IMPROVING CODE LOCALIZATION WITH REPOSITORY MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Code localization is a fundamental challenge in repository-level software engineering tasks such as bug fixing. While existing methods equip language agents with comprehensive tools/interfaces to fetch information from the repository, they overlook the critical aspect of *memory*, where each instance is typically handled from scratch assuming no prior repository knowledge. In contrast, human developers naturally build long-term repository memory, such as the functionality of key modules and associations between various bug types and their likely fix locations. In this work, we augment language agents with such memory by leveraging a repository's *commit history*—a rich yet underutilized resource that chronicles the codebase's evolution. We introduce tools that allow the agent to retrieve from a non-parametric memory encompassing recent historical commits and linked issues, as well as functionality summaries of actively evolving parts of the codebase identified via commit patterns. We demonstrate that augmenting such a memory can significantly improve LocAgent, a state-of-the-art localization framework, on both SWE-bench-verified and the more recent SWE-bench-live benchmarks. Our research contributes towards developing agents that can accumulate and leverage past experience for long-horizon tasks, more closely emulating the expertise of human developers.

1 INTRODUCTION

Repository-level software engineering tasks, such as bug fixing, are a promising application for Large Language Model (LLM)-powered agents (Jimenez et al., 2024). A crucial first step in these tasks is **code localization**: identifying the specific files and code segments that need to be modified to resolve the issue at hand. Existing methods mainly focus on building powerful toolsets that help agents navigate and reason over code relationships (Liu et al., 2025; Yu et al., 2025; Ouyang et al., 2025; Chen et al., 2025b; Ma et al., 2025). A leading framework is LocAgent (Chen et al., 2025b), which parses codebases into directed heterogeneous graphs that capture code structures and dependencies, enabling effective search for relevant entities.

Despite steady progress, current approaches share a key limitation: they treat every problem as a fresh puzzle, solved from scratch assuming no prior knowledge of the repository. Human developers, by contrast, accumulate and leverage long-term repository memory over time—this includes cached understanding of the purpose of core and actively evolving modules, and various associations between recurring bug patterns and their likely fix locations. This accumulated memory is what allows developers to grow into experts in a codebase.

The importance of such memory is also clear when looking at failure cases of existing localization frameworks. To illustrate, consider a failure case of LocAgent on a bug in the `django` repository from SWE-bench, as illustrated in Figure 2. The example is about `django`'s migration system, which generates migration programs from a user-defined schema. Here, the challenge is to find where import statements for certain base classes are synthesized, since the bug stems from missing imports in the generated program. Without prior knowledge of the repository, an agent must embark on a complex investigation, carefully tracing data/control flows and function calls across different folders and files to find the source of the error. In this example, while the agent successfully located some initial key entities, it eventually failed to complete the reasoning chain and stopped prematurely.

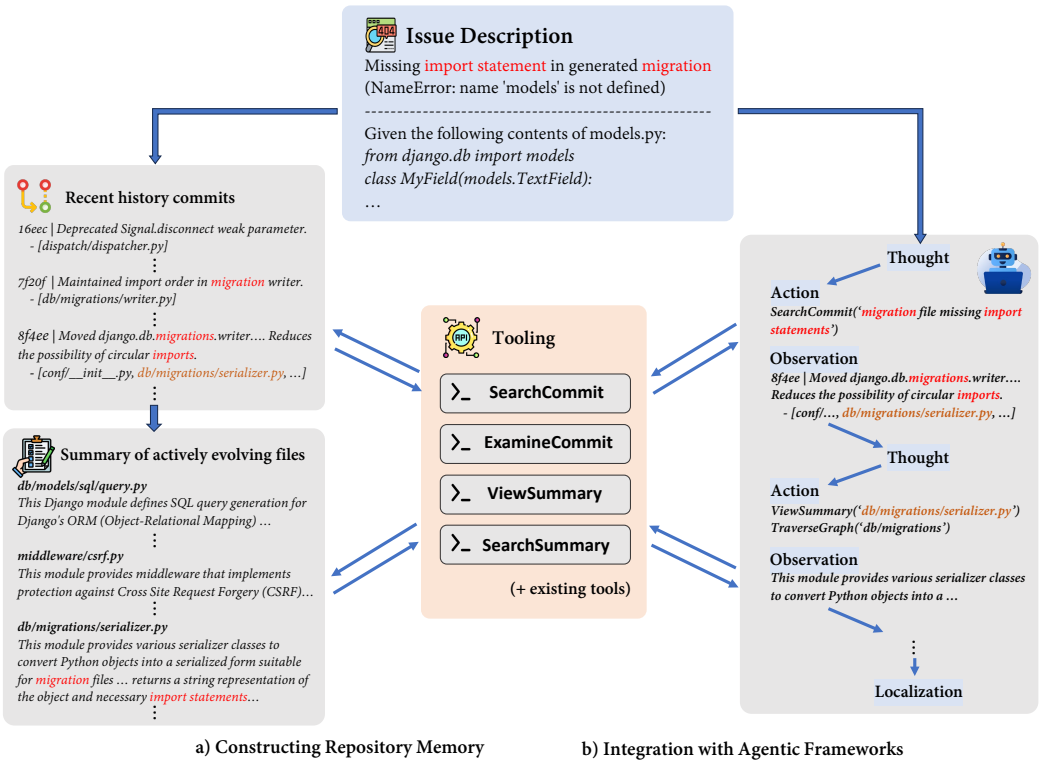

Figure 1: An overview of our repository memory design. **(a)** We construct the memory by leveraging the recent commit history of the repository. This involves creating a searchable database of past commits and their linked issues, and identifying frequently edited files to let LLMs generate high-level functionality summaries. **(b)** The memory is accessed by the language agent via a set of tools that perform search based on custom queries and support closer examination of individual memory entries. Details in §3.

Experienced developers would likely approach the problem differently. They could draw on *episodic memory* of past issues/codebase changes related to the migration system, or recall from *semantic memory* the modules that are potentially responsible for handling such imports within the codebase. These memories could provide strong priors for the investigation, guiding the search/reasoning to more effectively reach the error source.

How can we equip agents with such kind of memory? We propose to leverage the repository's *commit history*—a natural record of its past evolution. In particular, new problems are often connected to some past changes, where the related commit patches and linked issue contents could provide valuable data source for approximating the episodic memory. Commit statistics could also naturally reveal which parts of the codebase are most active, making them prime candidates for building semantic memory. Returning to the example in Figure 2, we find that even a simple keyword search (*"migration"*, *"import"*) over the commit messages retrieves many related history patches in the `django` migration system, such as problems with circular imports and nested classes. Likewise, analyzing commit frequency highlights the target file as a module under active development, and a pre-computed summary of its functionality—managing object-to-string conversion and import statements—could provide a strong signal of its relevance to the issue.

Building on these intuitions, we design two simple memory mechanisms to augment existing frameworks:

- **Episodic Memory of Past Commits.** We crawl and preprocess the commit history and linked data, and provide tools for agents to 1) search this corpus via custom queries that are matched with the commit messages, and 2) examine the details of individual commits, such as linked

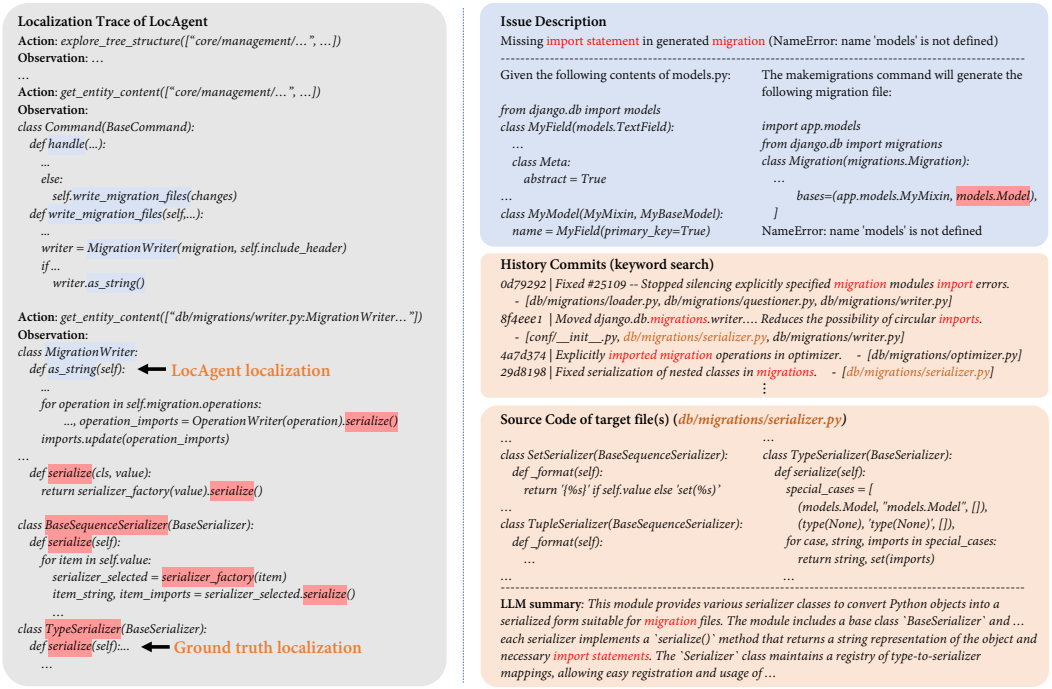

Figure 2: **(Left)** Localization trajectory of a failure case of LocAgent on SWE-bench-verified (*django__django-14580*). While the agent successfully traces some initial key entities, it fails to reason in greater depth and granularity to pinpoint the error source, resulting in wrong localizations. **(Right)** The original issue description (top), accompanying history commits obtained via simple keyword search on commit messages (middle), the source code and LLM-generated functionality summary of the ground truth target file containing the error source (bottom).

issues and commit patches. The episodic memory allows agents to reference past codebase changes to aid in resolving the current issue.

- **Semantic Memory of Active Code Functionality.** We identify the most active parts of the codebase by analyzing commit frequency to find the most frequently edited files. For these key files, we use an LLM to generate high-level summaries of their functionalities. This creates a compact knowledge base of the most dynamic parts of the codebase, which the agent can query to understand the purpose of potentially relevant modules.

Experiments show that augmenting LocAgent with these memory components could significantly improve localization performance, where we observe strong gains on both SWE-bench-verified (Jimenez et al., 2024) and the more recent SWE-bench-live (Zhang et al., 2025) benchmark.

To summarize, we make the following contributions:

- We identify the lack of long-term memory as a critical limitation in current language agents for repository-level software engineering tasks such as code localization.
- We propose to leverage commit history as a natural and rich source for building repository memory, and introduce two simple memory mechanisms—episodic memory of past commits and semantic memory of active code functionality—that integrate easily into existing frameworks.
- We show that these mechanisms yield substantial improvements in code localization, highlighting the value of incorporating long-term memory into agent workflows.

2 Related Work

2.1 Code Localization in Repository-level Software Engineering tasks

Existing methods for code localization could be broadly categorized into three types: 1) retrieval-based, 2) agentic approaches and 3) procedural approaches.

Retrieval-based methods represent the most conventional approach, leveraging lexical or semantic matching to rank code snippets based on their proximity to the query (Wang et al., 2025b). Recent advances have focused on improving the quality of code embeddings, often through contrastive learning objectives (Li et al., 2022; Wang et al., 2023; Zhang et al., 2024a; Suresh et al., 2025). Among these, CoRNStack (Suresh et al., 2025) represents the current state of the art, achieving strong performance through large-scale training coupled with rigorous data filtering and hard negative mining strategies.

Agentic frameworks. Agentic approaches augment LLMs with the ability to interact with an external environment to gather information via a set of tools/interfaces, where the major focus has been to improve the comprehensiveness of the tool designs (Yang et al., 2024; Cognition, 2024; Örwall, 2024; Zhang et al., 2024b; Chen et al., 2025b; Wang et al., 2025a; Yu et al., 2025; Ouyang et al., 2025; Ma et al., 2025; Liu et al., 2025). Notably, LocAgent (Chen et al., 2025b) is a SoTA agentic framework for code localization. It parses codebases into heterogeneous graphs capturing code structures and various kinds of dependencies (e.g., import, invoke and inherit relationships), which allows LLM agents to more effectively comprehend and navigate through the codebase.

Procedural approaches directly employ LLMs to perform localization in a pre-designed procedure (Zhang et al., 2023; Wu et al., 2024; Xia et al., 2025; Liang et al., 2024), which avoids the complex setups of agentic approaches. The most representative and high-performing method is Agentless (Xia et al., 2025), which performs localization by prompting LLMs with the issue description and a concise representation of the repository's file and directory structure.

2.2 Memory-Enhanced Language Agents

Our work is connected with the broader literature on enhancing language agents with memory or experience (Qian et al., 2024; Chen et al., 2025a; Wang et al., 2024; 2025c; Zheng et al., 2025). The most related work is arguably Chen et al. (2025a), which distills procedural knowledge from an agent's past success and failure trajectories to facilitate online problem-solving. Orthogonally, our approach leverages commit histories to construct a repository-specific memory, providing knowledge that is grounded in the codebase's evolution rather than the agent's individual experience.

3 Repository Memory

To bridge the gap between memoryless agents and experienced developers, we tap into the repository's commit history—a rich, structured chronicle of its evolution. We structure this historical data into two complementary memory stores, designed to be lightweight and easily integrated into existing agentic frameworks. The first, an episodic memory, captures the narrative of specific past changes. The second, a semantic memory, distills high-level functional knowledge about the codebase's most dynamic areas. An illustration of this design is provided in Figure 1.

3.1 Episodic Memory of Past Commits

Memory Construction. This memory captures concrete entries of past problems and their solutions. We build a structured corpus from the repository's recent commit history, storing the code patches and also the rich metadata surrounding them: commit messages, timestamps, and links to associated issues. The corpus only includes commits made prior to the issue to be resolved (to avoid contamination). We further filter this datastore to remove issues that have overlapping text with the test instance and commits that are linked to these issues, to prevent leakage.

Memory Interfaces. The agent interacts with this historical database through a dedicated interface, allowing it to query for past events that are related to its current task:

- `SearchCommit(query, top_k)`: This tool performs case-based retrieval. The agent can issue a query, which could be derived from the given bug report or current problem-solving state, to find semantically related historical commits. We use BM25 for matching the query against the commit messages, as it is highly effective for the semi-structured, keyword-rich nature of commit messages. The interface returns a ranked list of the top-k relevant commits, including their unique IDs (commit SHAs), messages, and modified files in the commit patch.
- `ExamineCommit(id)`: Once a potentially relevant commit is identified, this tool allows the agent to "zoom in" and retrieve its full context based on its ID, including the complete code patch (in diff format) and any linked issues, providing a comprehensive view of the original problem and its corresponding solution.

By using these tools, the agent can ground its reasoning in historical precedent, leveraging past solutions as powerful exemplars to aid in its understanding of the codebase/problem and guide its investigation.

3.2 Semantic Memory of Active Code Functionality

Memory Construction. While episodic memory provides specific examples, semantic memory offers a generalized, high-level understanding of the codebase. The rationale is that files frequently modified in the recent past are "development hotspots"—areas that are either central to the repository's functionality or are undergoing active change, making them more likely to be relevant to new issues. We first analyze the commit history to identify the top-k most edited files, where k is much smaller than the total amount of files in the codebase. Then, for each of these files, we use an LLM to read its source code and distill its functionalities into a high-level natural language summary. This process creates a compact semantic knowledge base that maps critical files to their core responsibilities, focusing exclusively on the most dynamic parts of the repository.

Memory Interfaces. The agent accesses this knowledge base again through a simple query interface:

- `ViewSummary(file_name)`: This retrieves the cached summary for a specific file (if it exists in the memory), allowing the agent to quickly understand a file's purpose without needing to read its entire source code.
- `SearchSummary(query, top_k)`: This allows the agent to perform a keyword-based search over the entire collection of file summaries. It returns the top-k most relevant (file, summary) pairs, helping the agent to locate modules that are related to the issue or current exploration intent.

The semantic memory provides the agent with crucial architectural context, biasing its search towards more promising areas and preventing it from getting lost in the vast, irrelevant or stable parts of the codebase.

3.3 Integration with LocAgent

The memory tools are designed to be modular and can be straightforwardly integrated into existing agentic frameworks. In this work, we integrate them into **LocAgent**, a state-of-the-art localization framework that operates based on the ReAct paradigm (Yao et al., 2023). A LocAgent-powered agent iteratively cycles through a "Thought, Act, Observation" loop. In the "Act" step, it synthesizes an API call to one of its available tools, whose execution feedback is returned to the agents via the next "Observation" entry. For context, LocAgent's core tools allow it to navigate a heterogeneous graph representation of the codebase:

- `SearchEntity`: Searches the codebase for entities matching a keyword query, typically serving as an entry point for exploration.
- `TraverseGraph`: Performs a multi-hop, type-aware breadth-first search from a starting entity to explore code relationships, which include 1) basic *contain* relationships between folders and files, 2) *invoke* relationships between functions and classes, 3) *import* relationships from files to functions/classes, and 4) the *inherit* relationship between classes.
- `RetrieveEntity`: Fetches the full source code and detailed information for a specific code entity (e.g., a file, class, or function).

Table 1: Main results on code localization benchmarks. RepoMem significantly outperforms all other methods across both benchmarks, demonstrating the effectiveness of incorporating repository memory. Both episodic and semantic memory components contribute positively, with their combination yielding the best performance.

Methods	SWE-bench-verified			SWE-bench-live		
	Acc@1	*Acc@3*	*Acc@5*	*Acc@1*	*Acc@3*	*Acc@5*
CodeRankEmbed (Suresh et al., 2025)	29.6	45.1	54.3	26.2	44.6	52.3
Agentless (Xia et al., 2025)	53.3	67.8	71.4	40.0	60.0	62.3
LocAgent (Chen et al., 2025b)	64.8	70.4	71.6	59.2	60.8	63.1
RepoMem (episodic-only)	67.8	72.4	74.3	60.0	61.5	64.6
RepoMem (semantic-only)	65.0	71.0	72.8	56.9	61.5	63.9
RepoMem	**68.6**	**74.5**	**76.5**	**60.8**	**63.9**	**66.2**

Our integration simply expands the action space with the memory-based tools, as illustrated in Figure 1. Intuitively, the memory-based tools could nicely complement the existing toolset in LocAgent. For example, an agent can now use memory search tools to fetch related commits or files, combined with concrete examination of individual entries when necessary, to form an experience-based hypothesis. It can then use LocAgent's tools to perform a more detailed investigation of the code entities surrounding these candidates. This creates a powerful synergy between high-level, memory-guided direction and low-level, structural code analysis.

4 Experiments

4.1 Setup

Datasets. We evaluate our approach on two benchmarks: **SWE-bench-verified** (Jimenez et al., 2024), which contains 500 examples from 12 repositories, and the more recent **SWE-bench-live** (Zhang et al., 2025) benchmark. For SWE-bench-live, we use a high-quality subset created from the intersection of its 'lite' and 'verified' splits, filtering for instances requiring five or fewer files to be modified. This results in 130 examples across 62 repositories.

Baselines. We compare our method, **RepoMem**, against several state-of-the-art methods in different types of approaches:

- **CodeRankEmbed** (Suresh et al., 2025), a leading retrieval-based method leveraging large-scale training with careful data filtering and hard negative mining.
- **Agentless** (Xia et al., 2025), a leading procedural method that prompts an LLM with repository structure.
- **LocAgent** (Chen et al., 2025b), a state-of-the-art agentic framework for localization as discussed earlier, which our RepoMem method is built directly upon. This also allows for a direct comparison of the impact of integrating repository memory.

Evaluation Metrics. We evaluate file-level localization performance via **Accuracy@k** (following prior work (Chen et al., 2025b)), defined as the percentage of examples where the set of top-k predicted files completely covers the ground-truth files.

Implementation Details. All experiments use GPT-4o (2024-05-13) as the backbone LLM. For memory construction, we consider the 7,000 commits prior to the given issue's base commit, and identify the top 200 most frequently edited files for constructing the semantic memory.

4.2 Main Results

Table 1 presents the main experimental results. **RepoMem** consistently outperforms baselines on both benchmarks. On SWE-bench-verified, RepoMem achieves an *Acc@5* of 76.5%, a 4.9% absolute improvement over the strong LocAgent baseline. The gains are also consistent on the more diverse SWE-bench-live dataset. Ablating on the effect of each memory, using only episodic memory ('episodic-only') provides a significant boost over LocAgent, demonstrating the value of

Table 2: Per-repository performance comparison (Acc@5) on SWE-bench-verified. Repositories are sorted by the average number of historical commits available, where "others" is the union of repos with less than 10K commits. Our method sees strong gains in repositories with rich commit histories but can be hindered in those with limited or irrelevant history.

Repo	matplotlib	sympy	astropy	django	scikit-learn	sphinx	pytest	others
# Examples	34	73	22	231	32	44	18	46
# Avg. Commits	43.9K	39.8K	31.2K	29.2K	25.1K	17.2K	12.4K	4.5K
LocAgent *Acc@5*	76.5	69.9	86.4	72.3	93.8	47.7	61.1	67.4
RepoMem *Acc@5*	82.4 (+5.9)	72.6 (+2.7)	86.4 (+0.0)	79.7 (+7.4)	96.9 (+3.1)	59.1 (+11.4)	77.8 (+16.7)	54.3 (-13.1)

referencing past commit history. Similarly, using only semantic memory ('semantic-only') also improves performance by helping the agent focus on actively developed parts of the codebase. The best results are achieved when both memory components are combined, indicating that they provide complementary information: episodic memory offers concrete solutions to similar past problems, while semantic memory provides high-level architectural context for the agent to leverage.

Table 2 provides a breakdown of performance by repository on SWE-bench-verified, sorted by the average number of historical commits available. The results reveal a clear correlation: repositories with a rich commit history benefit the most from RepoMem. This strongly supports our hypothesis that commit history is a valuable source for memory building. Conversely, for the "others" group which consists of repositories with limited history, performance degrades. This is likely because the memory contains too little relevant information, and the agent's exploration of this sparse history can be more distracting than helpful.

4.3 Analysis

We perform a series of analyses of RepoMem on SWE-bench-verified, to gain deeper insights into the effect of integrating repository memory.

Shift in Agent Behavior. The introduction of memory drastically alters the agent's problem-solving strategy. As shown in Figure 3, agents equipped with the memory significantly reduce their reliance on exhaustive exploration tools (`TraverseGraph`) and direct code inspection (`RetrieveEntity`). This reflects a strategic shift from brute-force navigation to a more targeted, hypothesis-driven investigation, where the agent integrates its accumulated repository knowledge to form hypotheses, and performs detailed exploration/verification leverating the original LocAgent tools—a process that more closely mirrors an experienced human developer's workflow.

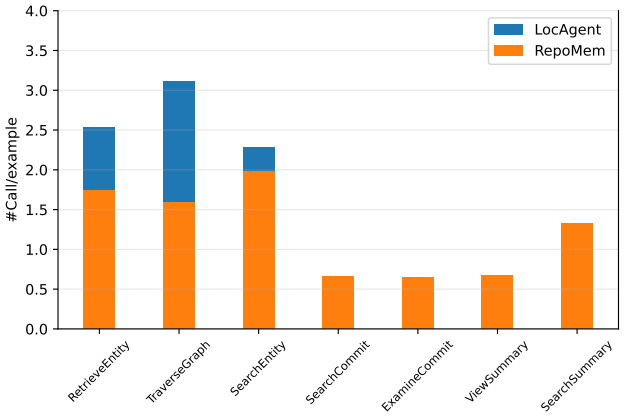

Figure 3: **Tool use distribution for LocAgent vs. RepoMem.** The introduction of memory-based tools drastically alters agent behavior. RepoMem significantly reduces its reliance on exhaustive exploration tools like `TraverseGraph` and direct code reading (`RetrieveEntity`), indicating a strategic shift from brute-force navigation to a more targeted, hypothesis-driven investigation guided by memory.

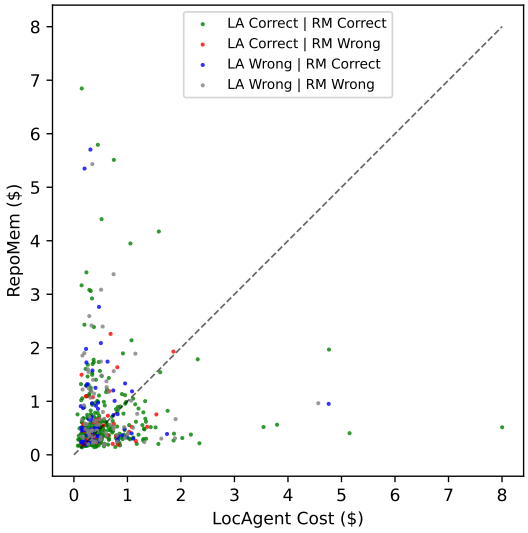

Figure 4: **Per-example cost comparison (LA: LocAgent, RM: RepoMem).** This scatter plot shows the LLM API cost for each example, where the x and y coordinates correspond to the cost of LocAgent and RepoMem, respectively. Points below the diagonal line indicate RepoMem was cheaper, while points above indicate it was more expensive. The high variance reveals that the efficiency impact of integrating memory is problem-dependent: it provides significant savings on some tasks but incurs overhead on others, a nuance missed by average cost metrics.

Table 3: **Cross-comparison of LLM API cost (LA: LocAgent, RM: RepoMem).** Each cell shows the average cost per example for LocAgent → RepoMem. The largest cost increase occurs in the bottom two quadrants, which are examples where LocAgent fails. This indicates that the additional cost is primarily a strategic investment to improve accuracy on difficult tasks.

	RM Succeeds	**RM Fails**
LA Succeeds	\$0.58 → \$0.68	\$0.59 → \$0.66
LA Fails	\$0.54 → \$0.89	\$0.59 → \$0.87

Table 4: **Impact of retrieval method for memory interface on the performance of `django` repository.** Sparse retrieval using BM25 with a custom LLM-based tokenizer outperforms both a standard tokenizer and a strong dense retrieval model (GritLM-7B).

Retrieval Methods	**django/django**		
	Acc@1	*Acc@3*	*Acc@5*
Dense retrieval	65.8	71.9	73.6
BM25 (whitespace)	67.1	74.5	77.9
BM25 (LLM)	70.1	76.6	79.7

Efficiency Analysis. We find that integrating the memory introduces a strategic cost-effectiveness trade-off instead of a uniform overhead. First, as shown in the cross-comparison in Table 3, the additional expenditure is primarily allocated to solving difficult problems—the most significant cost increase occurs in examples where LocAgent fails. This indicates that overall, our method strategically invests additional resources to solve challenging problems that the baseline cannot, rather than spending wastefully on problems that could already be solved without resorting to the memory.

More interestingly, the cost impact is highly variable at the instance level. Figure 4 shows a scatter plot of per-example costs, again cross-comparing the two methods. While the average cost increases, the plot reveals high variance across the examples. For many problems, RepoMem is significantly cheaper than LocAgent (points far below the diagonal), likely because the memory provided a more direct hint to the solution. For some others, it could instead be much more expensive (points far above the diagonal), likely on problems where the memory proved fruitless and only added overhead and distractions. This heterogeneity highlights that average cost can be a misleading metric, and the efficiency of our memory-augmented agent is highly dependent on the relationship between the current problem and the repository's history.

These findings also suggest a promising future direction: training agents to be more strategic about when to use memory tools. An agent that could first assess a problem's novelty might learn to

rely on memory for issues that are related to the past experience, while defaulting to first-principle explorations for unprecedented ones, optimizing the cost-effectiveness trade-off.

Retrieval Methods for Memory Interfaces. Here, we investigate the choice of retrieval method for our memory interfaces. We compare three approaches on the `django` repository, with results shown in Table 4. Here, we use the strong GritLM-7B model for dense retrieval (Muennighoff et al., 2025). Our default method in the main results uses BM25 with an LLM-based tokenizer that recognizes code entity names, which outperforms standard whitespace tokenization. More notably, sparse retrieval methods significantly outperform dense retrieval. We hypothesize this is due to the unique vocabulary of code-related utterances in software repositories—for example, entities like 'MigrationWriter' and 'OperationWriter' may be semantically close but are functionally very distinct. Sparse retrieval methods, which rely on exact keyword matches, excel at handling this "rigid" vocabulary. Similar phenomena are also observed in prior work, e.g., Sciavolino et al. (2021) finds that dense retrievers could drastically underperform sparse methods in entity-centric question-answering.

Error Analysis. We conducted a small-scale analysis of the failure cases of RepoMem to better understand its limitations. As expected, the primary failure mode occurs when memory retrieval yields little useful information about the issue, a problem stemming from either the novelty of the issue or shortcomings in the retrieval methods. In such instances, the agent receives irrelevant information that can pollute its reasoning context and distract it—a well-known challenge for LLMs (Shi et al., 2023). This can lead to performance worse than the baseline, as observed in repositories with sparse histories (Table 2). These findings highlight promising directions for future work, such as designing/training better memory interfaces and developing mechanisms that enable the agent to dynamically decide whether to rely on the memory or instead fall back on first-principles reasoning (as discussed earlier).

5 Conclusion

In this work, we take an initial step toward addressing a key limitation of current language agents for software engineering: their lack of long-term repository memory. We propose a simple yet effective solution that leverages the rich contextual information embedded in a repository's commit history. By building two complementary memory stores—an episodic memory of past commits and linked issues, and a semantic memory of active code functionality—we enable agents to draw on past knowledge when tackling future tasks. Our experiments show that this memory-augmented approach substantially improves code localization performance on established benchmarks. Further analysis reveals a shift in agent behavior toward a more experience-guided strategy that better reflects human expertise. Overall, this work underscores the importance of incorporating long-term memory into agent workflows, paving the way for more capable and experienced software engineering assistants.

Ethics Statement

All authors of this paper have read and adhered to the ICLR Code of Ethics. Our research is built upon publicly available datasets, which are derived entirely from open-source software repositories. The study does not involve human subjects, and our data processing steps do not introduce any new ethical concerns regarding privacy, bias, or fairness. The proposed methods are designed for software engineering assistance and do not present foreseeable risks of misuse or negative societal impact.

Reproducibility Statement

We have made every effort to ensure our work is reproducible. Our experiments are conducted on the public SWE-bench-verified and SWE-bench-live benchmarks. The methodology for constructing the episodic and semantic memory components is detailed in §3, and the implementation details are provided in §4.1. To further facilitate replication, we provide comprehensive documentations, examples, and prompts used for our agent in Appendix A and Appendix B. The source code for our framework and experiments will also be made publicly available upon publication.

REFERENCES

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. Swe-exp: Experience-driven software issue resolution, 2025a. URL https://arxiv.org/abs/2507.23361.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

Cognition. Introducing devin. https://www.cognition.ai/blog/introducing-devin, 2024. Blog post.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. CodeRetriever: A large scale contrastive pre-training method for code search. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 2898–2910, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.187. URL https://aclanthology.org/2022.emnlp-main.187/.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, wei jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. Repofuse: Repository-level code completion with fused dual context, 2024. URL https://arxiv.org/abs/2402.14323.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.7. URL https://aclanthology.org/2025.naacl-long.7/.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. How to understand whole software repository?, 2025. URL https://arxiv.org/abs/2406.01422.

Niklas Muennighoff, Hongjin SU, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. Generative representational instruction tuning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=BC4lIvfSzv.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=dw9VUsSHGB.

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, YiFei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Experiential co-learning of software-developing agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5628–5640, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.305. URL https://aclanthology.org/2024.acl-long.305/.

Christopher Sciavolino, Zexuan Zhong, Jinhyuk Lee, and Danqi Chen. Simple entity-centric questions challenge dense retrievers. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 6138–6148, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.496. URL https://aclanthology.org/2021.emnlp-main.496/.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pp. 31210–31227. PMLR, 2023.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=iyJOUELYir.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. LLMs in the imaginarium: Tool learning through simulated trial and error. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10583–10604, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.570. URL https://aclanthology.org/2024.acl-long.570/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1069–1088, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.68. URL https://aclanthology.org/2023.emnlp-main.68/.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-bench: Can retrieval augment code generation? In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3199–3214, Albuquerque, New Mexico, April 2025b. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.176. URL https://aclanthology.org/2025.findings-naacl.176/.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025c. URL https://openreview.net/forum?id=NTAhi2JEEE.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: An LLM agent framework for software issue localization. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=LyUfPOvM6I.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=vfzRRjumpX.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live!, 2025. URL https://arxiv.org/abs/2505.23419.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pp. 1592–1604, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Albert Örwall. Moatless tools. https://github.com/aorwall/moatless-tools, 2024. GitHub repository, MIT License.

A Documentation and Example Responses of Memory Tools

Figure 5, 6, 7, 8 show the documentation and example responses of the memory tools.

SearchCommit
Doc:
This tool searches the repository's commit history to find commits similar to a given query - typically a hypothetical commit message. It uses the BM25 ranking algorithm to match the input against historical commit messages, surfacing relevant past changes. Using this tool early in the investigation is recommended, as it can help identify related issues or changes from the past, allowing you to quickly locate potential areas in the codebase to focus on.

Parameters: {
'type': 'object',
'properties': {
'query_list': {
'type': 'array',
'items': {'type': 'string'},
'description': (
"A list of queries. Each query can be a hypothetical commit message, which would be matched with the history commit messages to retrieve similar commits."
)
},
'top_k': {
'type': 'integer',
'description': (
"top_k commits will be returned."
),
'default': 20
}
},
'required': ['query_list'],
}

Example API call: *SearchCommit(query='Fix Cannot drop unique_together constraint on a single field with its own unique=True constraint', top_k=20)*
Observation:
##Searching for query `Fix Cannot drop unique_together constraint on a single field with its own unique=True constraint`...
Search Result:
SHA: 73a6ab638 | Commit message: Fixed #25551 -- Fixed migration operations ordering when adding fields and a unique_together constraint. | Edited files: ['django/db/migrations/autodetector.py']
SHA: 533b20877 | Commit message: Fixed #29224 -- Fixed removing index_together indexes if exists unique_together constraint on the same fields. | Edited files: ['django/db/backends/base/schema.py']
SHA: f5a324391 | Commit message: Fixed #33125 -- Avoided redundant unique constraint when converting a non-unique field to primary key on MySQL and PostgreSQL. | Edited files: ['django/db/backends/base/schema.py', 'django/db/backends/oracle/schema.py']
...
==========
ExamineCommit
Doc:
This tool examines the details of a commit based on the short (9-digit) commit sha, including the commit patch and optionally the issue description.
Note that the line numbers displayed in the patch are from those history commits, not the current repo you are working on.

Parameters: {
'type': 'object',

Figure 5: Documentation and example outputs from the memory tools.

```
    'properties': {
        'sha_list': {
            'type': 'array',
            'items': {'type': 'string'},
            'description': (
                "A list of short 9-digit commit sha that you wish to examine."
            )
        },
        'display_issue': {
            'type': 'boolean',
            'description': (
                "Set to True if you wish to include the issue description in the output."                    ),
            'default': False
        }
    },
    'required': ['sha_list'],
}
```

Example API call: *ExamineCommit(sha=['50931dfa5'], display_issue=True)*
Observation:
##Commit for sha `50931dfa5`...
issue summary: Allow management commands to check if database migrations are applied
issue description: When creating a new project, you can sometimes forget to run `manage.py migrate` before creating the initial superuser (especially if you don't execute `runserver` before, which display a warning about migrations not applied). The resulting error make sense, it can't access to auth_user, since it does not exist yet:
{{{
$ django-admin.py startproject sample
$ cd sample/ && python manage.py createsuperuser
...
... but with a little try/except, it could be nicer and give a more meaningful information:

{{{
$ python manage.py createsuperuser
CommandError: You must execute `manage.py migrate` once before creating a super user
}}}
I have a patch ready (as I said, it just a try/except, see attachment) but, if accepted, it would need unit tests to be complete.
Edited file: django/contrib/auth/management/commands/changepassword.py
Patch:
@@ -12,7 +12,7 @@
 class Command(BaseCommand):
     help = "Change a user's password for django.contrib.auth."
-
+   requires_migrations_checks = True
     requires_system_checks = False
     def _get_pass(self, prompt="Password: "):

Edited file: django/contrib/auth/management/commands/createsuperuser.py
Patch:
@@ -23,6 +23,7 @@ class NotRunningInTTYException(Exception):
 class Command(BaseCommand):

Figure 6: Documentation and example outputs from the memory tools.

```
    help = 'Used to create a superuser.'
+   requires_migrations_checks = True
    def __init__(self, *args, **kwargs):
        super(Command, self).__init__(*args, **kwargs)
```

==========

ViewSummary

Doc:

This tool views summary of python files in the repo. You can input a list of file names you wish to examine.

Parameters: {

```
  'type': 'object',
  'properties': {
    'file_list': {
      'type': 'array',
      'items': {'type': 'string'},
      'description': (
        "A list of filenames that you wish to view the summaries."
      )
    },
  },
  'required': ['file_list'],
}
```

Example API call: *ViewSummary(file_list=["django/db/migrations/serializer.py"]):*

Observation:

##Summary for file `django/db/migrations/serializer.py`...

The `serializer.py` module in Django's migration system provides various serializer classes to convert Python objects into a serialized form suitable for migration files. The module includes a base class `BaseSerializer` and multiple subclasses, each handling specific types of objects such as lists, dictionaries, datetime objects, models, and custom functions. Each serializer implements a `serialize()` method that returns a string representation of the object and necessary import statements. The `Serializer` class maintains a registry of type-to-serializer mappings, allowing easy registration and usage of serializers through the `serializer_factory` function. This ensures objects are properly serialized when creating or applying migrations in Django.

==========

SearchSummary

Doc:

This tool searches the repository's cached file summaries to find files similar to a given query - typically a message describing some functionalities or keywords. It uses BM25 to match the input query against file summaries and returns top-k ones.

Parameters: {

```
  'type': 'object',
  'properties': {
    'query': {
      'type': 'string',
      'description': (
        "A query, which can be a message describing some functionalities or keywords, to be matched with the available file summaries to retrieve similar ones."
      )
    },
    'top_k': {
```

Figure 7: Documentation and example outputs from the memory tools.

```
            'type': 'integer',
            'description': (
                "the top_k files to return."
            ),
            'default': 5
        }
    },
    'required': ['query'],
}
```

Example API call: *SearchSummary(query='*generates a migration file that is missing an import statement*', top_k=4):*
Observation:
Searching for query `generates a migration file that is missing an import statement`...
Search Result:
file: django/db/migrations/operations/special.py | summary: The `special.py` file in `django/db/migrations/operations` defines three classes for managing database migrations in Django:\n\n1. **`SeparateDatabaseAndState`**: This class separates migration operations into those affecting the database and those affecting the state. It allows for operations that do not support state change to still apply the changes or vice versa. It includes methods to deconstruct operations, forward and backward migrations specifically for state and database, and a description of the combined state/database change.\n\n2. **`RunSQL`**: This class is used to run raw SQL statements during migrations. It supports providing a reverse SQL statement for reversible migrations and can handle state changes represented by state operations. It includes methods for deconstructing the operations, performing forward and backward migrations with SQL, and a description of the raw SQL operation.\n\n3. **`RunPython`**: This class facilitates running custom Python code during migrations, suitable for versioned ORM operations. It accepts a callable for forward migrations and an optional callable for reverse migrations. It includes methods to deconstruct the operation, perform forward and backward database migrations, and a description of the raw Python operation. It also ensures all models are reloaded to accommodate possible delays.\n\nEach class inherits from `Operation`, ensuring compatibility with Django's migration framework and providing essential interfaces for migration processes.

file: django/db/migrations/serializer.py | summary: The `serializer.py` module in Django's migration system provides various serializer classes to convert Python objects into a serialized form suitable for migration files. The module includes a base class `BaseSerializer` and multiple subclasses, each handling specific types of objects such as lists, dictionaries, datetime objects, models, and custom functions. Each serializer implements a `serialize()` method that returns a string representation of the object and necessary import statements. The `Serializer` class maintains a registry of type-to-serializer mappings, allowing easy registration and usage of serializers through the `serializer_factory` function. This ensures objects are properly serialized when creating or applying migrations in Django.

file: django/core/management/commands/makemigrations.py | summary: This Django management command, `makemigrations.py`, is used to create new database migration files for specified apps. It offers various options such as performing a dry run, merging migration conflicts, creating empty migrations, and controlling verbosity. It ensures consistent migration history across databases, checks for migration conflicts, and handles user prompts interactively or non-interactively. The script generates migration files based on detected model changes, writes them to disk, and can display the details for review. It also includes functionality for merging conflicting migrations interactively, ensuring consistency and resolving dependencies.

file: django/core/management/sql.py | summary: ...

Figure 8: Documentation and example outputs from the memory tools.

B AGENT PROMPT

We use the same task instruction prompt from the original LocAgent framework to guide the agent, which is displayed in Figure 9 for completeness.

Prompt

````
Given the following GitHub problem description, your objective is to localize the specific files, classes or functions, and lines
of code that need modification or contain key information to resolve the issue.

Follow these steps to localize the issue:
## Step 1: Categorize and Extract Key Problem Information
- Classify the problem statement into the following categories:
  Problem description, error trace, code to reproduce the bug, and additional context.
- Identify modules in the '{{package_name}}' package mentioned in each category.
- Use extracted keywords and line numbers to search for relevant code references for additional context.

## Step 2: Locate Referenced Modules
- Accurately determine specific modules
  - Explore the repo to familiarize yourself with its structure.
  - Analyze the described execution flow to identify specific modules or components being referenced.
- Pay special attention to distinguishing between modules with similar names using context and described execution flow.
- Output Format for collected relevant modules:
  - Use the format: 'file_path:QualifiedName'
  - E.g., for a function `calculate_sum` in the `MathUtils` class located in `src/helpers/math_helpers.py`, represent it as:
    'src/helpers/math_helpers.py:MathUtils.calculate_sum'.

## Step 3: Analyze and Reproducing the Problem
- Clarify the Purpose of the Issue
  - If expanding capabilities: Identify where and how to incorporate new behavior, fields, or modules.
  - If addressing unexpected behavior: Focus on localizing modules containing potential bugs.
- Reconstruct the execution flow
  - Identify main entry points triggering the issue.
  - Trace function calls, class interactions, and sequences of events.
  - Identify potential breakpoints causing the issue.
  Important: Keep the reconstructed flow focused on the problem, avoiding irrelevant details.

## Step 4: Locate Areas for Modification
- Locate specific files, functions, or lines of code requiring changes or containing critical information for resolving the issue.
- Consider upstream and downstream dependencies that may affect or be affected by the issue.
- If applicable, identify where to introduce new fields, functions, or variables.
- Think Thoroughly: List multiple potential solutions and consider edge cases that could impact the resolution.

## Output Format for Final Results:
Your final output should list the locations requiring modification, wrapped with triple backticks ```
Each location should include the file path, class name (if applicable), function name, or line numbers, ordered by importance.
Your answer would better include about 5 files.

### Examples:
```
full_path1/file1.py
line: 10
class: MyClass1
function: my_function1

full_path2/file2.py
line: 76
function: MyClass2.my_function2

full_path3/file3.py
line: 24
line: 156
function: my_function3
```

Return just the location(s)
Note: Your thinking should be thorough and so it's fine if it's very long.
````

Figure 9: The task instruction prompt used to guide the agent's reasoning process.

C USE OF LARGE LANGUAGE MODELS

Large Language Models (LLMs) were used solely as a general-purpose writing aid in the preparation of this paper. Specifically, they were used to help polish grammar and improve the clarity of certain sentences. No LLMs were used for research ideation, experimental design, data analysis, or drawing conclusions. All substantive contributions to the research and writing were made by the authors.