GitTemporalAI: Leveraging Temporal Knowledge Graphs and LLMs for Multi-Agent Repository Intelligence

Dongsheng Luo [1], **Raju Rangaswami** [1], **Amir Rahmati** [2], **Erez Zadok**[2]

[1]Florida International University, Miami, U.S.
[2]Stony Brook University, New York, U.S.
{dluo, raju}@fiu.edu, amir.rahmati@stonybrook.edu, ezk@cs.stonybrook.edu

Abstract

Large open-source software repositories represent complex multi-agent ecosystems where developers, code, and artifacts continuously interact and evolve. However, understanding these dynamic interactions and leveraging them for automated issue resolution remains challenging. We present *GitTemporalAI*, a system that constructs a temporal knowledge graph from repository data to capture the dynamic relationships between multiple entities (developers, files, issues, and pull requests) and their evolution over time. Our system employs a multi-agent architecture consisting of an embedding agent to encode repository entities, a search agent to traverse the temporal graph, and a reasoning agent that synthesizes contextual information to answer queries. By leveraging historical context and modeling relationships between repository entities, the system provides insights into repository evolution. Evaluation on a large open-source project, PyTorch Geometric, demonstrates GitTemporalAI's ability to improve query responses, particularly for tasks involving temporal reasoning and understanding repository dynamics.

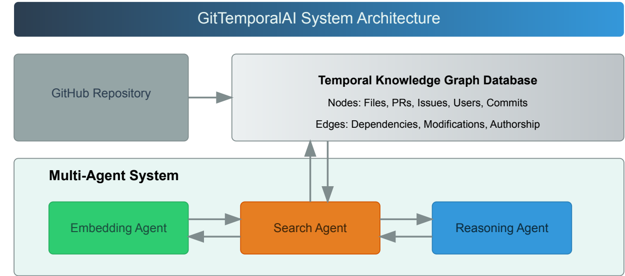

Figure 1: System architecture of GitTemporalAI. The system constructs a temporal knowledge graph from GitHub repository data to capture relationships between repository entities. Three agents work in coordination: an embedding agent for vector representations, a search agent for graph traversal and retrieval, and a reasoning agent for LLM-based response generation.

Introduction

Large open-source software repositories are complex systems that evolve through the interactions of multiple agents: developers, code, issues, and pull requests. These repositories contain valuable information about software development patterns, problem-solving approaches, and team dynamics. However, extracting and utilizing this knowledge effectively remains challenging. First, repository data is inherently temporal and interconnected. A single code change can trigger multiple effects throughout the repository ecosystem: it may introduce or resolve bug reports, spawn feature requests in the issue tracker, necessitate documentation updates, and involve collaborative efforts from several developers over time (Bird et al. 2008). Traditional approaches that analyze repository data as separate entities often miss crucial temporal and relational patterns (Thung et al. 2013). Second, while large language models (LLMs) have shown promise in code understanding and generation, they lack the ability to reason about the complex historical context of repository evolution (Feng et al. 2020). Despite their broad knowledge, LLMs face fundamental limitations in understanding repository evolution. The complex interplay between code changes, developer decisions, and task dependencies creates a dynamic context that emerges from repository-specific history rather than general patterns. These repository-specific evolutionary patterns exist outside the static code snapshots in LLMs' training data, making them inherently undiscoverable through general language modeling alone.

Consider a developer trying to understand why a previously working code component suddenly fails after a repository update. A traditional LLM can only offer generic troubleshooting suggestions based on common patterns. Understanding such issues requires analyzing the temporal evolution of the codebase - tracking when specific changes were introduced, how they propagated through different components, and which developers were involved in related modifications. This historical context is crucial for pinpointing the root cause and developing effective solutions, yet remains beyond the reach of conventional LLMs that process queries in isolation.

To address these challenges, we propose *GitTemporalAI*, a system that combines temporal knowledge graphs with LLMs to create an intelligent repository analysis framework. As shown in Figure 1, our approach integrates three

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

key components. First, we construct a temporal knowledge graph that captures both the static and dynamic relationships among repository entities, including developers, files, issues, and pull requests (Wu et al. 2022). This graph models the evolving connections within the repository, with nodes representing key entities and edges encoding interactions such as code dependencies, commit relationships, and issue references. Temporal attributes are embedded into the graph structure to track changes over time, ensuring a detailed representation of repository evolution (Cai et al. 2022, 2024).

Second, we employ a multi-agent architecture to process repository data effectively. This architecture consists of three specialized agents: An embedding agent transforms repository entities into 768-dimensional semantic embeddings using the nomic-embed-text model (Nussbaum et al. 2024), ensuring that the temporal context is preserved in the vector space. A search agent performs temporal graph traversal, leveraging the knowledge graph to identify relevant historical patterns and relationships among entities, creating a temporal context. A reasoning agent, powered by LLMs (Brown et al. 2020), synthesizes insights by combining this temporal context with advanced reasoning capabilities. Finally, we implement a progressive analysis system that processes queries in multiple rounds, progressively refining the search scope and enhancing the response quality.

We evaluate *GitTemporalAI* on PyTorch Geometric[1], a popular graph neural network library with over 500 contributors. Our results demonstrate the system's ability to leverage historical repository context to improve automated issue resolution and facilitate tasks such as developer recommendation and bug analysis.

Related Work

Conversational Agents

Recent advances in software repository mining have sparked interest in developing intelligent agents to make repository data more accessible and actionable. Several studies have explored different approaches to this challenge: MSR-Bot (Abdellatif, Badran, and Shihab 2020) pioneered the use of natural language interfaces for repository querying but was limited to predefined patterns and intents. Repairnator (Urli et al. 2018) demonstrated autonomous bug fixing through repository mining. More recent work has explored using LLMs for repository-related tasks, though primarily focusing on code-centric applications like code generation (Tian et al. 2023), code review (Tufano et al. 2022), and bug fixing (Fan et al. 2023). However, these approaches have generally focused on specific repository tasks rather than providing comprehensive repository intelligence. Additionally, existing solutions typically operate as single agents with limited ability to reason about temporal relationships and complex interactions between repository entities. Our work extends this line of research by introducing a multi-agent architecture that combines temporal knowledge graphs with LLMs, enabling more sophisticated analysis of repository evolution patterns and interactions between developers, code, and artifacts over time.

[1] *https://github.com/pyg-team/pytorch_geometric*

Knowledge Graphs in Software Engineering

Knowledge graphs have gained significant attention in software engineering for their ability to capture complex relationships in software systems. Consider a typical scenario in the PyTorch Geometric repository: a performance bug is reported in a graph convolution operation, affecting multiple dependent modules. Early work by Li et al. (2017) constructed knowledge graphs from open-source projects that could represent this as static relationships between the bug report, affected code files, and developer communications, though their approach lacked integration with modern AI techniques. More recent approaches have applied knowledge graphs to specific software engineering tasks: Du et al. (2018) used knowledge graphs for tracking vulnerability relationships in software components, while Zhang et al. (2020) employed them for bug localization through code-knowledge graph attention mechanisms. Han et al. (2018) demonstrated knowledge graph embedding techniques for reasoning about software weaknesses. While these approaches show the potential of knowledge graphs in software engineering, they typically focus on narrow applications and do not address the broader challenge of modeling temporal evolution in software repositories. In contrast, our work combines temporal knowledge graphs with a multi-agent LLM to capture and reason about the dynamic nature of software development, enabling more sophisticated analysis of repository evolution and interactions between various software entities over time.

Methods

Temporal Knowledge Graph Construction

The graph contains five types of nodes: code chunks, pull requests, commits, issues, and users. Code chunk nodes $v_c \in \mathcal{V}$ represent distinct components within Python source files, including classes, functions, methods, and module-level imports, each preserving their hierarchical structure. Pull request nodes $v_p \in \mathcal{V}$ track the lifecycle of code change proposals from creation to closure. Issue nodes $v_i \in \mathcal{V}$ store problem reports and feature requests. Commit nodes $v_m \in \mathcal{V}$ represent atomic changes to the codebase, while user nodes $v_u \in \mathcal{V}$ represent repository contributors.

The edges $\mathcal{E}$ encode five primary relationship types: code dependencies that capture import relationships between code chunks, commit relationships that link atomic changes to specific code chunks, pull request connections that associate commits within their respective PRs, issue references that connect reported problems to affected code chunks, and authorship edges that link developers to their contributions. Each edge maintains temporal attributes to track when these relationships were established or modified.

Semantic Embedding Generation

To enable semantic search and analysis over the knowledge graph, we develop an embedding strategy that captures code semantics, contextual relationships, and temporal evolution. We use the nomic-embed-text model (Nussbaum et al. 2024) to generate 768-dimensional embeddings for each node in the graph.

For code chunk nodes, we generate embeddings from a structured text representation that includes several components in sequence: metadata about the chunk (file path, chunk type, and name), defined symbols within the chunk's scope, dependency information showing import relationships with other modules, the actual code content of the chunk, and finally any relevant historical modifications that affected this specific chunk of code.

For pull request nodes, we create embeddings from the pull request title, description, and associated commits including both commit messages and specific file changes. For issue nodes, we combine the issue title, description, and any explicit references to code chunks or files mentioned in the issue text.

The resulting embeddings form a semantic space that preserves both code relationships and temporal evolution, enabling efficient similarity computations for search and analysis tasks.

Progressive Multi-Round Analysis System

To effectively leverage the temporal knowledge graph and semantic embeddings, we develop a progressive analysis system that processes queries through three rounds of increasing specificity (Yang et al. 2024). This approach enables efficient processing of large repositories while maintaining high response accuracy. The system coordinates a search agent for knowledge graph traversal and a reasoning agent for result synthesis, with round-specific objectives guiding their interaction.

Our system operates through three progressive rounds, each with specific search strategies and objectives:

- **Round 1 (Initial Scanning)** begins with the search agent performing broad retrieval using the knowledge graph embeddings to identify file chunks, issues, and pull requests that match the query. The search agent returns a candidate set. The reasoning agent then analyzes these candidates to select 5 most relevant items by evaluating their metadata and descriptions, establishing initial directions for deeper investigation.
- **Round 2 (Focused Analysis)** narrows the search scope based on Round 1 findings. The search agent uses the embeddings and node metadata to identify closely related content. For file nodes, this includes examining the fine-grained code chunks and their associated metadata such as modification history and import relationships. For pull requests, the agent examines the commit history, changed files, and specific code modifications. For issues, it considers file references and issue descriptions. This semantic similarity-based search helps establish relationships between different repository entities for deeper investigation."
- **Round 3 (Deep Investigation)** accesses the complete content and context of the most relevant items. For code files, this includes the full source code, chunked into logical components (classes, functions, methods), along with their modification history. For pull requests, it examines the complete commit history with line-by-line changes. For issues, it analyzes the full description and referenced files. The reasoning agent then generates responses using this comprehensive repository context, including specific code references, relevant changes, and associated discussions.

Table 1: Node Types and Their Key Attributes in the Knowledge Graph

Node Type	Key Attributes	Count
Source Files	Path, symbols, in/out degree	12,486
Pull Requests	Title, state, timestamps	9,607
Commits	Hash, message, timestamp	27,986
Issues	Title, state, timestamps	3,561
Users	Contribution stats, roles	2,854

Table 2: Temporal Characteristics of Key Events

Event Type	Temporal Information
Code Changes	Commit timestamps, sequential ordering
Issue Lifecycle	Creation, updates, resolution times
PR Process	Creation, review, merge timestamps
Dependencies	Import relationship evolution
User Activity	Contribution patterns over time

The system progressively enhances queries by incorporating key terms and identifiers from relevant results in previous rounds. This refinement process helps focus the search while maintaining important contextual information.

Result Integration and Memory Management Results from each round are integrated using a priority-based merging strategy that adds new findings first while selectively retaining relevant results from previous rounds. The system enforces a round-specific limit on retained results: up to 15 results from Round 2 and 5 from Round 3, preventing information overload while maintaining context. A shared memory structure enables coordination between the embedding agent, search agent, and reasoning agent by maintaining the sequence of queries, retrieved contexts, and generated responses. This integration approach allows GitTemporalAI to refine its focus while preserving relevant historical and contextual information from earlier exploration phases.

Experiment

Dataset

We evaluate GitTemporalAI on PyTorch Geometric, a popular open-source graph neural network library. Table 1 summarizes the primary node types and their key characteristics in our graph. The edge distribution in Table 2 shows how our embedding approach preserves various temporal aspects of repository events, enabling our progressive analysis system to reason about temporal relationships effectively.

Evaluation Setup

We evaluate GitTemporalAI through 30 queries related to PyTorch Geometric, spanning five key dimensions of software repository analysis: temporal evolution, cross-component interactions, bug and issue patterns, API usage,

and performance characteristics. These queries examine different abstraction levels, from specific implementation details to broader architectural patterns, providing comprehensive coverage of both technical depth and historical context.

- **Temporal Analysis (6 queries):** Questions tracking the evolution of components over time, such as changes to implementations, documentation, and feature sets.
- **Cross-Component Relationships (6 queries):** Queries examining dependencies and interactions between different modules within the repository.
- **Bug and Issue Resolution (6 queries):** Questions about how specific bugs, compatibility issues, and user-reported problems were addressed.
- **API Usage Patterns (6 queries):** Queries investigating API changes, common pitfalls, and guidelines for using different components.
- **Performance Optimization (6 queries):** Questions about computational bottlenecks, scalability improvements, and efficiency enhancements.

The evaluation compares two configurations:

- **Baseline LLM:** GPT-4o with standard prompting, without access to repository information
- **GitTemporalAI:** Our full system incorporating temporal knowledge graph, semantic embeddings, and progressive multi-round analysis

Evaluation Metrics To ensure unbiased evaluation, we employe an independent GPT-4o model as an expert evaluator to assess the responses from both systems. Each response pair is evaluated on four key dimensions as follows: (1) Technical Accuracy & Depth: Correctness of technical details and depth of explanation. (2) Historical Context:Effective use of repository history and temporal information. (3) Evidence & References: Citation of specific commits, issues, or code changes. (4) Completeness: Overall coverage of the query's requirements.

For each query, the evaluator assign scores on a 1–10 scale for both systems and provided detailed reasoning for the assessment. This evaluation approach allows us to quantitatively compare system performance while capturing qualitative insights about their strengths and limitations.

Results

Our evaluation of GitTemporalAI demonstrates the effectiveness of combining temporal knowledge graphs with a multi-agent architecture for repository intelligence. The system is evaluated against a baseline large language model that lacked access to temporal repository information. Figure 2 summarizes the overall performance comparison.

Analysis of performance by query type revealed varying effectiveness of the multi-agent architecture, as shown in Table 3. While the baseline model showed competitive performance in API usage queries, it was generally less effective across other categories. In contrast, GitTemporalAI demonstrated consistent performance across different query types, with powerful results in temporal and bug-related queries.

This superior performance stems from our system's rich historical representation, which tracks detailed line-level changes and commit contexts in the temporal knowledge graph. The comprehensive issue-to-code mapping approach further enhances bug-related query performance by maintaining explicit relationships between issue reports and affected code components. These results demonstrate the effectiveness of our multi-agent architecture, which shows particular strength in queries requiring temporal understanding and bug analysis. GitTemporalAI's consistent performance across all categories, compared to the baseline's variable effectiveness, validates its robust approach to repository intelligence tasks.

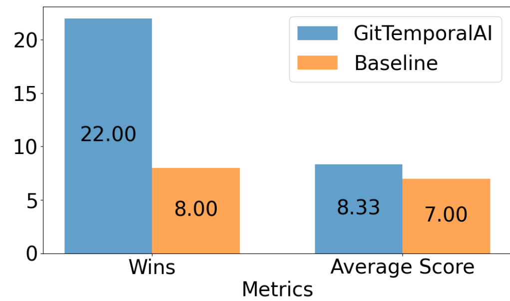

Figure 2: Performance Comparison

Table 3: Performance Comparison by Query Type

Query Type	**GitTemporalAI**	**Baseline**
Temporal	8.33	7.17
Cross-Component	8.50	7.33
Bug Issue	8.50	6.33
API Usage	8.00	7.33
Performance	8.33	6.83

Conclusion

In this work, we have presented GitTemporalAI, a system that combines temporal knowledge graphs and LLMs to analyze and leverage the complex dynamics of multi-agent interactions in software repositories. By constructing a temporal knowledge graph to model evolving relationships among repository entities and employing a progressive multi-agent architecture, our system demonstrates the ability to address diverse repository intelligence tasks effectively.

GitTemporalAI's current implementation faces computational limitations, particularly in embedding generation for large repositories with extensive histories. Additionally, the system's focus on Python repositories leaves room for expansion to multilingual codebases. Future work includes 1) improving the efficiency of GitTemporal and 2) extending to support more complex repository ecosystems and exploring its applications in other domains, such as multi-agent collaborations and dynamic knowledge systems. By bridging the gap between structured temporal data and the reasoning capabilities of LLMs, GitTemporalAI offers a promising direction for advancing intelligent software repository analysis.

References

Abdellatif, A.; Badran, K.; and Shihab, E. 2020. MSRBot: Using bots to answer questions from software repositories. *Empirical Software Engineering*, 25: 1834–1863.

Bird, C.; Pattison, D.; D'Souza, R.; Filkov, V.; and Devanbu, P. 2008. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 24–35.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781713829546.

Cai, B.; Xiang, Y.; Gao, L.; Zhang, H.; Li, Y.; and Li, J. 2022. Temporal knowledge graph completion: A survey. *arXiv preprint arXiv:2201.08236*.

Cai, L.; Mao, X.; Zhou, Y.; Long, Z.; Wu, C.; and Lan, M. 2024. A Survey on Temporal Knowledge Graph: Representation Learning and Applications. *arXiv preprint arXiv:2403.04782*.

Du, D.; Ren, X.; Wu, Y.; Chen, J.; Ye, W.; Sun, J.; Xi, X.; Gao, Q.; and Zhang, S. 2018. Refining traceability links between vulnerability and software component in a vulnerability knowledge graph. In *Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18*, 33–49. Springer.

Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; and Tan, S. H. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1469–1481. IEEE.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547.

Han, Z.; Li, X.; Liu, H.; Xing, Z.; and Feng, Z. 2018. Deepweak: Reasoning common software weaknesses via knowledge graph embedding. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 456–466. IEEE.

Li, W.; Wang, J.; Lin, Z.; Zhao, J.; Zou, Y.; and Xie, B. 2017. Software knowledge graph building method for open source project [J]. *Journal of Frontiers of Computer Science and Technology*, 11(006): 851–862.

Nussbaum, Z.; Morris, J. X.; Duderstadt, B.; and Mulyar, A. 2024. Nomic embed: Training a reproducible long context text embedder. *arXiv preprint arXiv:2402.01613*.

Thung, F.; Bissyande, T. F.; Lo, D.; and Jiang, L. 2013. Network structure of social coding in github. In *2013 17th European conference on software maintenance and reengineering*, 323–326. IEEE.

Tian, H.; Lu, W.; Li, T. O.; Tang, X.; Cheung, S.-C.; Klein, J.; and Bissyandé, T. F. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*.

Tufano, R.; Masiero, S.; Mastropaolo, A.; Pascarella, L.; Poshyvanyk, D.; and Bavota, G. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, 2291–2302.

Urli, S.; Yu, Z.; Seinturier, L.; and Monperrus, M. 2018. How to design a program repair bot? insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 95–104.

Wu, T.; Khan, A.; Yong, M.; Qi, G.; and Wang, M. 2022. Efficiently embedding dynamic knowledge graphs. *Knowledge-based systems*, 250: 109124.

Yang, D.; Rao, J.; Chen, K.; Guo, X.; Zhang, Y.; Yang, J.; and Zhang, Y. 2024. Im-rag: Multi-round retrieval-augmented generation through learning inner monologues. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 730–740.

Zhang, J.; Xie, R.; Ye, W.; Zhang, Y.; and Zhang, S. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, 219–229.

Appendix

Evaluation Queries

Temporal Analysis Queries

1. "How has the GCN implementation in torch_geometric.nn.conv.GCNConv evolved since its first introduction?"
2. "What were the major changes to the MessagePassing base class over time?"
3. "What are the key milestones in the development of torch_geometric.datasets?"
4. "How has the documentation for torch_geometric.utils changed?"
5. "How has the feature set of torch_geometric.utils been expanded since its inception?"
6. "Trace the evolution of graph neural network models available in PyG."

Cross-Component Analysis Queries

1. "Which modules depend on torch_geometric.nn.conv.GATConv implementation?"
2. "What's the relationship between DataLoader implementations and sampling methods?"
3. "How do the implementations of different convolutional layers interact with PyG's utils?"
4. "What impact do transforms have on the performance of dataset preprocessing?"
5. "How does torch_geometric.data interact with PyG's neural network modules?"
6. "Examine the integration of torch_geometric.transforms with DataLoader for efficient data processing."

Bug and Issue Analysis Queries

1. "What common issues were reported with the GraphSAGE implementation?"
2. "How were memory leaks in torch_geometric.data.Data addressed over time?"
3. "How have issues related to dataset loading been resolved in PyG?"
4. "What are the most frequent user-reported problems with torch_geometric.sampler modules?"
5. "Identify the resolution process for compatibility issues between different PyG versions."
6. "What solutions have been implemented to address the challenges of graph sampling in PyG?"

API Usage Queries

1. "How has the interface of torch_geometric.transforms evolved?"
2. "What breaking changes were introduced in the NeighborSampler API?"
3. "What are the common pitfalls when using torch_geometric.data.Data?"
4. "How did the Python version upgrades affect PyG API compatibility?"
5. "How has the API for torch_geometric.nn.conv changed to accommodate new graph convolution techniques?"
6. "What are the guidelines for using torch_geometric.datasets in custom applications?"

Performance Analysis Queries

1. "What optimizations were made to the cluster_gcn.py implementation?"
2. "How has the memory efficiency of PyG's basic operations improved?"
3. "What specific changes were made to enhance the scalability of large graphs in PyG?"
4. "What were the computational bottlenecks identified in PyG's Transform classes and how were they addressed?"
5. "Evaluate the performance improvements in the latest versions of PyG's sampling methods."
6. "What are the most significant performance gains achieved through updates to torch_geometric.data?"