CODE UNDERSTANDING WITH GITHUB ISSUES AND RAG

Benjamin Chou * **Wenru Hu** *

ABSTRACT

Incomplete documentation makes external code challenging to use. We propose a system that enhances documentation by leveraging large language models and insights from GitHub issues. The system prioritizes and classifies issues, retrieves relevant code, and generates context-aware documentation. Evaluations, based on LLM-generated survey scores, show improvements of 116% in README setup guidance, 32% in contextual insights, and 16% in function-level guidance. This approach simplifies onboarding and integrates solutions directly into the codebase.

1 INTRODUCTION

Many barriers come in the way of understanding another developer's code. Lack of proper documentation and README can cause issues during the setup, usage, and adaptation of another developer's code. Developers use `Github Issues` to report issues they face when working with repositories, and developers and repository maintainers can engage in conversations to solve these issues. GitHub issues provide a rich source of contextual information about the challenges, decisions, and ongoing development of a repository. Developers can reference these issues to solve their own issues.

Some of the insights from these discussions are good candidates for documentation enhancement. By integrating these insights into documentation, developers can forgo the process of (1) doing an online search to solve their issues, (2) doing a keyword search in the repositories of GitHub issues. And (3) manually find the best solution proposed in the discussion.

In this paper, we develop a pipeline using LLMs to integrate the insights from GitHub Issues into a repository's documentation.

To evaluate whether LLMs can effectively distill the content of GitHub issues into code documentation, we investigate the impact of our generated comments via an LLM survey. Our findings show that integrating GitHub issues into repository documentation can reduce the start-up cost for developers adopting unfamiliar codebases and provide valuable insights into understanding the code as well as avoiding common issues related to the code. Our code is open-sourced and can be accessed here.

2 RELATED WORK

Studies have shown methods of prioritizing GitHub issues for repository maintainers to save time. (Caddy & Treude, 2024; Dhasade et al., 2020) We acknowledge that not all insights from GitHub issues are suitable for documentation enhancement, so we adapt similar methods to (Caddy & Treude, 2024; Dhasade et al., 2020) to prioritize GitHub issues for our system. Since the motivation to prioritize GitHub issues is different, we provide insights we obtained from developing our LLM documentation enhancer that is distinct from prior work.

Addressing issues in open-source repositories often involves a critical step of code retrieval to locate relevant sections of the repository that correspond to the user's problem or query. Code retrieval techniques can be broadly categorized into dense retrieval and sparse retrieval, each serving different use cases based on the nature of the query and the structure of the codebase. The existing methods are mainly divided into two types, sparse and dense.

Sparse retrieval methods(Das & Chakraborty, 2018; Nishida et al., 2023), such as TF-IDF (Term Frequency-Inverse Document Frequency) (Spärck Jones, 1972), rely on matching keywords between the query and the codebase. These methods use a bag-of-words representation, focusing on the frequency and importance of terms within the repository. Sparse retrieval is straightforward and computationally efficient, and effective for pinpointing exact matches based on terms like error codes, variable names, or function identifiers.

Dense retrieval methods (Xu et al., 2022) rely on embedding-based representations to measure the semantic similarity between queries (e.g., a user's issue description) and the repository's code. These embeddings encode the meaning of a query or code snippet into a continuous vector space, allowing for nuanced matches that go beyond exact keyword

overlaps.

3 METHOD

To address the challenges of enhancing code understanding and reducing onboarding friction for developers, we propose a systematic methodology that integrates insights from GitHub issues into code documentation using large language models (LLMs). Figure 1 is a framework overview.

3.1 Github Issue Prioritization

We rank GitHub issues to identify those most relevant for documentation enhancement. This process involves data collection, scoring, and ranking.

Issues are fetched using the GitHub GraphQL API, including metadata such as titles, labels, comments, reactions, and author information. Contributors and the repository owner are identified, while bot accounts are excluded. To improve efficiency, issue data is cached locally to avoid redundant queries.

Following (Caddy & Treude, 2024), we only select closed issues that have been assigned a default GitHub issue label. Each issue is scored based on several factors. Issues authored by the repository owner or contributors receive higher scores. Additional weight is given to issues with more reactions, descriptive titles, or labels indicating documentation relevance, such as "documentation" or "question". Issues like "bugs" are discarded, since these are better addressed by repository maintainers updating the codebase. Issues containing explanatory comments, such as code snippets or detailed descriptions, are prioritized. We achieve this by doing a keyword search for common terms like "how do i", "works" ect. Conversely, issues with exclusionary labels like "spam" or "invalid," or authored by bots, are excluded.

After scoring, issues are ranked in descending order of priority. This ensures that only actionable and relevant issues are used for documentation enhancement, avoiding unresolved or irrelevant cases. This prioritization framework supports the goal of reducing onboarding challenges and improving code understanding.

3.2 Issue Classification

To identify how to process the issues, we employ an LLM to categorize each prioritized issue into one of four distinct types:

- **C**ategory 1: Issues where code is provided and the repository is used as a tool. In these issues, users treat the repository as a tool or framework that they are utilizing. They provide code examples that highlight their problem or show how they are using the repository. The user's goal is typically to fix a usage error, clarify how to use a function, or troubleshoot behavior they observe when using the repository as intended. For example, they encountered situations like misusing an API or getting unexpected results from a method or class provided by the repository.

- **C**ategory 2: Issues where code is provided and the user is modifying or implementing repository source code. These issues occur when users are modifying or extending the repository's source code. Their focus is not on how to use the repository, but on altering its behavior or adding new functionality. In most circumstances, they are trying to debug custom changes that interact with existing repository code.

- **C**ategory 3: Issues where no code is provided and the problem is general to the repository. The issue is general to the repository and often involves environment setting up, configuration, or understanding how to use the repository broadly. These issues may involve unclear documentation, installation problems, or general guidance.

- **C**ategory 4: Issues where no code is provided, but the problem relates to a specific code block in the repository. The issue might arise from confusion about how a particular function or block of code works or why it behaves unexpectedly in certain cases.

This classification allows us to tailor our retrieval and comment generation strategy to the specific context of each issue.

3.3 AST-based Code Segmentation

The abstract syntax tree (AST) (Tian et al., 2022; Welty, 1997) is a structured representation of source code that captures its syntax and structure in a tree form. Each node in the AST represents a construct from the code, such as a variable declaration, loop, or function call. By parsing the code into an AST, it becomes easier to understand the hierarchical and syntactic relationships within the code. To perform code search effectively, the code needs to be segmented into meaningful units that can be independently analyzed or compared, which divides the source code into smaller, logical pieces, such as functions, methods, or statements.

3.4 Context-Driven Code Retrieval

To ensure high precision in locating the relevant parts of the repository, we employ different retrieval methods based on the issue category. Each category requires a tailored approach to locate the relevant information or provide the necessary guidance.

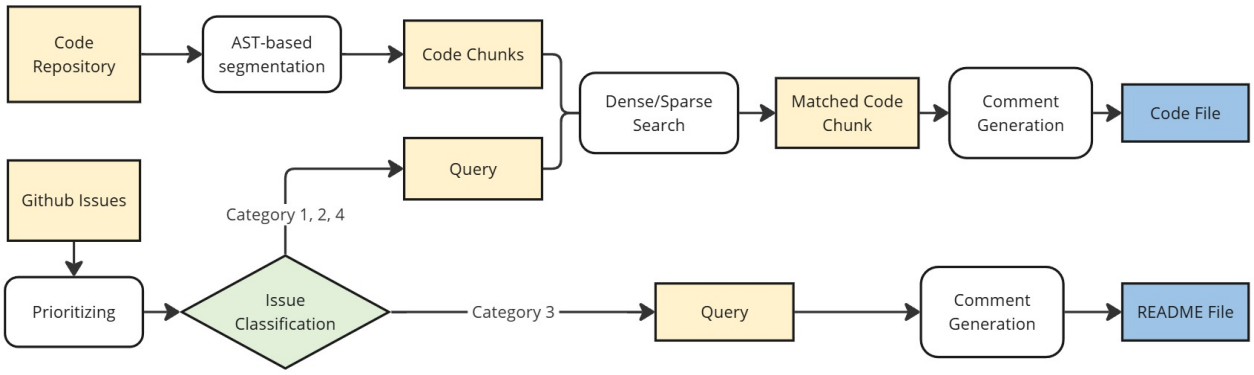

Figure 1. Overview of the proposed pipeline for integrating GitHub issues into documentation.

For Category 1 issues, where users are utilizing the repository as a tool and provide specific code snippets, embedding-based dense retrieval is the most effective method. Dense retrieval methods use vector embeddings to capture semantic similarity, making them ideal for finding related code snippets or documentation sections that match the user's input. This approach allows us to locate functions, classes, or usage examples in the repository that align closely with the user's provided code or error description.

For Category 2 issues, where users are modifying or implementing changes in the repository source code, sparse retrieval methods (e.g., TF-IDF or keyword-based search) are more suitable. Sparse retrieval focuses on exact or near-exact keyword matches, making it ideal for identifying specific code references, class names, or methods mentioned in the user's query. Since users in this category are typically working with internal details of the repository, they often use technical terms or directly reference parts of the codebase, which aligns well with sparse retrieval techniques.

For Category 3 issues, where users do not provide code and their problems are general to the repository, the focus should be on enhancing the repository's README file and documentation. These issues often stem from unclear setup instructions, insufficient guidance, or gaps in the repository's documentation. By improving the README and associated documentation, similar issues can be preemptively addressed for other users.

For Category 4 issues, where users do not provide code but point to specific repository functions or features, embedding-based dense retrieval is again the most appropriate method. Users often describe their issue in natural language, referencing specific functions or behaviors. Embedding-based retrieval excels at matching such descriptions with relevant code sections or documentation, even when the terminology used by the user is not exact.

	Code Provided	Retrieval Method	File to Modify
C1	✓	Dense	Code File
C2	✓	Sparse	Code File
C3	✗	/	README
C4	✗	Dense	Code File

Table 1. Issue Categorization Matrix for Retrieval and Modification Contexts: The table summarizes different categories of user issues and the corresponding retrieval methods and file modifications. Category 1 (C1) refers to issues where users provide code snippets and require dense retrieval methods to locate relevant code examples. Category 2 (C2) involves users modifying repository code, requiring sparse retrieval methods like TF-IDF for finding specific code references. Category 3 (C3) relates to general issues without code snippets, where improvements to the README file are necessary. Category 4 (C4) involves issues where users describe problems related to specific repository functions or features, requiring dense retrieval methods to match user descriptions to relevant code.

3.5 Comment Generation

The process begins by determining whether the issue has a solution in followup replies. If no solution is provided in the post or related discussions, the issue is skipped to avoid introducing unsupported or speculative comments.

For issues with solutions, a concise summary of the solution is created. This summary synthesizes key information from the issue discussion, focusing on the resolution or improvement relevant to the code context. The extraction process balances detail and brevity, aiming to preserve clarity while avoiding unnecessary verbosity.

To generate the final comment, the extracted solution summary is combined with the corresponding code context. A carefully designed prompt is provided to the LLM to guide its response. The prompt template is as follows in Listing 1:

Listing 1. Prompt template for generating a code comment based on code snippet and issue context.

```
You are an experienced software engineer tasked with
    reviewing the following Python code.
Your goal is to identify potential issues and provide
    actionable recommendations to improve it.

The objective is to generate a code comment that:
- Helps developers understand, use, or improve the
    code effectively.
- Is relevant and insightful based on the provided
    code snippet.

### Code Snippet:
<code>
{ additional_context }
</code>

### Context from Related GitHub Issue:
- **Issue Title**:
  < issue_title >
  </ issue_title >

- **Issue Content**:
  <issue_content>
  </issue_content>

### Relevant Discussion Summaries:
{relevant_summaries}

**Your Task**:
Write a clear, concise, and actionable code comment
    with the following goals:
1. Focus primarily on the provided code snippet.
    Identify potential issues, assumptions, or edge
    cases.
2. Provide non-obvious insights directly relevant to
    the code, such as functionality, design
    decisions, or risks developers may overlook.
3. Suggest specific improvements to make the code
    more robust, maintainable, or efficient.
4. Reference the issue and discussion **only if it
    enhances the understanding or application of the
    code**.

The comment should be **highly relevant to the code
    **, informed by the issue discussion but not
    driven by it. Avoid generic or redundant
    suggestions.
```

This template helps ensure that the generated code comments are comprehensive, actionable, and contextually relevant to the issue at hand.

4 EVALUATION

We evaluated the system by comparing its outputs, which incorporate GitHub issue descriptions, against a baseline that uses an LLM to predict potential issues based solely on the code. This ensures alignment between the baseline and the proposed system in leveraging LLM capabilities. The evaluation focused on three criteria where GitHub issues are expected to improve performance: surfacing non-obvious insights, reducing setup hassle, and avoiding potential issues. GPT-4 was used as a reviewer to score and rank outputs.

```
prompt = f"""
You are evaluating comments generated for a {
    evaluation_type }.

Function/README Content:
{function_body}

Baseline Comment:
{baseline_comment}

Proposed System Comment:
{proposed_comment}

Evaluate both comments on the following criteria:
1. Non-Obvious Insight: Does the comment provide
    useful information not obvious from the content?
2. Reduction of Setup Hassle: For README updates,
    does it make setup easier? For function comments,
    does it clarify the code purpose?
3. Helps Avoid Potential Issues: Does the comment
    highlight potential problems and provide
    guidance to prevent or address them?

Rate each on a scale of 1 to 5 and provide an overall
    ranking (1 = better, 2 = worse).

Output your response in this format:
Ratings:
Proposed System: Non-Obvious Insight: X, Reduction of
    Setup Hassle: X, Helps Avoid Potential Issues:
    X
Baseline: Non-Obvious Insight: X, Reduction of Setup
    Hassle: X, Helps Avoid Potential Issues: X
Ranking:
Proposed System: X
Baseline: X
"""
```

Figure 2. Using GPT-4 to evaluate the baseline and proposed comments based on updated criteria.

4.1 Experiment Design

For each code block and its corresponding GitHub issue, two outputs were generated:

- **Proposed System:** Generated using both the code block and the GitHub issue description.
- **Baseline:** Generated using only the code block.

GPT-4 scored the outputs on a scale of 1 to 5 for the follow-

ing criteria:

- **Non-Obvious Insight:** Information that cannot be inferred directly from the code.
- **Reduction of Setup Hassle:** Effectiveness in minimizing barriers to setting up or using the repository.
- **Avoiding Potential Issues:** Anticipation of errors or challenges users might encounter.

The detailed prompt used for the LLM survey is shown in Figure 2

4.2 Results

We evaluated our system by comparing it to a baseline that generates documentation using only code snippets without GitHub issue descriptions. Our evaluation focused on contextual relevance, surfacing non-obvious insights, and reducing setup challenges.

Our system performed well for README-related issues, as shown in Table 2. It reduced setup challenges significantly, achieving a score of 4.50 compared to the baseline's 2.08. It also provided better contextual relevance and surfaced insights not evident from the code alone. These results highlight our system's strength in integrating GitHub issues to enhance documentation and improve developer onboarding.

For function-related issues, our system showed less improvement. Table 3 shows that it performed only slightly better than the baseline. This underperformance occurred because our code matching algorithm struggled to pair issues with specific function implementations accurately. While our system effectively aligns GitHub issues with broader documentation needs, it does not fully capture the granular context required for function-level improvements.

Our results demonstrate that the system excels in enhancing README documentation but requires improvements in its code matching algorithm to address function-related issues more effectively.

Category	Baseline	Proposed
Non-Obvious Insight	3.17	4.17
Reduction of Setup Hassle	2.08	4.50
Helps Avoid Potential Issues	3.75	4.83

Table 2. Evaluation results for README-related issues.

4.3 Qualitative

We provide qualitative examples to illustrate the system's functionality in synthesizing solutions and recommendations from GitHub issues.

Category	Baseline	Proposed
Non-Obvious Insight	4.25	3.75
Reduction of Setup Hassle	3.17	3.67
Helps Avoid Potential Issues	4.58	4.50

Table 3. Evaluation results for function-related issues.

In the first example, the system resolves a CUDA out-of-memory issue in the A-LLMRec project. The steps for resolution are shown in Listing 2:

Listing 2. Steps to resolve CUDA out-of-memory issue in A-LLMRec project.

```
# If you encounter CUDA out of memory issues during
    Stage 2, consider:
# 1. Reducing batch size or limiting item title
    lengths (dataset-dependent).
# 2. Verifying dataset thresholds (e.g., user/item
    counts in preprocessing).
# 3. Installing specific library versions as needed (
    e.g., transformers==4.32.1).
# 4. Use _libgcc_mutex==0.1 to resolve environment
    setup issues.
```

The system extracts steps to resolve memory issues and configure the environment.

In the second example, the system addresses an installation error in the Auto-GPT project. The solution is provided in Listing 3:

Listing 3. Solution to installation error in Auto-GPT project.

```
# Issue: Installation error with 'googlesearch-
    python' while trying to run 'pip install -r
    requirements.txt' in the Auto-GPT project.
# Solutions:
# 1. Clone the 'googlesearch' repository:
#    git clone https://github.com/Nv7-GitHub/
    googlesearch.git
# 2. Create a new 'requirements.txt' file that
    includes the line:
#    beautifulsoup4==4.9.3
# 3. Replace the original 'googlesearch-python' entry
    in the Auto-GPT 'requirements.txt' with:
#    git+https://github.com/Nv7-GitHub/googlesearch.
    git
# 4. Run 'pip install -r requirements.txt' in the
    Auto-GPT directory.
```

The system identifies installation steps and suggests verifying dependencies and using specific file formats to prevent errors.

In the third example, the system suggests improvements to the yt-dlp repository for a URL extractor issue. The

suggested improvements are detailed in Listing 4:

Listing 4. Improvements for yt-dlp URL extractor.

```
# The current implementation of the URL extractor in
    yt-dlp tool seems to have issues with certain
    websites,
# like artmuseum.pl, where it's unable to extract the
    correct video URL. This results in an
    UnsupportedError.
# Improvements:
# - Enhance the URL extractor to handle encoded
    URLs.
# - Add error handling to provide more descriptive
    error messages.
# - Add support for more types of webpages or video
    players.
```

The system suggests handling encoded URLs, adding error messages, and supporting more webpage types.

These examples demonstrate how the system identifies resolution steps, provides guidance for preventing issues, and suggests improvements for repository maintainers.

5 Conclusion

This work presents a system that integrates insights from GitHub issues into code documentation using large language models. The system prioritizes and classifies issues, retrieves relevant code contexts, and generates concise comments to improve repository documentation. By using GitHub issues, the system provides developers with direct guidance, reducing the need for external searches or manual issue resolution.

The evaluation shows that the system improves README-related documentation by increasing setup hassle scores by 116%, non-obvious insights by 32%, and function-related setup hassle scores by 16%. These results indicate that the system effectively integrates GitHub issue insights into documentation. Future work will focus on refining function-specific retrieval methods to further enhance performance.

This system provides a structured approach for integrating GitHub issues into documentation workflows, demonstrating measurable benefits for developers working with unfamiliar repositories.

6 Future Work

Future work will focus on addressing the system's varying performance across different repositories. The current implementation shows differences in effectiveness depending on the repository, indicating the need for more adaptability. To improve this, we plan to implement repository-specific profiling to optimize performance based on repository characteristics. We will also add features to generate graphs for visualizing performance metrics and include hyperlinks within the code analysis to provide access to relevant resources or documentation. These changes aim to improve the system's reliability and usability.

References

Caddy, J. and Treude, C. Prioritising GitHub Priority Labels. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 52–55, Porto de Galinhas Brazil, July 2024. ACM. ISBN 9798400706752. doi: 10.1145/3663533.3664041. URL https://dl.acm.org/doi/10.1145/3663533.3664041.

Das, B. and Chakraborty, S. An improved text sentiment classification model using TF-IDF and next word negation. *CoRR*, abs/1806.06407, 2018. URL http://arxiv.org/abs/1806.06407.

Dhasade, A. B., Venigalla, A. S. M., and Chimalakonda, S. Towards Prioritizing GitHub Issues. In *Proceedings of the 13th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, pp. 1–5, Jabalpur India, February 2020. ACM. ISBN 978-1-4503-7594-8. doi: 10.1145/3385032.3385052. URL https://dl.acm.org/doi/10.1145/3385032.3385052.

Nishida, K., Yoshinaga, N., and Nishida, K. Sparse neural retrieval model for efficient cross-domain retrieval-based question answering. In *Proceedings of the 37th Pacific Asia Conference on Language, Information and Computation*, pp. 819–830, 2023.

Spärck Jones, K. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972. doi: 10.1108/eb026526. URL https://doi.org/10.1108/eb026526.

Tian, Z., Tian, B., and Lv, J. Combining ast segmentation and deep semantic extraction for function level vulnerability detection. In *The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, pp. 93–100. Springer, 2022.

Welty, C. A. Augmenting abstract syntax trees for program understanding. In *Proceedings 12th IEEE International Conference Automated Software Engineering*, pp. 126–133. IEEE, 1997.

Xu, C., Guo, D., Duan, N., and McAuley, J. LaPraDoR: Unsupervised pretrained dense retriever for zero-shot text retrieval. In *ACL 2022 (Findings)*, 2022.