Enhancing LLMs Interactions for Python: A Smart API Framework for Extracting and Utilizing Semantic Code Information

1st Nouman Ahmad*
School of Software Engineering
Northeastern University
Shenyang, China
2027017@stu.neu.edu.cn

2nd Changsheng Zhang
School of Software Engineering
Northeastern University
Shenyang, China
zhangchangsheng@mail.neu.edu

***Abstract*— In order to engage with large language models (LLMs) in a meaningful way, it is necessary to create prompts that are both instructive and precise. However, especially when working with complicated codebases, basic prompts frequently fall short in providing adequate context or extracting useful insights. In order to overcome this difficulty, we introduce a brand-new system that uses FastAPI to automatically extract semantic data from Python codebases. In order to recognize import statements, custom functions, and class declarations, as well as how they are used within modules and classes, our system uses libcst to parse and analyze Python source files. Furthermore, the framework facilitates extraction from particular branches, commits, or tags by supporting Git-based version control. We create distinctive, context-rich prompts for LLMs like OpenAI's GPT models by fusing these capabilities. By producing structured JSON answers, the API makes it easier to integrate with open-source LLM systems like ChatGPT. In addition to improving the caliber of interactions with LLMs, this method gives developers the ability to effectively query and comprehend intricate codebases. Lastly, we curate a dataset of Python repositories from GitHub based on size and activity criteria, and train a GPT-4 based model evaluated using BERTScore, BLEU, and ROUGE, achieving research-acceptable metrics.**

Keywords*— *Prompt Engineering, Semantic Code Analysis, LLM Interactions

I. Introduction

The increasing usage of large language models (LLMs) for tasks like code creation and debugging shows how important it is for humans and AI systems to communicate effectively. [1, 2] Developing accurate and contextually rich prompts is essential to achieving significant outcomes from these models. However, manually crafting these prompts can be laborious and prone to errors, especially when working with intricate codebases that span multiple classes, modules, and dependencies.

A recently proposed benchmark system called CrossCodeEval [3] was created to assess code-generation models utilizing a variety of real-world repositories that are permissively licensed and span four popular programming languages: Python, Java, TypeScript, and C#. CrossCodeEval uses real, open-source codebases to enable a more realistic evaluation of model performance in real-world situations, in contrast to artificial or toy datasets. This method's primary drawback is its sole emphasis on static analysis, which evaluates code generation within individual files without taking into account cross-file dependencies, project-wide context, or dynamic interactions (such as runtime behavior or multi-module imports). Although this makes evaluation easier, it ignores important issues in actual software development, including preserving consistency across sizable codebases or resolving external references, which could lead to an overestimation of the model's capabilities for end-to-end development activities.

In Generative AI and Internet of Things applications, prompt engineering has been very helpful in enhancing task-specific performance and allowing for more exact control over model outputs [4]. Well-crafted prompts aid AI in interpreting sensor data, producing alerts, and effectively automating replies in IoT systems. Effective prompting also improves text generation, code synthesis, and the creation of innovative material in generative AI [5]. Crafting comprehensive, context-rich prompts that take into consideration intricate, multi-step activities or domain-specific subtleties, however, continues to present difficulties [3]. Many prompts are still unduly basic and do not fully utilize the capabilities of sophisticated models, especially in dynamic IoT contexts where real-time adaptability is essential. Although methods such as few-shot learning and chain-of-thought prompting have produced better outcomes, robustness in edge circumstances is still constrained by the absence of standardized approaches for granular prompt design.

We propose a framework based on FastAPI that automatically extracts semantic information from Python codebases as an innovative solution to this gap. Our solution makes use of libcst's [1] capabilities to scan source code, locate, and organize crucial elements such as function definitions, import declarations, and class hierarchies. Additionally, by supporting Git repositories using Pythongit [2], it enables users to mark branches, commits, or tags for targeted analysis. This extracted data is then used to generate the highly contextualized LLM prompts, ensuring that the models have all the information, they require to provide accurate and relevant responses.

Because our system's results are released in JSON format (Figure 1), developers can easily integrate with current solutions

[1] https://github.com/Instagram/LibCST
[2] https://github.com/gitpython-developers/GitPython

or directly interact with open-source LLM platforms like ChatGPT. By making it easier to extract and use semantic code information, this tool improves developer efficiency and understanding of complicated codebases while also enhancing the caliber of interactions with LLMs.

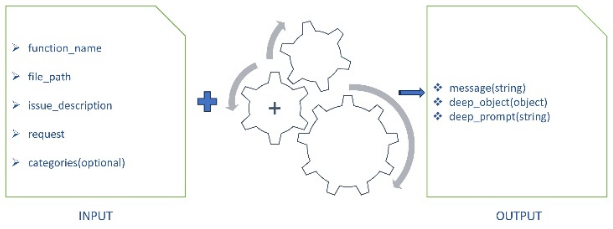

Figure 1. Processing of input format for API to get the output in both JSON format

II. Literature Review

Code language models [6] like Codex [7], CodeGen [8], and StarCoder [9] have shown that they can greatly increase developer productivity, especially when it comes to code completion jobs. These models produce syntactically accurate and contextually relevant recommendations by utilizing extensive pretraining on a variety of codebases. Researchers have established a number of evaluation benchmarks (e.g., [6,10,11,12]) to gauge the models' performance. These benchmarks examine the models' ability to finish code snippets based on the immediate context within a single file. Although these benchmarks offer a helpful place to start, they frequently function on oversimplified assumptions that fail to adequately represent the complexities of actual software development.

In reality, contemporary software projects are organized over a number of interdependent files, and code creation and comprehension heavily rely on cross-file dependencies like shared variables, imported functions, and inherited classes. Nevertheless, the majority of current benchmarks do not take these dependencies into consideration, assessing models separately instead of in the context of the larger repository. Because real-world code completion sometimes necessitates comprehending and utilizing relationships across files, this constraint results in an incomplete evaluation of a model's actual capabilities. In order to adequately represent the difficulties developers encounter in typical programming settings, more realistic assessment frameworks that take multi-file dependencies into account are urgently needed.

III. Methodology

A. *Function Extraction*

Initially, the Abstract Syntax Tree (AST) module is used to extract the target function from a Python file [13]. This module parses the function structure to identify relevant metadata, including variables, docstrings, comments, and references to other functions or classes within the same file. Because AST can offer a structured representation of code and is reliable in handling Python's syntactic constructs utilizing Graph Neural Networks (GNN) [14], we recommend using it.

B. *Cross-Module Dependency Analysis*

The issue of tracking dependencies between modules is resolved by libcst. This library allows the system to locate the definitions of referenced functions or classes in external files or modules, making it simpler to identify their source. It is recommended to use libcst because it provides a visual representation of a syntax tree, which is essential for analyzing cross-module dependencies in intricate projects.

C. *Semantic Information Aggregation*

In addition to syntactic analysis, we gather semantic information like file size, module structure, and contextual usage patterns. This step is necessary to increase the generated response's accuracy. To ensure that the request is suitable for processing by external models, we recommend including semantic data in scenarios where contextual information is essential.

D. *API Request Construction*

The collected data is used to construct an API request. If an OpenAI API key is provided, the system forwards this request to the OpenAI API [15] for processing. Since the response is a plain string, it may also be utilized with open-source platforms like chatgpt.com or any available LLM model. This dual approach ensures flexibility in integrating the solution with both open-source and proprietary language models.

E. *Dynamic Response Handling*

The system dynamically adjusts the request to satisfy a range of user needs based on user input and preferences. The final output includes both a plain-text representation of the code and an object-based answer. By providing consumers with clarity on the contents of the created string, this dual-format output is recommended to assist users in making better judgments.

F. *User Interaction*

The solution is developed using the FAST API framework [9], which ensures scalable and efficient user interaction. This method, which is specifically made for Python scripts, addresses the challenge of manually monitoring frequently called functions and classes across modules. We recommend FASTAPI due to its outstanding performance and ease of interaction with modern web technologies. The complete prompt-creation process is shown in the Figure 2.

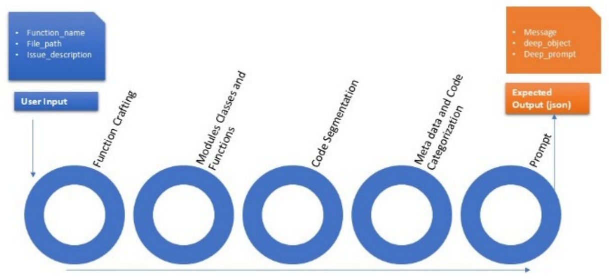

Figure 2. The workflow of crafting a prompt using the information available in a python

IV. Experiments

By taking in detailed user inputs such as the target function name, absolute file path, detailed issue description, and classified task type (data processing, API endpoints, or database activities), the system allows for accurate code search and solution development. By finding the designated function throughout the whole codebase and automatically recognizing all internal function and class calls—even those that span multiple files—it carries out deep code analysis. The development of comprehensive dependency maps that disclose cross-file linkages and execution needs is another example of this contextual understanding, giving users complete insight into the environment surrounding the code.

The system uses the text-davinci-003 model (which can be modified to use other models) to produce extremely particular solutions when an OpenAI API key is available. It does this by feeding it the code context that has been evaluated. It provides a structured JSON answer with a thorough deep_object and extensive metadata in situations where API access is not available. Semantic relationships, cross-file dependencies, execution requirements, and full function and class definitions contained within the target function are all included in this. A deep_prompt string, a streamlined, LLM-ready form of the analysis, is also provided in the response. Users can enter this string into any open-source language model for additional processing. The architecture of the system supports both automated AI processing and manual analysis operations, guaranteeing comprehensive code evaluation while preserving flexibility in solution delivery.

Code segments with their corresponding metadata, dependency graphs, and runtime needs are all included in the deep object of the JSON output, which systematically arranges information. Regardless of file borders, developers can immediately comprehend complicated code interactions thanks to this framework. The accompanying deep prompt, which has all the necessary information in a single string prepared for efficient AI processing, acts as an optimal input for external language models. The system accommodates a range of user requirements by fusing accurate code search capabilities with diverse output forms, from instantaneous AI-generated solutions to manual code exploration, all the while keeping track of the entire codebase architecture and linkages Figure 3.

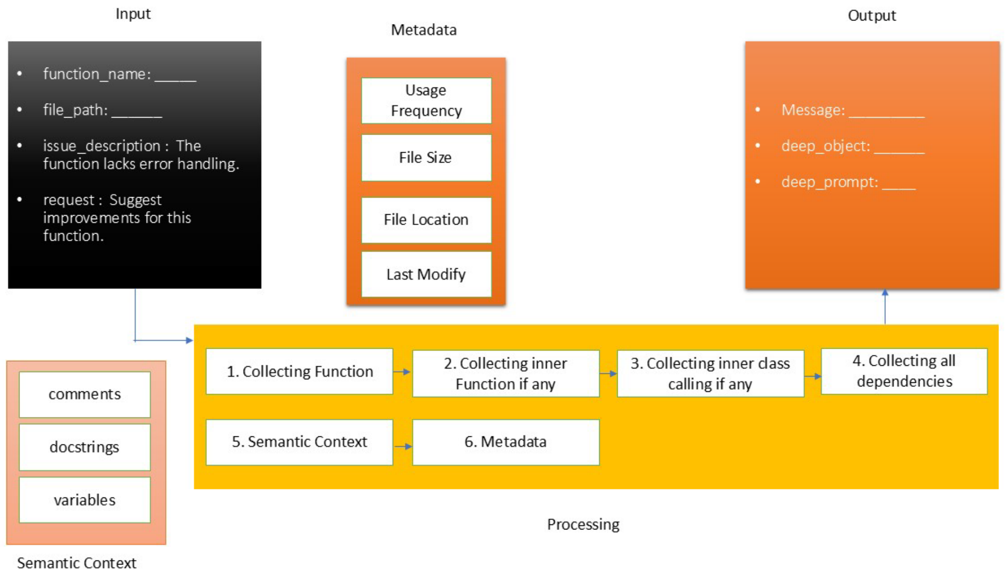

Figure 3. Cross-file code analysis system locates target functions, traces all dependencies (even across files), and delivers AI-enhanced solutions or structured JSON outputs with full metadata

V. Training & Results

Using a number of stringent criteria to ensure quality, relevance, and recent activity, we filtered 100 Python repositories from GitHub to create a diverse and robust dataset for model training. Specifically, projects were selected if they had more than 1000 stars and 500 forks, used Python as their primary language, and had been actively updated with commits pushed after January 1, 2023.

Additionally, repositories were grouped according to their size, as indicated by the quantity of files, in order to balance the dataset across various project scales:

- Large projects included repositories with more than 500 files.
- Projects with 100–500 files were considered medium.
- Repositories containing less than 100 files were considered small projects.

The GPT-4 model architecture, which was selected for training due to its sophisticated code interpretation and natural language generating capabilities, was used. Using the varied dataset gathered from GitHub, the training pipeline entailed optimizing the model especially for Python code comprehension tasks.

Three commonly used measures in text generation research—BERTScore, BLEU, and ROUGE-L—were used to assess the model's performance. Deep contextual embeddings were used to quantify the semantic similarity between the generated outputs and reference texts using BERTScore. An assessment based on n-gram overlaps was offered by BLEU; this was especially helpful for organized outputs such as documentation and code comments. In order to capture recall-focused overlaps between generated and target sequences, ROUGE-L evaluated the longest shared subsequence.

Throughout the test dataset, the GPT-4 based model continuously met or surpassed these thresholds Table 1, confirming the efficacy of both the model training approach and the repository selection procedure.

Table 1. GPT-4 trained dataset results across multiple size of repositories. The word Repo in the table heading represent the word repositories which is a source code of Github.

	File Size	**No. of Repo**	**BERTScore**	**BLEU**	**ROUGE-L**
Small	<100	37	0.81	0.25	0.35
Medium	<500	33	0.79	0.27	0.39
Large	>500	30	0.83	0.29	0.36

VI. Conclusion

Using file and function names, our experiment shows that enabling cross-file code search enhances conventional single-file search techniques. When an API key is supplied, the solution effectively pulls pertinent code segments and metadata, providing results in JSON format for manual processing or immediately via the model. This method improves context awareness and code navigation, which increases its adaptability to real-world development contexts.

Code Availability

The complete implementation code is available at Github https://github.com/NoumanAhmad448/multi_files_codebase

Acknowledgment

Utilizing Northeastern University's facilities and computer resources, this study was carried out with additional funding from the Chinese Government Scholarship (CSC) program, which made the experimental phase possible.

References

[1] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," *ACM Comput. Surv.*, vol. 57, no. 6, pp. 1–39, 2025.

[2] X. Hou et al., "Large language models for software engineering: A systematic literature review," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 8, pp. 1-79, 2024.

[3] Y. Ding et al., "CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion," Adv. Neural Inf. Process. Syst., vol. 36, pp. 46701-46723, 2023.

[4] B. Xiao, B. Kantarci, J. Kang, D. Niyato, and M. Guizani, "Efficient prompting for LLM-based generative internet of things," *IEEE Internet of Things Journal*, 2024.

[5] J. Wen et al., "From generative AI to generative Internet of Things: Fundamentals, framework, and outlooks," *IEEE Internet Things Mag.*, vol. 7, no. 3, pp. 30–37, May 2024, doi: 10.1109/IOTM.001.2300255.

[6] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[7] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," arXiv:2203.13474, 2022.

[8] R. Li et al., "StarCoder: May the source be with you!," arXiv:2305.06161, 2023.

[9] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," arXiv:2102.04664, 2021.

[10] B. Athiwaratkun et al., "Multi-lingual evaluation of code generation models," arXiv:2210.14868, 2022.

[11] J. Austin et al., "Program synthesis with large language models," arXiv:2108.07732, 2021

[12] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.

[13] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2020

[14] Y. Lappalainen and N. Narayanan, "Aisha: A custom AI library chatbot using the ChatGPT API," *Journal of Web Librarianship*, vol. 17, no. 3, pp. 37–58, 2023.

[15] P. Bansal and A. Ouda, "Study on integration of FastAPI and machine learning for continuous authentication of behavioral biometrics," in *Proc. 2022 Int. Symp. Networks, Computers and Communications (ISNCC)*, pp. 1–6, 2022.

.