bool or number, or pair of bools or numbers
        Set axis scale(s) to log. A single value sets the data axis for any numeric
        axes in the plot. A pair of values sets each axis independently.
        Numeric values are interpreted as the desired base (default 10).
        When `None` or `False`, seaborn defers to the existing Axes scale.
        .. versionadded:: v0.13.0\
    """),
    native_scale=dedent("""\
    native_scale : bool
        When True, numeric or datetime values on the categorical axis will maintain
        their original scaling rather than being converted to fixed indices.
        .. versionadded:: v0.13.0\
    """),
    formatter=dedent("""\
```

```
    formatter : callable
        Function for converting categorical data into strings. Affects both grouping
        and tick labels.
        .. versionadded:: v0.13.0\
    """),
    legend=dedent("""\
    legend : "auto", "brief", "full", or False
        How to draw the legend. If "brief", numeric `hue` and `size`
        variables will be represented with a sample of evenly spaced values.
        If "full", every group will get an entry in the legend. If "auto",
        choose between brief or full representation based on number of levels.
        If `False`, no legend data is added and no legend is drawn.
        .. versionadded:: v0.13.0\
    """),

# Code from: seaborn/_core/plot.py
"variables": variables,
            "structure": structure,
            "wrap": wrap,
        }
        new = self._clone()
        new._facet_spec.update(spec)
        return new
    # TODO def twin()?
    def scale(self, **scales: Scale) -> Plot:
        """
        Specify mappings from data units to visual properties.
        Keywords correspond to variables defined in the plot, including coordinate
        variables (`x`, `y`) and semantic variables (`color`, `pointsize`, etc.).
        A number of "magic" arguments are accepted, including:
            - The name of a transform (e.g., `"log"`, `"sqrt"`)
            - The name of a palette (e.g., `"viridis"`, `"muted"`)
            - A tuple of values, defining the output range (e.g. `(1, 5)`)
            - A dict, implying a :class:`Nominal` scale (e.g. `{"a": .2, "b": .5}`)
            - A list of values, implying a :class:`Nominal` scale (e.g. `["b", "r"]`)
        For more explicit control, pass a scale spec object such as :class:`Continuous`
        or :class:`Nominal`. Or pass `None` to use an "identity" scale, which treats
        data values as literally encoding visual properties.
        Examples
        --------
        .. include:: ../docstrings/objects.Plot.scale.rst
        """
        new = self._clone()
        new._scales.update(scales)
        return new
    def share(self, **shares: bool | str) -> Plot:
        """
        Control sharing of axis limits and ticks across subplots.
        Keywords correspond to variables defined in the plot, and values can be
        boolean (to share across all subplots), or one of "row" or "col" (to share
        more selectively across one dimension of a grid).
        Behavior for non-coordinate variables is currently undefined.
        Examples
        --------
        .. include:: ../docstrings/objects.Plot.share.rst
        """
        new = self._clone()
        new._shares.update(shares)
        return new
    def limit(self, **limits: tuple[Any, Any]) -> Plot:
        """
        Control the range of visible data.
        Keywords correspond to variables defined in the plot, and values are a
        `(min, max)` tuple (where either can be `None` to leave unset).
        Limits apply only to the axis; data outside the visible range are
        still used for any stat transforms and added to the plot.
        """
```

F.2 RepoMap context

The RepoMap context is presented below. Due to its considerable length, we only present a limited portion of the RepoMap context.

```
# We provide you with structures of files that are imported by this target file, which only include their
↪  structure names such as global variable, class and function names, and their code implementations are
↪  omitted.
# These structures can help you understand the overall structure of imported files, and the relationships
↪  between the target file and its dependencies.
# For each imported file, we provide you with its file name, followed by its structure.

# seaborn/_core/typing.py
ColumnName
Vector
VariableSpec
VariableSpecList
```

```
DataSource
OrderSpec
NormSpec
PaletteSpec
DiscreteValueSpec
ContinuousValueSpec
class Default:
    def __repr__(self):
class Deprecated:
    def __repr__(self):
default
deprecated

# seaborn/_core/rules.py
class VarType(UserString):
    allowed
    def __init__(self, data):
    def __eq__(self, other):
def variable_type(
    vector: Series,
    boolean_type: Literal["numeric", "categorical", "boolean"] = "numeric",
    strict_boolean: bool = False,
) -> VarType:
    def all_numeric(x):
    def all_datetime(x):
def categorical_order(vector: Series, order: list | None = None) -> list:

# seaborn/_core/plot.py
default
class Layer(TypedDict, total=False):
    mark: Mark
    stat: Stat | None
    move: Move | list[Move] | None
    data: PlotData
    source: DataSource
    vars: dict[str, VariableSpec]
    orient: str
    legend: bool
    label: str | None
class FacetSpec(TypedDict, total=False):
    variables: dict[str, VariableSpec]
    structure: dict[str, list[str]]
    wrap: int | None
class PairSpec(TypedDict, total=False):
    variables: dict[str, VariableSpec]
    structure: dict[str, list[str]]
    cross: bool
    wrap: int | None
def theme_context(params: dict[str, Any]) -> Generator:
def build_plot_signature(cls):
...

# seaborn/_core/properties.py
RGBTuple
RGBATuple
ColorSpec
DashPattern
DashPatternWithOffset
MarkerPattern
Mapping
class Property:
    legend
    normed
    def __init__(self, variable: str | None = None):
    def default_scale(self, data: Series) -> Scale:
    def infer_scale(self, arg: Any, data: Series) -> Scale:
    def get_mapping(self, scale: Scale, data: Series) -> Mapping:
        def identity(x):
    def standardize(self, val: Any) -> Any:
    def _check_dict_entries(self, levels: list, values: dict) -> None:
    def _check_list_length(self, levels: list, values: list) -> list:
class Coordinate(Property):
    legend
    normed
...
```

F.3 Retriever agent contexts

The cross-file contexts retrieved by the retriever agent are shown below. There are a total of 12 context blocks, but only the initial three are displayed, and the remaining blocks are omitted due to their substantial length.

```
An intelligent agent analyzed this repository and extracted the following relevant context for completing
↪  the target function.
```

```
The agent explored the codebase and identified these contexts as potentially useful based on patterns,
↪  dependencies, and contextual relevance.

Context 1: Looking at the base Scale class label method - this is the parent method that our
↪  implementation should be similar to
Type: code_snippet
The `label()` method in the Scale base class is a compatibility method currently implemented as a no-op:

```python
def label(self):
    # This is a no-op method for compatibility
    new = copy(self)
    new._label_params = {}
    return new
```

This suggests that our implementation should follow this pattern of returning a copy of the scale with
↪ updated parameters.

Context 2: Understanding how parameters are passed from label() method to _get_formatter()
Type: code_snippet
The Scale class's _get_scale method shows how label parameters are used:

```python
def _get_scale(self, name: str, forward: Callable, inverse: Callable):
    major_locator, minor_locator = self._get_locators(**self._tick_params)
    major_formatter = self._get_formatter(major_locator, **self._label_params)

    class InternalScale(mpl.scale.FuncScale):
        def set_default_locators_and_formatters(self, axis):
            axis.set_major_locator(major_locator)
            if minor_locator is not None:
                axis.set_minor_locator(minor_locator)
            axis.set_major_formatter(major_formatter)

    return InternalScale(name, (forward, inverse))
```

This shows that `self._label_params` are passed as kwargs to the _get_formatter method.

Context 3: Examining Continuous.tick() method signature and implementation
Type: code_snippet
The `Continuous.tick()` method is highly relevant as it shows how to implement an interface for configuring
↪ ticks:

```python
def tick(
    self,
    locator: Locator | None = None, *,
    at: Sequence[float] | None = None,
    upto: int | None = None,
    count: int | None = None,
    every: float | None = None,
    between: tuple[float, float] | None = None,
    minor: int | None = None,
) -> Continuous:
    """
    Configure the selection of ticks for the scale's axis or legend.

    Parameters
    ----------
    locator : :class:`matplotlib.ticker.Locator` subclass
        Pre-configured matplotlib locator; other parameters will not be used.
    at : sequence of floats
        Place ticks at these specific locations (in data units).
    upto : int
        Choose "nice" locations for ticks, but do not exceed this number.
    count : int
        Choose exactly this number of ticks, bounded by `between` or axis limits.
    every : float
        Choose locations at this interval of separation (in data units).
    between : pair of floats
        Bound upper / lower ticks when using `every` or `count`.
    minor : int
        Number of unlabeled ticks to draw between labeled "major" ticks.

    Returns
    -------
    scale
        Copy of self with new tick configuration.

    """
    # Input checks
    if locator is not None and not isinstance(locator, Locator):
        raise TypeError(
            f"Tick locator must be an instance of {Locator!r}, "
            f"not {type(locator)!r}."
```
```

```
        )
    log_base, symlog_thresh = self._parse_for_log_params(self.trans)
    if log_base or symlog_thresh:
        if count is not None and between is None:
            raise RuntimeError("`count` requires `between` with log transform.")
        if every is not None:
            raise RuntimeError("`every` not supported with log transform.")

    new = copy(self)
    new._tick_params = {
        "locator": locator,
        "at": at,
        "upto": upto,
        "count": count,
        "every": every,
        "between": between,
        "minor": minor,
    }
    return new
```

This pattern follows the same approach - validate parameters, create a copy of self, update parameter
↪ dictionary, and return new instance.
```

## F.4 Forecaster agent

The cross-file contexts retrieved by the forecaster agent are shown below. The prediction of the forecaster agent is incorrect for this sample; the target function is `label`, but the agent predicts `_default_spacer`. We only show the initial three context blocks, and the remaining blocks are omitted due to their substantial length.

```
An intelligent agent analyzed this repository and extracted the following relevant context for completing
↪ the target function.
The agent explored the codebase and identified these contexts as potentially useful based on patterns,
↪ dependencies, and contextual relevance.

Context 1: This implementation provides a simple function that returns a default spacing value of 1 for
↪ any input series. This is consistent with the Scale._spacing method's fallback behavior when there's
↪ no variance in the data.
Source: predicted_function
Type: predicted_implementation
```python
def _default_spacer(x: Series) -> float:
    """
    Return a default spacing value of 1 for any input.

    This function is used as a default spacing function for scales where
    specific spacing behavior is not needed.

    Parameters
    ----------
    x : Series
        Input data (not used in this implementation)

    Returns
    -------
    float
        Default spacing value (1.0)
    """
    return 1.0
```

Context 2: This implementation calculates spacing based on the range of the data divided by a standard
↪ number of bins (10). This approach provides spacing proportional to the data range while ensuring a
↪ reasonable number of divisions.
Source: predicted_function
Type: predicted_implementation
```python
def _default_spacer(x: Series) -> float:
    """
    Calculate a default spacing value based on the data range.

    Returns the range of the data divided by 10 as a reasonable default spacing.
    If the data contains fewer than 2 unique values or is all NaN, returns 1.0.

    Parameters
    ----------
    x : Series
        Input data series

    Returns
    -------
    float
```

```
        Calculated spacing value
    """
    clean_x = x.dropna()
    if len(clean_x) < 2:
        return 1.0

    x_min, x_max = clean_x.min(), clean_x.max()
    if x_min == x_max:
        return 1.0

    return (x_max - x_min) / 10
```

```
Context 3: This implementation is similar to how the spacing is handled in the existing Continuous scale
↪  class, using the minimum difference between sorted values to determine natural spacing.
Source: predicted_function
Type: predicted_implementation
```python
def _default_spacer(x: Series) -> float:
    """
    Calculate spacing based on the minimum difference between adjacent values.

    Finds the minimum difference between adjacent values in the sorted data,
    which is useful for determining appropriate bar widths or point spacing.

    Parameters
    ----------
    x : Series
        Input data series

    Returns
    -------
    float
        Minimum difference between adjacent values or 1.0 if undetermined
    """
    clean_x = x.dropna().unique()
    if len(clean_x) < 2:
        return 1.0

    sorted_x = np.sort(clean_x)
    min_diff = np.min(np.diff(sorted_x))

    # Guard against zero spacing
    return max(min_diff, 1e-8) if min_diff > 0 else 1.0
```
```

F.5 SpecAgent contexts

The cross-file contexts retrieved by the SpecAgent are shown below. Interestingly, the SpecAgent's prediction for the target function is incorrect in this example; it predicts `_make_log_transforms`, but the real target function is actually `Continuous.scale`. We only show the initial three context blocks, and the remaining blocks are omitted due to their substantial length.

```
An intelligent agent analyzed this repository and extracted the following relevant context for completing
↪  the target function.
The agent explored the codebase and identified these contexts as potentially useful based on patterns,
↪  dependencies, and contextual relevance.

Context 1: The most likely missing function is `_make_log_transforms`. This function creates forward and
↪  inverse transform functions for logarithmic scaling. Looking at the imports and existing transforms,
↪  here's the implementation:
Type: code_snippet
```python
def _make_log_transforms(base: float | None = None) -> TransFuncs:

    fs: TransFuncs
    if base is None:
        fs = np.log, np.exp
    elif base == 2:
        fs = np.log2, partial(np.power, 2)
    elif base == 10:
        fs = np.log10, partial(np.power, 10)
    else:
        def forward(x):
            return np.log(x) / np.log(base)
        fs = forward, partial(np.power, base)

    def log(x: ArrayLike) -> ArrayLike:
        with np.errstate(invalid="ignore", divide="ignore"):
            return fs[0](x)

    def exp(x: ArrayLike) -> ArrayLike:
        with np.errstate(invalid="ignore", divide="ignore"):
```
```

````
            return fs[1](x)

    return log, exp
```

This function is used extensively throughout the codebase for logarithmic transformations with different
↪  bases.

Context 2: Function usage in `ContinuousBase._get_transform` method in `seaborn/_core/scales.py`:
Source: seaborn/_core/scales.py
Type: code_snippet
The `_make_log_transforms` function is used in multiple places in the `_get_transform` method. Here are the
↪  relevant parts:

```python
def _get_transform(self):
    arg = self.trans
    def get_param(method, default):
        if arg == method:
            return default
        return float(arg[len(method):])

    if arg is None:
        return _make_identity_transforms()
    elif isinstance(arg, tuple):
        return arg
    elif isinstance(arg, str):
        if arg == "ln":
            return _make_log_transforms()
        elif arg == "logit":
            base = get_param("logit", 10)
            return _make_logit_transforms(base)
        elif arg.startswith("log"):
            base = get_param("log", 10)
            return _make_log_transforms(base)
        # Other transform options...
```

This shows that `_make_log_transforms` is called when the transformation type is "ln" or when it starts
↪  with "log". The function supports both natural logarithm and logarithms with specific bases.

Context 3: Usage in other transformation functions in `seaborn/_core/scales.py`:
Source: seaborn/_core/scales.py
Type: code_snippet
The `_make_log_transforms` function is also used within other transform functions:

```python
def _make_logit_transforms(base: float | None = None) -> TransFuncs:
    log, exp = _make_log_transforms(base)

    def logit(x):
        with np.errstate(invalid="ignore", divide="ignore"):
            return log(x) - log(1 - x)

    def expit(x):
        with np.errstate(invalid="ignore", divide="ignore"):
            return exp(x) / (1 + exp(x))

    return logit, expit

def _make_symlog_transforms(c: float = 1, base: float = 10) -> TransFuncs:
    # From https://iopscience.iop.org/article/10.1088/0957-0233/24/2/027001
    log, exp = _make_log_transforms(base)

    def symlog(x):
        with np.errstate(invalid="ignore", divide="ignore"):
            return np.sign(x) * log(1 + np.abs(np.divide(x, c)))

    def symexp(x):
        with np.errstate(invalid="ignore", divide="ignore"):
            return np.sign(x) * c * (exp(np.abs(x)) - 1)

    return symlog, symexp
```

This shows that `_make_log_transforms` serves as a building block for more complex transformations.
````

F.6 Oracle agent contexts

The contexts retrieved by the oracle agent are shown below. Notably, while the oracle agent has access to the ground-truth function completion, it does not provide a direct solution in the contexts. There are a total of 12 context blocks, but only the initial three are displayed, and the remaining blocks are omitted due to their substantial length.
````

```
An intelligent agent analyzed this repository and extracted the following relevant context for completing
↪  the target function.
The agent explored the codebase and identified these contexts as potentially useful based on patterns,
↪  dependencies, and contextual relevance.

Context 1: Implementation of _get_formatter method for Continuous scales that handles formatter logic
Source: seaborn/_core/scales.py
Type: code_snippet
From the file seaborn/_core/scales.py, the Continuous class has a _get_formatter method that shows how
↪  formatters are created and configured based on parameters:

```python
def _get_formatter(self, locator, formatter, like, base, unit):
    log_base, symlog_thresh = self._parse_for_log_params(self.trans)
    if base is default:
        if symlog_thresh:
            log_base = 10
        base = log_base

    if formatter is not None:
        return formatter

    if like is not None:
        if isinstance(like, str):
            if "{x" in like or "{pos" in like:
                fmt = like
            else:
                fmt = f"{{x:{like}}}"
            formatter = StrMethodFormatter(fmt)
        else:
            formatter = FuncFormatter(like)

    elif base is not None:
        # We could add other log options if necessary
        formatter = LogFormatterSciNotation(base)

    elif unit is not None:
        if isinstance(unit, tuple):
            sep, unit = unit
        elif not unit:
            sep = ""
        else:
            sep = " "
        formatter = EngFormatter(unit, sep=sep)

    else:
        formatter = ScalarFormatter()

    return formatter
```

Context 2: Temporal class label method showing similar pattern
Source: seaborn/_core/scales.py
Type: code_snippet
The Temporal class's label method in seaborn/_core/scales.py shows a similar pattern for configuring tick
↪ label formatters:

```python
def label(
    self,
    formatter: Formatter | None = None, *,
    concise: bool = False,
) -> Temporal:
    """
    Configure the appearance of tick labels for the scale's axis or legend.

    .. note::
        This API is under construction and will be enhanced over time.

    Parameters
    ----------
    formatter : :class:`matplotlib.ticker.Formatter` subclass
        Pre-configured formatter to use; other parameters will be ignored.
    concise : bool
        If True, use :class:`matplotlib.dates.ConciseDateFormatter` to make
        the tick labels as compact as possible.

    Returns
    -------
    scale
        Copy of self with new label configuration.

    """
    new = copy(self)
    new._label_params = {"formatter": formatter, "concise": concise}
    return new
```
```

````
```

Context 3: Boolean class label method implementation
Source: seaborn/_core/scales.py
Type: code_snippet
From seaborn/_core/scales.py, the Boolean class shows a simpler implementation of the label method:

```python
def label(self, formatter: Formatter | None = None):
    new = copy(self)
    new._label_params = {"formatter": formatter}
    return new
```
````
````

G Retrieval methods cross-comparisons

In this section, we compare traditional retrieval methods commonly used in production systems (BM25 and RepoMap) with the oracle agent described in Section 4.2. Our analysis focuses on cases where one method leads to a correct function completion (i.e., passes unit tests) while another fails. In particular, we investigate whether these discrepancies stem from missing or insufficient cross-file context—cases we term *context decisive*.

To make this determination, we use an LLM judge to assess why one method succeeds while the other does not, and whether the difference can be attributed to context decisiveness. From these results, we derive insights into why BM25 and RepoMap underperform compared to the oracle agent, which in turn inform the design of our indexing-time agents to mitigate these limitations. We also provide representative examples for each insight. For clarity, the "oracle agent" referenced in this section denotes an earlier version of the agent, which is less performant than the one reported in the main text.

G.1 Evaluation setup

For each REPOCOD sample in which one cross-file retrieval method succeeds while the other fails, we ask an LLM judge to analyze the cause of this discrepancy. Specifically, the judge assesses (i) the likely reason for the outcome (e.g., context quality, missing imports, wrong API usage, or logic errors), (ii) whether the difference is *context decisive*, and (iii) whether the retrieved contexts were helpful for function completion. We employ Claude 3.7 Sonnet (Anthropic, 2025) as the judge, and provide it with a structured prompt. The exact prompt format is shown below.

````
You are an expert software engineer analyzing why two different code completion methods produced different
↪  results on the same task.

**Context:**
- Repository: {repo_name}
- Function: {function_name}
- One method succeeded (passed unit tests) while the other failed
- Success method: {success_method.upper()}

**Task Information:**

**1. Original prompt given to both models:**
```
{prompt}
```

**2. Target file content (for context):**
```python
{target_file_content[:2000]}{"..." if len(target_file_content) > 2000 else ""}
```

**3. Ground truth (correct implementation):**
```python
{ground_truth}
```

**4. BM25 Method Output:**
```python
{bm25_output}
```

**5. Oracle Agent Method Output:**
```python
{oracle_output}
```

**6. Cross-file contexts used by each method:**

{bm25_context_str}

{oracle_context_str}

**Analysis Task:**
Analyze why the {success_method.upper()} method succeeded while the other failed. Focus on:

1. **Context Quality**: How did the different cross-file contexts influence the results?
2. **Missing Information**: What critical information was missing in the failing method's context?
3. **Context Relevance**: Which context was more relevant to solving the task?
4. **Specific Differences**: What specific differences in the outputs led to success/failure?

**Response Format:**
Provide a JSON response with:
````

```
{{
    "success_method": "{success_method}",
    "primary_difference_reason": "context_quality|missing_imports|wrong_api_usage|logic_error|other",
    "context_was_decisive": true/false,
    "bm25_context_assessment": "helpful|partially_helpful|unhelpful|missing",
    "oracle_context_assessment": "helpful|partially_helpful|unhelpful|missing",
    "detailed_analysis": "Detailed explanation of why one method succeeded",
    "critical_missing_context": "What context was missing in the failed method",
    "context_quality_comparison": "Compare the quality and relevance of both contexts",
    "confidence": "high|medium|low"
}}

Only respond with the JSON, no additional text, no markdown code blocks.
```

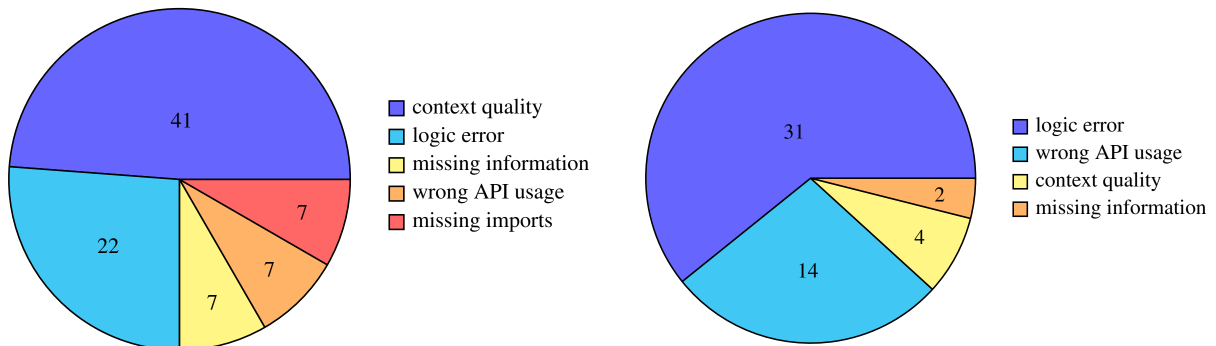
G.2 BM25 versus oracle agent

G.2.1 Summary

The LLM judge analysis reveals clear differences between BM25 and the oracle agent:

- The oracle agent achieves **98.8% context decisiveness**, compared to BM25's **25.5%**, indicating that oracle's successes are far more often driven by relevant cross-file context.
- Among all samples analyzed, the oracle agent provides **helpful context in 100% of cases**, whereas BM25 does so in only **21.6% of cases**.

A breakdown of the reasons behind each method's successful completions is shown in Figure 4.



(a) Reasons for oracle agent successes. Context quality is the dominant factor.

(b) Reasons for BM25 successes. Logic errors are most common.

Figure 4: Breakdown of success factors for BM25 and the oracle agent. Oracle successes are usually context-driven, while BM25's are often due to issues unrelated to context quality.

As shown in Figure 4, the oracle agent's advantage primarily stems from its ability to retrieve high-quality cross-file context, which directly enables correct function completions. In contrast, BM25's rare successes tend to arise from logic-related issues where context quality plays little to no role. These results underscore that context retrieval is the decisive factor behind the oracle agent's superior performance.

In the following subsection, we present detailed case studies to better understand the common failure modes of BM25 and why it often fails to surface the most relevant context.

G.2.2 Insights

In this subsection, we present a set of detailed insights into why BM25 often fails to retrieve the most helpful contexts, while the oracle agent avoids these pitfalls. For each insight, we provide concrete examples drawn from our experiments. These examples demonstrate how the oracle agent's access to richer and semantically aligned information enables more accurate code completion, while BM25's reliance on sparse lexical similarity leads to systematic blind spots. Together, these insights highlight the advantages of using an intelligent retrieval agent over traditional methods based purely on token similarity.

1. **Insight**: The ground-truth implementation may invoke other functions in the repository that are semantically important but lexically unrelated. Since BM25 relies on surface similarity in terms of

docstrings and function signatures, it frequently fails to retrieve these functions. The oracle agent, by contrast, has access to the target function body and therefore prioritizes the contexts of functions that are directly invoked.

Example: In repository ID 67 from the seaborn repository, the target function is `tick` in the `Continuous` class. The function internally calls `_parse_for_log_params`. The oracle agent retrieves the definition and usages of this helper function, allowing the completion model to include the correct call in its generated output:

```
def tick(
    self,
    locator: Locator | None = None, *,
    at: Sequence[float] | None = None,
    upto: int | None = None,
    count: int | None = None,
    every: float | None = None,
    between: tuple[float, float] | None = None,
    minor: int | None = None,
) -> Continuous:
    if locator is not None and not isinstance(locator, Locator):
        err = (
            f"Tick locator must be an instance of {Locator!r}, "
            f"not {type(locator)!r}."
        )
        raise TypeError(err)

    log_base, symlog_thresh = self._parse_for_log_params(self.trans)

    if log_base or symlog_thresh:
        if count is not None and between is None:
            raise RuntimeError("`count` requires `between` with log transform.")
        if every is not None:
            raise RuntimeError("`every` not supported with log transform.")

    new = copy(self)
    new._tick_params = {
        "locator": locator,
        "at": at,
        "upto": upto,
        "count": count,
        "every": every,
        "between": between,
        "minor": minor,
    }
    return new
```

Contexts related to this helper function are included in the following two blocks in the oracle agent's retrieved contexts:

````
**Explanation**: Log parameter detection in `Continuous._parse_for_log_params` method

From `seaborn/_core/scales.py`, the `Continuous._get_locators` method shows the logic for selecting
↪  different locators based on parameters:

```python
def _get_locators(self, locator, at, upto, count, every, between, minor):
    log_base, symlog_thresh = self._parse_for_log_params(self.trans)

    if locator is not None:
        major_locator = locator

    elif upto is not None:
        if log_base:
            major_locator = LogLocator(base=log_base, numticks=upto)
        else:
            major_locator = MaxNLocator(upto, steps=[1, 1.5, 2, 2.5, 3, 5, 10])

    elif count is not None:
        if between is None:
            # This is rarely useful (unless you are setting limits)
            major_locator = LinearLocator(count)
        else:
            if log_base or symlog_thresh:
                forward, inverse = self._get_transform()
                lo, hi = forward(between)
                ticks = inverse(np.linspace(lo, hi, num=count))
            else:
                ticks = np.linspace(*between, num=count)
            major_locator = FixedLocator(ticks)

    elif every is not None:
        if between is None:
````
````

```
            major_locator = MultipleLocator(every)
        else:
            lo, hi = between
            ticks = np.arange(lo, hi + every, every)
            major_locator = FixedLocator(ticks)

    elif at is not None:
        major_locator = FixedLocator(at)

    else:
        if log_base:
            major_locator = LogLocator(log_base)
        elif symlog_thresh:
            major_locator = SymmetricalLogLocator(linthresh=symlog_thresh, base=10)
        else:
            major_locator = AutoLocator()

    if minor is None:
        minor_locator = LogLocator(log_base, subs=None) if log_base else None
    else:
        if log_base:
            subs = np.linspace(0, log_base, minor + 2)[1:-1]
            minor_locator = LogLocator(log_base, subs=subs)
        else:
            minor_locator = AutoMinorLocator(minor + 1)

    return major_locator, minor_locator
```

Explanation: Log parameter detection in `Continuous._parse_for_log_params` method

From `seaborn/_core/scales.py`, the `_parse_for_log_params` method shows how to detect log and symlog
↪ parameters:

```python
def _parse_for_log_params(
    self, trans: str | TransFuncs | None
) -> tuple[float | None, float | None]:

    log_base = symlog_thresh = None
    if isinstance(trans, str):
        m = re.match(r"^log(\d*)", trans)
        if m is not None:
            log_base = float(m[1] or 10)
        m = re.match(r"symlog(\d*)", trans)
        if m is not None:
            symlog_thresh = float(m[1] or 1)
    return log_base, symlog_thresh
```

With this contextual support, the generated code passes the unit tests. By contrast, BM25 retrieves contexts unrelated to `_parse_for_log_params`, as there is no lexical similarity between its name or docstring and the target function. Consequently, the model fails to call this helper function, and the resulting completion is incorrect. This example illustrates a structural weakness of BM25: it cannot capture dependency chains that are critical for correctness.

2. **Insight**: BM25 often overlooks repository-wide conventions that shape function behavior. Such conventions include default fallback values, strategies for handling missing inputs, and typical patterns of error handling. Since these conventions are rarely reflected in surface-level lexical overlap, BM25 is blind to them. The oracle agent, however, surfaces conventions by retrieving semantically related examples and type information.

   Example: In repository ID 15 from more-itertools, the target function is `divide`, which should return a list of iterators. The target's signature provides no type information. BM25 misses the type hints in `more.pyi` specifying the return type as `list[Iterator[_T]]`, and it fails to bring in related functions such as `distribute()` that demonstrate the correct use of iterators. The oracle agent, by contrast, retrieves the type stubs, similar functions such as `chunked_even`, and examples of using `divmod` for element partitioning:

```
**Explanation**: The `chunked_even` function shows dividing an iterable into evenly sized chunks

From `more_itertools/more_itertools/more.py`:
```python
def chunked_even(iterable, n):
    """Break *iterable* into lists of approximately length *n*.
    Items are distributed such the lengths of the lists differ by at most
```
```

```
    1 item.

    >>> iterable = [1, 2, 3, 4, 5, 6, 7]
    >>> n = 3
    >>> list(chunked_even(iterable, n))  # List lengths: 3, 2, 2
    [[1, 2, 3], [4, 5], [6, 7]]
    >>> list(chunked(iterable, n))  # List lengths: 3, 3, 1
    [[1, 2, 3], [4, 5, 6], [7]]

    """
    iterable = iter(iterable)

    # Initialize a buffer to process the chunks while keeping
    # some back to fill any underfilled chunks
    min_buffer = (n - 1) * (n - 2)
    buffer = list(islice(iterable, min_buffer))

    # Append items until we have a completed chunk
    for _ in islice(map(buffer.append, iterable), n, None, n):
        yield buffer[:n]
        del buffer[:n]

    # Check if any chunks need addition processing
    if not buffer:
        return
    length = len(buffer)

    # Chunks are either size `full_size <= n` or `partial_size = full_size - 1`
    q, r = divmod(length, n)
    num_lists = q + (1 if r > 0 else 0)
    q, r = divmod(length, num_lists)
    full_size = q + (1 if r > 0 else 0)
    partial_size = full_size - 1
    num_full = length - partial_size * num_lists

    # Yield chunks of full size
    partial_start_idx = num_full * full_size
    if full_size > 0:
        for i in range(0, partial_start_idx, full_size):
            yield buffer[i : i + full_size]

    # Yield chunks of partial size
    if partial_size > 0:
        for i in range(partial_start_idx, length, partial_size):
            yield buffer[i : i + partial_size]
```

Explanation: The function type signature for `divide` from `more.pyi` (based on found imports)

From `more_itertools/more_itertools/more.pyi`:
```python
def divide(n: int, iterable: Iterable[_T]) -> list[Iterator[_T]]: ...
```
```

Equipped with these contexts, the model produces a correct implementation that returns iterators rather than lists. Without them, BM25's contexts bias the model toward incorrect type usage. This example underscores the role of semantic retrieval in preserving repository-specific coding conventions.

3. **Insight**: Many repositories exhibit "multi-hop context" dependencies, where the correct implementation requires knowledge not only of the immediate class APIs but also of the APIs of member objects within that class. BM25 generally fails to capture these multi-hop relationships, as they involve broader structural reasoning across classes. The oracle agent, by design, retrieves relevant cross-references and usage patterns.

   **Example**: In repository ID 26 from seaborn, the target function is `refline` in the `JointGrid` class. The code completion model with access to the oracle agent's contexts generates the following correct function implementation:

```
def refline(
    self, *, x=None, y=None, joint=True, marginal=True,
    color='.5', linestyle='--', **line_kws
):
    """Add a reference line(s) to joint and/or marginal axes.

    Parameters
    ----
    x, y : numeric
        Value(s) to draw the line(s) at.
```
```

```
    joint, marginal : bools
        Whether to add the reference line(s) to the joint/marginal axes.
    color : :mod:`matplotlib color <matplotlib.colors>`
        Specifies the color of the reference line(s).
    linestyle : str
        Specifies the style of the reference line(s).
    line_kws : key, value mappings
        Other keyword arguments are passed to :meth:`matplotlib.axes.Axes.axvline`
        when ``x`` is not None and :meth:`matplotlib.axes.Axes.axhline` when ``y``
        is not None.

    Returns
    -------
    :class:`JointGrid` instance
        Returns ``self`` for easy method chaining.

    """
    line_kws['color'] = color
    line_kws['linestyle'] = linestyle

    if x is not None and joint:
        self.ax_joint.axvline(x=x, **line_kws)
    if y is not None and joint:
        self.ax_joint.axhline(y=y, **line_kws)
    if x is not None and marginal:
        self.ax_marg_x.axvline(x=x, **line_kws)
    if y is not None and marginal:
        self.ax_marg_y.axhline(y=y, **line_kws)

    return self
```

Clearly, to generate a correct completion, the model must understand the roles of the `ax_marg_x` and `ax_marg_y` attributes, which are matplotlib axes objects. BM25 fails to retrieve detailed contexts about the structure of `JointGrid` or how its axes are used. The oracle agent, however, surfaces examples of how reference lines are added to axes, how the `JointGrid` object manages its subplots, and how marginal axes are manipulated:

````
**Explanation**: `JointGrid` class property that gives access to the axes

From the code in the target file, we can see how `JointGrid`'s axes are structured:

```python
class JointGrid(_BaseGrid):
    # ...
    def __init__(
        self, data=None, *,
        x=None, y=None, hue=None,
        height=6, ratio=5, space=.2,
        palette=None, hue_order=None, hue_norm=None,
        dropna=False, xlim=None, ylim=None, marginal_ticks=False,
    ):
        # ...
        # Set up the subplot grid
        f = plt.figure(figsize=(height, height))
        gs = plt.GridSpec(ratio + 1, ratio + 1)

        ax_joint = f.add_subplot(gs[1:, :-1])
        ax_marg_x = f.add_subplot(gs[0, :-1], sharex=ax_joint)
        ax_marg_y = f.add_subplot(gs[1:, -1], sharey=ax_joint)

        self._figure = f
        self.ax_joint = ax_joint
        self.ax_marg_x = ax_marg_x
        self.ax_marg_y = ax_marg_y
```
````

With this information, the completion model successfully generates the correct function. This example illustrates how semantic retrieval captures higher-order class dependencies that token-based similarity cannot.

4. **Insight**: BM25 may provide partial coverage of API calls but often fails to retrieve contexts showing how different functions interact together in realistic workflows. The oracle agent excels at recovering these co-usage patterns, which are crucial for correctness.

   Example: In repository ID 28 from more-itertools, the target function is `locate`. Its implementation requires the combined use of `compress` and `count`:

```python
def locate(iterable, pred=bool, window_size=None):
    if window_size is None:
        try:
            return compress(count(), map(pred, iterable))
        except TypeError:
            pass
    if window_size < 1:
        raise ValueError('window_size must be at least 1')
    it = iter(iterable)
    windows = windowed(it, window_size)
    return compress(count(), map(lambda w: pred(*w), windows))
```

BM25 retrieves fragments showing these functions individually but not in combination. As a result, the model lacks examples demonstrating their joint usage. By contrast, the oracle agent retrieves definitions, test cases, and helper functions showing how these two functions interact:

````
**Explanation**: The pattern for combining `itertools.compress` with `count()` to generate indexes is
↪  seen in other functions

From `more_itertools/more_itertools/recipes.py`:
```python
def iter_index(iterable, value, start=0, stop=None):
    """Yield the index of each place in *iterable* that *value* occurs,
    beginning with index *start* and ending before index *stop*.


    >>> list(iter_index('AABCADEAF', 'A'))
    [0, 1, 4, 7]
    >>> list(iter_index('AABCADEAF', 'A', 1))  # start index is inclusive
    [1, 4, 7]
    >>> list(iter_index('AABCADEAF', 'A', 1, 7))  # stop index is not inclusive
    [1, 4]

    The behavior for non-scalar *values* matches the built-in Python types.

    >>> list(iter_index('ABCDABCD', 'AB'))
    [0, 4]
    >>> list(iter_index([0, 1, 2, 3, 0, 1, 2, 3], [0, 1]))
    []
    >>> list(iter_index([[0, 1], [2, 3], [0, 1], [2, 3]], [0, 1]))
    [0, 2]

    See :func:`locate` for a more general means of finding the indexes
    associated with particular values.

    """
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        # Slow path for general iterables
        it = islice(iterable, start, stop)
        for i, element in enumerate(it, start):
            if element is value or element == value:
                yield i
    else:
        # Fast path for sequences
        stop = len(iterable) if stop is None else stop
        i = start - 1
        try:
            while True:
                yield (i := seq_index(value, i + 1, stop))
        except ValueError:
            pass
```
````

This richer context leads to a correct implementation, while BM25's incomplete view results in failure. The lesson here is that the retrieval method must capture not only components but also how they are orchestrated together.

5. **Insight**: BM25 systematically struggles with retrieving semantically similar but lexically distinct functions. These functions may use different terminology or naming conventions, making lexical similarity an unreliable signal. The oracle agent is robust to this mismatch because it prioritizes semantic connections over token overlap.

   Example: In repository ID 6 from more-itertools, the target function is `classify_unique`, which references `unique_everseen` and `unique_justseen` in its docstring:

```python
def classify_unique(iterable, key=None):
```

```python
    seen_set = set()
    seen_list = []
    use_key = key is not None
    prev_key = None
    for element in iterable:
        k = key(element) if use_key else element
        try:
            if k not in seen_set:
                seen_set.add(k)
                is_everseen = True
            else:
                is_everseen = False
            if prev_key is None or k != prev_key:
                is_justseen = True
                prev_key = k
            else:
                is_justseen = False
            yield (element, is_justseen, is_everseen)
        except TypeError:
            if k not in seen_list:
                seen_list.append(k)
                is_everseen = True
            else:
                is_everseen = False
            if prev_key is None or k != prev_key:
                is_justseen = True
                prev_key = k
            else:
                is_justseen = False
            yield (element, is_justseen, is_everseen)
```

BM25 fails to retrieve these related functions due to low lexical similarity, leaving the model without a complete picture. The oracle agent retrieves both, along with test cases that clarify the intended behavior:

````
**Explanation**: The `unique_everseen` function from `recipes.py` would be highly relevant as the
↪  `classify_unique` function description suggests returning similar functionality.

From `more_itertools/recipes.py`:
```python
def unique_everseen(iterable, key=None):
    """
    List unique elements, preserving order. Remember all elements ever seen.

        >>> list(unique_everseen('AAAABBBCCDAABBB'))
        ['A', 'B', 'C', 'D']
        >>> list(unique_everseen('ABBCcAD', str.lower))
        ['a', 'b', 'c', 'd']

    Sequences with a mix of hashable and unhashable items can be used.
    The function will be slower (i.e., O(n^2)) for unhashable items.

    """
    seenset = set()
    seenlist = []
    use_key = key is not None
    for element in iterable:
        k = key(element) if use_key else element
        try:
            if k not in seenset:
                seenset.add(k)
                yield element
        except TypeError:
            if k not in seenlist:
                seenlist.append(k)
                yield element
```

**Explanation**: The `groupby` function from itertools is used in several places and could be
↪  relevant for detecting consecutive duplicates:

From `more_itertools/recipes.py`:
```python
def unique_justseen(iterable, key=None):
    """List unique elements, preserving order. Remember only the element just seen.

        >>> list(unique_justseen('AAAABBBCCDAABBB'))
        ['A', 'B', 'C', 'D', 'A', 'B']
        >>> list(unique_justseen('ABBCcAD', str.lower))
        ['a', 'b', 'c', 'a', 'd']

    """
    return map(next, map(itemgetter(1), groupby(iterable, key)))
```
````

With this full context, the model generates a correct function. BM25's incomplete retrieval, by contrast, leads to missing logic and failed tests. This case exemplifies how token-based retrieval misfires when functional similarity is not aligned with naming similarity.

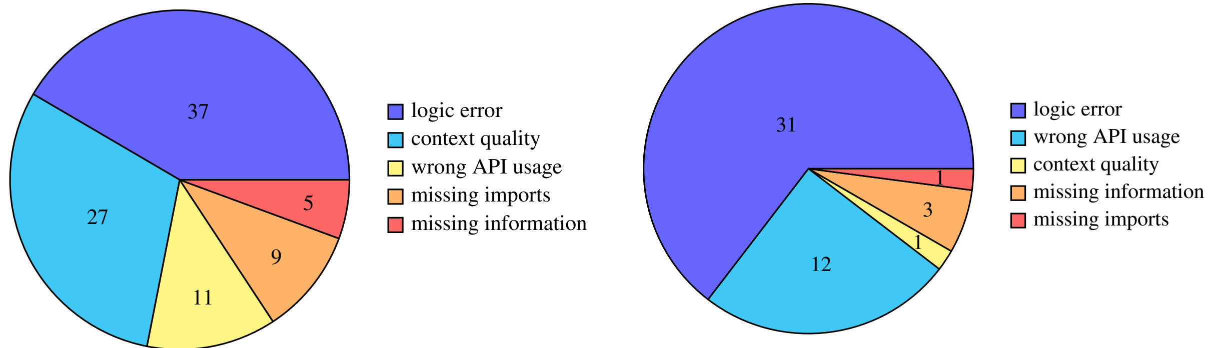
G.3 RepoMap versus oracle agent

G.3.1 Summary

The LLM-judge evaluation highlights a stark contrast between RepoMap and the oracle agent:

- The oracle agent achieves **97.8% context decisiveness**, compared to RepoMap's **41.7%**. This indicates that the oracle's completions are far more often enabled by retrieved cross-file context.
- Across all analyzed samples, the oracle agent provides **helpful context in 100% of cases**, whereas RepoMap succeeds in surfacing helpful context in only **16.7% of cases**.

The distribution of reasons behind each method's successful completions is shown in Figure 5.



(a) Oracle agent: most successes are driven by high-quality retrieved context and avoiding logic errors.

(b) RepoMap: most successes stem from logic-related issues, with almost no contribution from retrieved context.

Figure 5: Comparison of success factors for RepoMap and the oracle agent. The oracle agent's completions are typically enabled by relevant cross-file context, while RepoMap's successes are rarely context-driven.

Overall, the oracle agent's advantage lies in consistently retrieving semantically aligned, high-quality cross-file context that directly drives successful function completions. RepoMap, by contrast, succeeds mainly in cases where logic errors happen to be avoided, rather than because it surfaced helpful context. These results underscore that effective context retrieval is the decisive factor behind the oracle agent's superior performance.

In the next subsection, we provide detailed case studies illustrating RepoMap's common failure modes and explaining why it often fails to surface the most relevant context.

G.3.2 Insights

In this subsection, we provide a systematic analysis of why RepoMap often fails to retrieve the most useful contextual information, and how the oracle agent overcomes these limitations. Each insight is supported by concrete examples drawn from our experiments. These examples highlight the importance of retrieving semantically aligned, example-rich, and contextually relevant information for code completion. Whereas RepoMap primarily relies on API structures of imported files, which leads to systematic blind spots, the oracle agent provides richer evidence such as implementations, usage examples, conventions, and type annotations. Together, these findings underscore the benefits of employing an intelligent retrieval agent over traditional API-centric methods.

1. **Insight**: RepoMap includes only function signatures without examples of usage. Without usage patterns, the code completion model cannot infer how arguments should be passed or how results are handled. This gap often leads to incorrect or incomplete implementations.

   Example: In repository ID 30 from more-itertools, the target function is `make_decorator`, which inserts the result of the user function at the specified `result_index` position in the argument list passed to `wrapping_func`:

```python
def make_decorator(wrapping_func, result_index=0):
    def decorator_factory(*args, **kwargs):
        def decorator(user_func):
            def wrapper(*user_args, **user_kwargs):
                result = user_func(*user_args, **user_kwargs)
                args_list = list(args)
                args_list.insert(result_index, result)
                return wrapping_func(*args_list, **kwargs)
            return wrapper
        return decorator
    return decorator_factory
```

The RepoMap context was missing examples demonstrating how `result_index` should be applied:

```python
# more_itertools/recipes.py
__all__
_marker
_sumprod
def take(n, iterable):
def tabulate(function, start=0):
def tail(n, iterable):
def consume(iterator, n=None):
def nth(iterable, n, default=None):
def all_equal(iterable, key=None):
def quantify(iterable, pred=bool):
def pad_none(iterable):
padnone
def ncycles(iterable, n):
def dotproduct(vec1, vec2):
def flatten(listOfLists):
def repeatfunc(func, times=None, *args):
def _pairwise(iterable):
def pairwise(iterable):
class UnequalIterablesError(ValueError):
    def __init__(self, details=None):
def _zip_equal_generator(iterables):
def _zip_equal(*iterables):
def grouper(iterable, n, incomplete='fill', fillvalue=None):
def roundrobin(*iterables):
def partition(pred, iterable):
def powerset(iterable):
def unique_everseen(iterable, key=None):
def unique_justseen(iterable, key=None):
def unique(iterable, key=None, reverse=False):
def iter_except(func, exception, first=None):
def first_true(iterable, default=None, pred=None):
def random_product(*args, repeat=1):
def random_permutation(iterable, r=None):
def random_combination(iterable, r):
def random_combination_with_replacement(iterable, r):
def nth_combination(iterable, r, index):
def prepend(value, iterator):
def convolve(signal, kernel):
def before_and_after(predicate, it):
    def true_iterator():
def triplewise(iterable):
def _sliding_window_islice(iterable, n):
def _sliding_window_deque(iterable, n):
def sliding_window(iterable, n):
def subslices(iterable):
def polynomial_from_roots(roots):
def iter_index(iterable, value, start=0, stop=None):
def sieve(n):
def _batched(iterable, n, *, strict=False):
def batched(iterable, n, *, strict=False):
def transpose(it):
def reshape(matrix, cols):
def matmul(m1, m2):
def factor(n):
def polynomial_eval(coefficients, x):
def sum_of_squares(it):
def polynomial_derivative(coefficients):
def totient(n):
```

Consequently, the model's completion ignored the `result_index` argument entirely. In contrast, the oracle agent retrieved highly relevant test cases and type annotations that illustrated the expected behavior of `result_index`:

```
**Explanation**: Based on the examples, the `make_decorator` function creates a multi-level wrapper:
```

```
1. First it takes a function that processes iterables
2. Returns a function that takes arguments for that function
3. Which returns a decorator that takes a user function
4. Which returns a wrapped function that applies the iterable processor to the user function's result

```
make_decorator(wrapping_func) -> decorator_factory
decorator_factory(*args, **kwargs) -> decorator
decorator(user_function) -> wrapped_function
wrapped_function(*user_args, **user_kwargs) -> processed_result
```

Explanation: From the tests, the `result_index` parameter determines where in the arguments to
↪ `wrapping_func` the iterable (the function's result) should be placed. By default it's the first
↪ argument (position 0):

```python
slicer = mi.make_decorator(islice)  # result_index=0
```

But it can be changed:

```python
stringifier = mi.make_decorator(stringify, result_index=1)  # result_index=1
```

This means when stringifier is called, the result of the decorated function will be placed as the
↪ second argument to stringify.
```

2. **Insight**: RepoMap fails to capture repository-specific conventions. Many codebases follow implicit conventions (e.g., returning iterators instead of lists) that guide correct implementations. RepoMap's API-only approach omits this contextual knowledge.

   Example: In Example 2 of the BM25 versus oracle comparison in Section G.2.2, the target function should return an iterator rather than a list. RepoMap retrieved only bare signatures, with no hints of this convention:

```python
# more_itertools/recipes.py
__all__
_marker
_sumprod
def take(n, iterable):
def tabulate(function, start=0):
def tail(n, iterable):
def consume(iterator, n=None):
def nth(iterable, n, default=None):
def all_equal(iterable, key=None):
def quantify(iterable, pred=bool):
def pad_none(iterable):
padnone
def ncycles(iterable, n):
def dotproduct(vec1, vec2):
def flatten(listOfLists):
def repeatfunc(func, times=None, *args):
def _pairwise(iterable):
def pairwise(iterable):
class UnequalIterablesError(ValueError):
    def __init__(self, details=None):
def _zip_equal_generator(iterables):
def _zip_equal(*iterables):
def grouper(iterable, n, incomplete='fill', fillvalue=None):
def roundrobin(*iterables):
def partition(pred, iterable):
def powerset(iterable):
def unique_everseen(iterable, key=None):
def unique_justseen(iterable, key=None):
def unique(iterable, key=None, reverse=False):
def iter_except(func, exception, first=None):
def first_true(iterable, default=None, pred=None):
def random_product(*args, repeat=1):
def random_permutation(iterable, r=None):
def random_combination(iterable, r):
def random_combination_with_replacement(iterable, r):
def nth_combination(iterable, r, index):
def prepend(value, iterator):
def convolve(signal, kernel):
def before_and_after(predicate, it):
    def true_iterator():
def triplewise(iterable):
def _sliding_window_islice(iterable, n):
def _sliding_window_deque(iterable, n):
def sliding_window(iterable, n):
def subslices(iterable):
```

```python
def polynomial_from_roots(roots):
def iter_index(iterable, value, start=0, stop=None):
def sieve(n):
def _batched(iterable, n, *, strict=False):
def batched(iterable, n, *, strict=False):
def transpose(it):
def reshape(matrix, cols):
def matmul(m1, m2):
def factor(n):
def polynomial_eval(coefficients, x):
def sum_of_squares(it):
def polynomial_derivative(coefficients):
def totient(n):
```

By contrast, the oracle agent retrieved contexts containing type annotations and similar functions adhering to this convention, enabling the model to generate the correct completion.

3. **Insight**: RepoMap does not expose error-handling patterns. Without examples of defensive programming, the model tends to omit necessary checks.

   Example: In the previous example, the target function required handling empty input iterators. RepoMap's retrieved API structures contained no such examples, leading the model to miss critical error handling code:

```python
length = len(iterable)
if length == 0:
    return [iter([]) for _ in range(n)]
```

In contrast, the oracle agent retrieved functions that explicitly implemented error checks, guiding the model to produce a robust completion:

````
**Explanation**: The `split_before` function shows how to split an iterable into lists based on a
↪ predicate

From `more_itertools/more_itertools/more.py`:
```python
def split_before(iterable, pred, maxsplit=-1):
    """Yield lists of items from *iterable*, where each list ends just before
    an item for which callable *pred* returns ``True``:

        >>> list(split_before('OneTwo', lambda s: s.isupper()))
        [['O', 'n', 'e'], ['T', 'w', 'o']]

        >>> list(split_before(range(10), lambda n: n % 3 == 0))
        [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]

    At most *maxsplit* splits are done. If *maxsplit* is not specified or -1,
    then there is no limit on the number of splits:

        >>> list(split_before(range(10), lambda n: n % 3 == 0, maxsplit=2))
        [[0, 1, 2], [3, 4, 5], [6, 7, 8, 9]]
    """
    if maxsplit == 0:
        yield list(iterable)
        return

    buf = []
    it = iter(iterable)
    for item in it:
        if pred(item) and buf:
            yield buf
            if maxsplit == 1:
                yield [item] + list(it)
                return
            buf = []
            maxsplit -= 1
        buf.append(item)
    if buf:
        yield buf
```
````

4. **Insight**: RepoMap provides interfaces without implementations, docstrings, or usages. As a result, the code completion model may misinterpret function purposes and misuse them.

   Example: In Example 5, RepoMap provided only the signatures of `unique_justseen` and `unique_everseen`, without showing how they differ. The oracle agent's contexts included

implementations and usage patterns, clarifying that one checks against only the previous element while the other checks against all previously seen elements. The RepoMap-guided completion misunderstood this distinction:

```
def classify_unique(iterable, key=None):
    seen = set()
    prev = object()
    use_key = key is not None
    for element in iterable:
        k = key(element) if use_key else element
        if use_key:
            try:
                if k in seen:
                    yield (element, False, False)
                else:
                    seen.add(k)
                    yield (element, True, True)
            except TypeError:
                if k in seen:
                    yield (element, False, False)
                else:
                    seen.add(k)
                    yield (element, True, True)
        else:
            if element in seen:
                yield (element, False, False)
            else:
                seen.add(element)
                yield (element, True, True)
```

5. **Insight**: RepoMap does not prioritize relevance. It retrieves generic API structures without ranking them by contextual similarity, while implementations of related functions are often crucial for guiding completion.

   Example: In repository ID 16 from the Sympy repository, the target function was `to_loads` in the `ForceActuator` class. RepoMap retrieved unrelated signatures, whereas the oracle agent surfaced highly relevant contexts such as the abstract method `PathwayBase.to_loads` and examples of other actuator classes implementing their `to_loads` methods, leading to a correct completion:

```
def to_loads(self):
    return self.pathway.to_loads(self.force)
```

These examples helped the model generate the correct completion, while RepoMap's results lacked actionable guidance:

```
# sympy/abc.py
_latin
_greek
ns
_clash1
_clash2
_clash

# sympy/__init__.py
def enable_warnings():
def __sympy_debug():
SYMPY_DEBUG
test
doctest
__all__

# sympy/physics/mechanics/joint.py
__all__
class Joint(ABC):
    def __init__(self, name, parent, child, coordinates=None, speeds=None,
                 parent_point=None, child_point=None, parent_interframe=None,
                 child_interframe=None, parent_axis=None, child_axis=None,
                 parent_joint_pos=None, child_joint_pos=None):
    def __str__(self):
    def __repr__(self):
    def name(self):
    def parent(self):
    def child(self):
    def coordinates(self):
    def speeds(self):
    def kdes(self):
    def parent_axis(self):
```

```
    def child_axis(self):
    def parent_point(self):
    def child_point(self):
    def parent_interframe(self):
    def child_interframe(self):
    def _generate_coordinates(self, coordinates):
    def _generate_speeds(self, speeds):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
    def _to_vector(matrix, frame):
    def _axis(ax, *frames):
    def _choose_rotation_axis(frame, axis):
    def _create_aligned_interframe(frame, align_axis, frame_axis=None,
                                   frame_name=None):
    def _generate_kdes(self):
    def _locate_joint_pos(self, body, joint_pos, body_frame=None):
    def _locate_joint_frame(self, body, interframe, body_frame=None):
    def _fill_coordinate_list(self, coordinates, n_coords, label='q', offset=0,
                              number_single=False):
        def create_symbol(number):
class PinJoint(Joint):
    def __init__(self, name, parent, child, coordinates=None, speeds=None,
                 parent_point=None, child_point=None, parent_interframe=None,
                 child_interframe=None, parent_axis=None, child_axis=None,
                 joint_axis=None, parent_joint_pos=None, child_joint_pos=None):
    def __str__(self):
    def joint_axis(self):
    def _generate_coordinates(self, coordinate):
    def _generate_speeds(self, speed):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
class PrismaticJoint(Joint):
    def __init__(self, name, parent, child, coordinates=None, speeds=None,
                 parent_point=None, child_point=None, parent_interframe=None,
                 child_interframe=None, parent_axis=None, child_axis=None,
                 joint_axis=None, parent_joint_pos=None, child_joint_pos=None):
    def __str__(self):
    def joint_axis(self):
    def _generate_coordinates(self, coordinate):
    def _generate_speeds(self, speed):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
class CylindricalJoint(Joint):
    def __init__(self, name, parent, child, rotation_coordinate=None,
                 translation_coordinate=None, rotation_speed=None,
                 translation_speed=None, parent_point=None, child_point=None,
                 parent_interframe=None, child_interframe=None,
                 joint_axis=None):
    def __str__(self):
    def joint_axis(self):
    def rotation_coordinate(self):
    def translation_coordinate(self):
    def rotation_speed(self):
    def translation_speed(self):
    def _generate_coordinates(self, coordinates):
    def _generate_speeds(self, speeds):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
class PlanarJoint(Joint):
    def __init__(self, name, parent, child, rotation_coordinate=None,
                 planar_coordinates=None, rotation_speed=None,
                 planar_speeds=None, parent_point=None, child_point=None,
                 parent_interframe=None, child_interframe=None):
    def __str__(self):
    def rotation_coordinate(self):
    def planar_coordinates(self):
    def rotation_speed(self):
    def planar_speeds(self):
    def rotation_axis(self):
    def planar_vectors(self):
    def _generate_coordinates(self, coordinates):
    def _generate_speeds(self, speeds):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
class SphericalJoint(Joint):
    def __init__(self, name, parent, child, coordinates=None, speeds=None,
                 parent_point=None, child_point=None, parent_interframe=None,
                 child_interframe=None, rot_type='BODY', amounts=None,
                 rot_order=123):
    def __str__(self):
    def _generate_coordinates(self, coordinates):
    def _generate_speeds(self, speeds):
```

```
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):
class WeldJoint(Joint):
    def __init__(self, name, parent, child, parent_point=None, child_point=None,
                 parent_interframe=None, child_interframe=None):
    def __str__(self):
    def _generate_coordinates(self, coordinate):
    def _generate_speeds(self, speed):
    def _orient_frames(self):
    def _set_angular_velocity(self):
    def _set_linear_velocity(self):

# sympy/physics/mechanics/loads.py
__all__
class LoadBase(ABC, namedtuple('LoadBase', ['location', 'vector'])):
    def __add__(self, other):
    def __mul__(self, other):
class Force(LoadBase):
    def __new__(cls, point, force):
    def __repr__(self):
    def point(self):
    def force(self):
class Torque(LoadBase):
    def __new__(cls, frame, torque):
    def __repr__(self):
    def frame(self):
    def torque(self):
def gravity(acceleration, *bodies):
def _parse_load(load):

# sympy/physics/mechanics/pathway.py
__all__
class PathwayBase(ABC):
    def __init__(self, *attachments):
    def attachments(self):
    def attachments(self, attachments):
    def length(self):
    def extension_velocity(self):
    def to_loads(self, force):
    def __repr__(self):
class LinearPathway(PathwayBase):
    def __init__(self, *attachments):
    def length(self):
    def extension_velocity(self):
    def to_loads(self, force):
class ObstacleSetPathway(PathwayBase):
    def __init__(self, *attachments):
    def attachments(self):
    def attachments(self, attachments):
    def length(self):
    def extension_velocity(self):
    def to_loads(self, force):
class WrappingPathway(PathwayBase):
    def __init__(self, attachment_1, attachment_2, geometry):
    def geometry(self):
    def geometry(self, geometry):
    def length(self):
    def extension_velocity(self):
    def to_loads(self, force):
    def __repr__(self):
def _point_pair_relative_position(point_1, point_2):
def _point_pair_length(point_1, point_2):
def _point_pair_extension_velocity(point_1, point_2):

# sympy/physics/mechanics/rigidbody.py
__all__
class RigidBody(BodyBase):
    def __init__(self, name, masscenter=None, frame=None, mass=None,
                 inertia=None):
    def __repr__(self):
    def frame(self):
    def frame(self, F):
    def x(self):
    def y(self):
    def z(self):
    def inertia(self):
    def inertia(self, I):
    def central_inertia(self):
    def central_inertia(self, I):
    def linear_momentum(self, frame):
    def angular_momentum(self, point, frame):
    def kinetic_energy(self, frame):
    def set_potential_energy(self, scalar):
    def parallel_axis(self, point, frame=None):

# sympy/physics/vector/__init__.py
```

```
__all__
```

6. **Insight**: RepoMap shows function signatures but not how they are used. This omission can lead the model to pass arguments of the right type but with incorrect values.

   Example: In repository ID 30 from Flask, the ground-truth function relied on the `aborter` method of the `current_app` object:

```
from .globals import current_app

def abort(code, *args, **kwargs):
    if current_app:
        current_app.aborter(code, *args, **kwargs)
    _wz_abort(code, *args, **kwargs)
```

   RepoMap retrieved only the variable name, with no type or member function information:

```
# src/flask/globals.py
_no_app_msg
_cv_app
app_ctx
current_app
g
_no_req_msg
_cv_request
request_ctx
request
session
```

   As a result, the completion failed. The oracle agent, by contrast, retrieved definitions and usage examples of `current_app`, including its `aborter` member, enabling the correct implementation:

````
**Explanation**: The `redirect` function from `src/flask/helpers.py` follows a similar pattern to the
↪ `abort` function

The `redirect` function in `helpers.py` shows the pattern you need to follow for the `abort` function.
↪ It checks if `current_app` exists and delegates to it, otherwise it falls back to Werkzeug
↪ directly:

```python
# src/flask/helpers.py
def redirect(
    location: str, code: int = 302, Response: type[BaseResponse] | None = None
) -> BaseResponse:
    """Create a redirect response object.

    If :data:`~flask.current_app` is available, it will use its
    :meth:`~flask.Flask.redirect` method, otherwise it will use
    :func:`werkzeug.utils.redirect`.

    :param location: The URL to redirect to.
    :param code: The status code for the redirect.
    :param Response: The response class to use. Not used when
        ``current_app`` is active, which uses ``app.response_class``.

    .. versionadded:: 2.2
        Calls ``current_app.redirect`` if available instead of always
        using Werkzeug's default ``redirect``.
    """
    if current_app:
        return current_app.redirect(location, code=code)

    return _wz_redirect(location, code=code, Response=Response)
```

**Explanation**: The `current_app` object from `src/flask/globals.py`

The `current_app` object is imported from `globals.py` and is used to check if we are in an
↪ application context:

```python
# src/flask/globals.py
from __future__ import annotations

import typing as t
from contextvars import ContextVar

from werkzeug.local import LocalProxy

if t.TYPE_CHECKING:  # pragma: no cover
````
````

```
    from .app import Flask
    from .ctx import _AppCtxGlobals
    from .ctx import AppContext
    from .ctx import RequestContext
    from .sessions import SessionMixin
    from .wrappers import Request


_no_app_msg = """\
Working outside of application context.

This typically means that you attempted to use functionality that needed
the current application. To solve this, set up an application context
with app.app_context(). See the documentation for more information.\
"""
_cv_app: ContextVar[AppContext] = ContextVar("flask.app_ctx")
app_ctx: AppContext = LocalProxy(  # type: ignore[assignment]
    _cv_app, unbound_message=_no_app_msg
)
current_app: Flask = LocalProxy(  # type: ignore[assignment]
    _cv_app, "app", unbound_message=_no_app_msg
)
```
```

7. **Insight**: RepoMap does not handle multi-hop dependencies. It captures only one-hop function signatures and misses deeper chains of dependency.

   **Example**: In the same Flask case, RepoMap retrieved the `current_app` variable but completely missed its member function `aborter`, a two-hop dependency critical for correct completion.
```

H Hyperparameter settings

This section summarizes the main hyperparameter settings used in the experiments. Unless otherwise noted, the same settings are used for both completion backbones and across the main and appendix experiments.

Completion models and decoding. We evaluate two completion backbones, Qwen3-8B and Qwen3-30B-A3B (Yang et al., 2025). For generation, we use a maximum completion length of 4096 tokens, temperature 0.7, and nucleus sampling with top-$p = 0.8$.

Agent backbones. In the main experiments, all agents use Claude 3.7 Sonnet (Anthropic, 2025) as the backbone. In Section A, we repeat the indexing-time agent experiments with Qwen3-Coder (Team, 2025) while keeping the remaining settings unchanged.

Prompt context budgets. The left context, right context, and retrieved cross-file context are each capped at 10K tokens, as noted in Section 6.1. These budgets are applied uniformly across baselines and agent-based methods.

Agent budgets. For agent-based methods, the exploration budget is limited to 10 high-priority files per instance. The retriever, forecaster, SpecAgent, and oracle agent each return at most 12 context blocks. In the main SpecAgent configuration, these 12 blocks are composed of 9 retrieved blocks and 3 predicted blocks. The composition ablation in Section 6.3 varies the number of prediction blocks over {0, 1, 3, 6, 12}, with the number of retrieval blocks adjusted so that the total remains 12.

Sparse and dense retrieval settings. For retrieval baselines, repository files are segmented into 50-line chunks. The sparse retrieval query is formed from the final 50 non-empty lines of the target prompt. Reranking is performed over at most 200 candidate chunks drawn from up to 1000 candidate files, and the final prompt includes up to 5 retrieved cross-file chunks. BM25 is the default sparse ranking function. Dense retrieval uses the same candidate generation pipeline, with either UniXcoder (Guo et al., 2022) or CodeSage v2 large (Zhang et al., 2024) as the reranker.

RepoMap combination setting. For the BM25 + RepoMap hybrid, the interpolation weight on the RepoMap component is set to 0.3.

Serving setup. All experiments are run on a cluster with 8 A100 GPUs, matching the setup described in Section 6.1. When batched serving is used, tensor parallelism is set to 8 so that a model is sharded across the 8 GPUs.

I Potential risks

While SpecAgent introduces a practical and efficient framework for repository-aware code completion, several potential risks and considerations remain.

Repository privacy and data security. Indexing-time exploration involves broad repository access, which could expose sensitive information if applied to private codebases without proper access control or data governance. Deployments should ensure strict adherence to organizational privacy policies and perform indexing in secure, access-controlled environments.

Speculative generation reliability. SpecAgent's speculative predictions, while beneficial for recall and coverage, may occasionally introduce misleading or obsolete context if the actual repository evolution diverges significantly from the predicted trajectory. Ensuring proper validation, caching strategies, and developer oversight is important to mitigate hallucinated or stale suggestions.

Benchmark generalization. Although we design a leakage-free synthetic benchmark to avoid future context contamination, synthetic data may not perfectly represent the complexity or noise of real-world repositories. Performance reported here should therefore be interpreted as indicative of potential gains rather than definitive real-world accuracy.

Computational and environmental costs. Indexing-time agents perform extensive repository analysis and speculative computation. While amortized over future completions, large-scale indexing may still incur nontrivial compute and energy overhead. Future work should explore more efficient incremental indexing pipelines to reduce the environmental footprint.

Overreliance on automation. SpecAgent's proactive design could encourage overreliance on automated code suggestions. Developers should treat generated completions as assistive rather than authoritative, maintaining human oversight to ensure correctness, security, and maintainability.

J Licenses and responsible use

The REPOCOD dataset (Liang et al., 2025b) is distributed under the BSD-3-Clause license. The Qwen3 and Qwen3-Coder models (Yang et al., 2025; Team, 2025) are released under the Apache License 2.0. All other third-party tools and libraries used in this work comply with their respective open-source licenses. No proprietary or restricted data sources were used in our experiments.

Consistency with intended use. All artifacts used in this study were employed in a manner consistent with their stated intended use. REPOCOD was explicitly released for research on repository-level code understanding and generation, aligning with our use in evaluating retrieval and completion methods. Similarly, Qwen3 and Qwen3-Coder are open-source LLMs intended for research and development purposes, and our use was confined strictly to academic experimentation and analysis. All derived datasets and benchmarks introduced in this work are designed solely for research and evaluation under the same conditions, and are not intended for production or commercial deployment.

Data privacy and content verification. The REPOCOD dataset is derived from publicly available open-source repositories and does not contain personally identifying information or private code under restrictive licenses. Before conducting experiments, we verified that no data in our evaluation corpus contained names, credentials, or other sensitive identifiers. Our synthetic benchmark, created by applying the function removal process to REPOCOD, inherits this property and introduces no additional human-related or offensive content. No human annotation, crowdsourced labeling, or user-generated personal data was collected or processed in this study.

Anonymization and protection. All intermediate artifacts (e.g., indexed repository states and context blocks) were stored and analyzed on secure research servers with controlled access. No attempt was made to de-anonymize contributors to the original repositories, and no identifiers linking to individual developers were used in training or evaluation.