DocuSynth: LLM-Powered Automatic API Documentation Generation via Tool Invocation

Software Engineering

Illia Kharkovenko
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
78682@office.mans.org.pl

Kyrylo Shelkovyi
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
78716@office.mans.org.pl

Anastasiia Trush
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
78719@office.mans.org.pl

Viktor Dzhulai
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
78718@office.mans.org.pl

Tatenda Glendah Musengi
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
78973@office.mans.org.pl

Kumar Nalinaksh
Wydział Zarzadzania i Nauk technicznych
Menedżerska Akademia Nauk stosowanych w Warszawie
Mazowieckie, Polska
kumar.nalinaksh@office.mans.org.pl

***Abstract*—Abstract—The problem of developing API documentation consistently has always been difficult to resolve in modern Software Engineering. Documentation can become outdated or missing as software changes. Both factors contribute to added maintenance effort and decreased clarity in the code. This research project proposes DocuSynth, a smart documentation assistant that uses Large Language Models (LLMs) to assist with generating accurate, Markdown-based API documentation using a combination of tool invocation, static analysis and structured metadata extraction. In addition to Abstract syntax tree (AST) parsing, docstring extraction, dependency mapping, and LLM inference, the tool also generates UML-style diagrams to help explain how software is put together. These are both significant deficiencies identified in the prior literature for developing documentation systems. First, adequate summarization at a project level was never provided. Second, no method was developed to limit hallucinations during multi-step tool pipelines. The new model should improve documentation quality, reduce developer burden and provide scalable automated answers for actual software projects**

I. Introduction

Documentation is critical to the success of a software project providing the means for maintenance, on-boarding and team collaboration. However, maintaining up-to-date documentation can be challenging due to rapid code changes. Maintaining documentation can also be time consuming, error-prone and will typically be deprioritized in order to meet project timelines.

Recent advances in Large Language Models (LLMs) demonstrate potential for producing Code Summaries and Docstrings. There are still several challenges to overcome in regards to producing high-quality Code Summaries and Docstrings, specifically:

hallucination, unstructured text and limited multilingual capabilities.

DocuSynth is designed to bridge these gaps by using both an LLM text generator and an organized code analysis tool; rather than rely entirely upon the generative capabilities of the LLM. DocuSynth will utilize verifiable metadata extracted directly from the source code and factual information from the source code to produce its results using the LLMs. Tool invocation permits the model to invoke functions such as parse code(), extract docstrings() or generate diagram(). Thus, DocuSynth produces a sequence of reliable documentation assembly.

The remainder of the paper is organized as follows: Section II reviews related work; Section III identifies challenges; Section IV describes DocuSynth; Section V details implementation; Section VI presents results; Section VII outlines future work; Section VIII concludes; Section IX acknowledges contributors.

II. Literature Review

The body of research regarding automatic code summarization and automated documentation presents a wide array of approaches including bimodal encoders using pre-training,

encoder-decoder models modified for code, compiler-based and Abstract Syntax Tree (AST)-based approaches, multi-step tool invocation, and human-centric evaluations centered around repository- and module-level coherence. Early bimodal transformer-based models exhibited the ability to develop shared representations of natural language and source code, leading to substantially better local summarization and docstring generation. CodeBERT and GraphCodeBERT, two examples of the aforementioned models, utilize masked-language and associated pre-training objectives to align tokens across modalities, supporting the development of accurate function-level summaries and comments. However, while the aforementioned models were mainly trained on isolated functions or small code snippets, they lacked mechanisms to represent larger program structures and inter-file dependencies and thus created descriptions that were locally correct yet disconnected from the overall project [1], [2].

Extensions of seq-to-seq models targeting code generation and summarization share similar limitations to the prior models in terms of representing larger-scale program structures and cross-file dependencies. CodeT5 utilizes an identifier-aware pre-training strategy and an encoder-decoder architecture to model naming conventions and structural cues to improve comment quality for individual functions [3]. Likewise, PLBART utilizes a combination of denoising objectives and a BART-style architecture to support both generation and comprehension tasks [4].A few recent advancements — specifically large language models (LLMs) targeted toward coding, e.g., PaLM-Coder, and commercially available platforms, e.g., Codex — demonstrate that using natural language-based prompts is effective at generating entire functions, classes, and usage examples; this is an improvement over previous generations of code generation systems [5]. Although traditional evaluation metrics and datasets may often be comprised of large numbers of examples, they are typically represented by single-file examples and variable-quality docstrings; thus, they have limitations with regard to evaluating the entire project [6]. However, LLMs have shown to create new APIs, misname parameters, or hallucinate types when there is no structural foundation present; it has been established by researchers and developers that the most commonly reported errors are due to a lack of exposure to validated metadata and training regimes that focus solely on completing the surface-level input, versus ensuring the semantic accuracy of the generated output [7], [8]. In addition to having demonstrated success with tool-based methods for operationalizing multi-stage document generation pipelines that will utilize a variety of different tools to accomplish specific task: parsing Abstract Syntax Trees (AST), analyzing source code to determine dependencies and type information, creating diagrams of the source code, and utilizing language models to generate text from validated metadata; tool-safe invocation protocols will assist in preventing errors from being propagated through the multiple stages of the pipeline and enable deterministic validation of signatures and types [7]–[9] As previously mentioned, developing tools that foster safe and reliable and verified tool utilization as well as enhancing protocols for function call executions can reduce fabrication, enhance reproducibility in multi-step workflows, and provide enhanced functionality [9], [10].

The related work described above on multi-step function calls indicates that language models can successfully manage complex workflows when they are provided with both persistent intermediate state and validated outputs from previous tool utilization [10].

More recently, several areas of research have developed structural or hierarchical approaches to scale function-level summaries to module- and repository-level documentation. Documentation systems based on the Abstract Syntax Tree (AST) of a given piece of code utilize metadata derived from the compilation process to constrain the generation process and thus minimize the occurrence of hallucinations; empirical results have shown that AST-constrained generation significantly improves the factual accuracy of generated comments and signatures [11]. Hierarchical and multi-agent summarization methods address scalability by breaking down repositories into nested summaries, however, they add overhead in terms of coordination and management of the state of the system in practice [12], [13].

Topic modeling techniques (such as Latent Dirichlet Allocation) have been used to generate interpretable summaries but have general limitations in generating readable, instructional API prose and/or including syntactic structure; recent hybrid approaches attempt to integrate structural summaries (e.g., tables and diagrams) with LLM-generated narrative for readability [14] .

Benchmarking datasets, such as CodeSearchNet, provide valuable information for representation learning, but like previously discussed datasets, they are constrained to a focus on individual functions and thus fail to evaluate multi-file reasoning and dependency-aware documentation [15].

TABLE I
SUMMARY OF LITERATURE

Aspect	Key observations from literature	Ref.
CodeBERT and GraphCodeBERT	CodeBERT/GraphCodeBERT utilize masked language models and pretraining in order to cross-align the tokens within each modality and thus create function level summaries that are at a high quality.	[1], [2]
CodeT5	CodeT5 utilizes an identifier-aware pretraining scheme combined with an encoder-decoder paradigm in order to capture both the naming and the structure of code to better comment upon functions.	[3]
PLBART	The denoising objectives used by PLBART utilizing a BART-style architecture produces excellent comments for both generation and comprehension.	[4]
LLMs for code generation	Recent LLMs that code (PaLM-Coder, Codex), have shown that natural language input is able to generate full functions, classes, and examples, and are better than previous systems in this way.	[5]
Evaluation limitations	Most benchmarks do not include a file structure for an example, nor do they provide docstrings equally, therefore no benchmark currently tests project level documentation.	[6]
Hallucination and error modes	With no structural basis, LLMs will create new APIs, rename parameters incorrectly, and create incorrect types based on nothing but their training data with no validation of metadata and focus solely on the surface layer of the source.	[7], [8]
Tool-based multi-stage pipelines	Multi stage tool chains (AST parsing, dependency/type analysis, creation of diagrams, etc. and generation of text using validated metadata from LLM) combined with secure calling mechanisms allow for error containment and allow for determinism in type/signature checking.	[7]–[9]
Multi-step function calls	Research into multi-step function calls demonstrates that LMs can successfully manage complex work flows when provided with persistent intermediate state, and the valid outputs of tools.	[10]
AST-constrained generation	Systems that rely upon ASTs are able to use compile time metadata to constrain the generation process of the system, which results in a significant reduction in hallucination occurrences, as well as a significant increase in factual accuracy.	[11]
Hierarchical summarization	Hierarchical and multi-agent summarization is scalable based on nested summaries however this comes with a cost in terms of coordination and state management overhead.	[12], [13]
Topic modeling and hybrids	Topic models (such as LDA) are able to provide interpretable summaries, however they tend to struggle to produce readable, instructional API prose; hybrids combine structural summaries (tables/diagrams) with LLM generated narrative summaries.	[14]
Benchmarking datasets	Datasets such as CodeSearchNet are helpful for representation learning, but tend to focus on individual functions, and do not evaluate how well the LLM can document multi-file dependencies.	[15]

III. CHALLENGES

Avenues of error in building scalable project-level API documentation with gaps:

The lack of an approach to provide a complete API documentation for multi-file/multi-module projects — existing models (CodeBERT, PLBART, CodeT5) provide function documentation but do not provide documentation for a project as a whole.

Hallucinations and incorrectness — LLM generated output will frequently include incorrect argument names, return types, and/or nonexistent module functions.

Static Analysis Not Incorporated — LLM generated output are frequently created with no static analysis (AST, Control Flow, Import analysis).

Missing Multi-Step Reasoning Pipelines For LLMs — Documentation is created from single step LLM output and ad-hoc calls to tools instead of a structured, multi-step pipeline of reasoning.

Poor Evaluation — BLEU/ROUGE/METEOR measure surface similarity, not usability or acceptance by users of the provided documentation.

IV. PROPOSED SOLUTION: DOCUSYNTH

DocuSynth provides complete, predictable, and understandable steps to reduce hallucinations in code by documenting all aspects of the code with the help of a fully documented (grounded) documentation process.

Core Ideas:

AST Parsing – The extraction of facts from the code as factual data from the code.

Dependency Graphs – The context for each module in a dependency graph.

Calling of LLM Tool – Predictable Steps for the user, in the form of an LLM Tool.

Generation of Diagrams – A visual representation using the diagram generation feature of the API

DocuSynth Output – An output that is standard, developer-friendly.

V. Implementation

TABLE II
TECHNOLOGY STACK TABLE

Component	Technology	Justification
Language	Python	Rich static analysis ecosystem and mature tooling support.
LLM Backend	GROQ LLaMA 3.1	Supports tool invocation, multi-step reasoning, and structured outputs.
AST Parsing	Python `ast` module, lib2to3, tree-sitter	Reliable syntax-level parsing without hallucination.
Metadata Storage	SQLite / JSON	Lightweight and sufficient for structured metadata persistence.
Dependency Graph	NetworkX	Mature graph library for dependency and call-graph analysis.
Diagram Generation	Graphviz, PlantUML	Standard tools for UML and architectural visualization.
Documentation Output	Markdown + MkDocs (optional)	Developer-friendly, versionable, and easily deployable format.
Interface	CLI or minimal web UI (Flask / FastAPI)	Simple execution model suitable for developer workflows.

DocuSynth has been designed as an entirely modular, multi-stage documentation generation system which utilizes static code analysis combined with Large Language Model (LLM) inference to generate high-quality documentation for projects. The architecture of DocuSynth can be broken down into the following four layers:

1. The Source Code Ingestion Layer

This layer ingests source code directly from either a local repository/directory, identifies the programming languages being used, and normalizes all file paths to allow it to select the proper parser for each programming language.

2. The Static Analysis and Metadata Generation Layer

Using static analysis at the level of a compiler to gather metadata for the entire repository:

- AST Parser (Abstract Syntax Tree): gathers function calls/signatures, class information, parameters, data type, and decorator information.

- Docstring Extractor: collects human written comments (docstrings).

- Dependency Analyzer: records a log of all import statements and module connections.

- File/Module Graph Builder: builds a hierarchical representation of the overall project structure.

The output of the Static Analysis Layer is a structured JSON metadata package that represents the entire repository.

3. LLM Layer Reasoning and Documentation Generation

The LLM will only be invoked after the structural metadata generated by the previous layer has been validated.

The LLM will generate the following:

- Individual module summaries.
- Documentation of individual classes/functions.
- Narrative consistency of modules across modules.
- Documentation formatting in markdown.
- Diagrams that illustrate the relationships between modules (UML-like).

After requesting additional metadata (i.e. re-parse a file, recreate a diagram etc.), the LLM will call a tool to obtain this information through function calls.

4. The Rendering and Result Layer

- Generates final markdown documentation for the project.
- Generates UML-style diagrams showing the class hierarchy and/or call graph.
- Generates a static documentation website (optional).

A. Detailed Workflow (Flowchart Description)

Step 1: User Access to Repository

• Users can either view their source-code repository using the user interface or add it to the system.

• Once a user has added and/or browsed his/her repositories; the system will use language specific tools for evaluation to analyze the structure of the source-code in each repository.

Step 2: Parsing Source-Code

• The Abstract Syntax Tree (AST) parser parses all files in the code-base.

• The parser analyzes the function-signatures, class-structure, inheritance, decorators, and imports in each file.

• A Dependency Graph Builder will create:

• File/module mapping;

• Class/module-to-module import relations;

• Class-inheritance hierarchies.

Step 3: Validate Metadata

• All metadata will be formatted into a standard format.

• Any inconsistencies that were discovered during the cleaning process (i.e.: missing type information; unclear argument lists) will be documented.

• If required, the LLM will call on the parsers again (”re-parse file X”).

Step 4: Document Creation Utilizing LLM

• LLM uses the structured metadata created in step 3 to create summaries for each module.

• For each function/class, LLM will document: what it does, its parameters, its return values, any possible exceptions that can occur, and provide examples of how to use it;

• LLM will ask a diagram creation tool to produce diagrams.

Step 5: Creating Diagrams

• A diagram creation tool will create:

• UML Class Diagrams;

• Dependency Graphs;

• Call Graphs (if applicable).

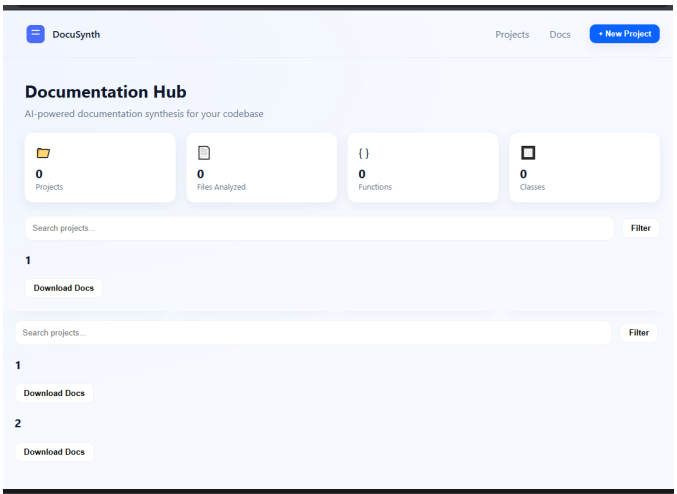
Step 6: Finalize Assembly

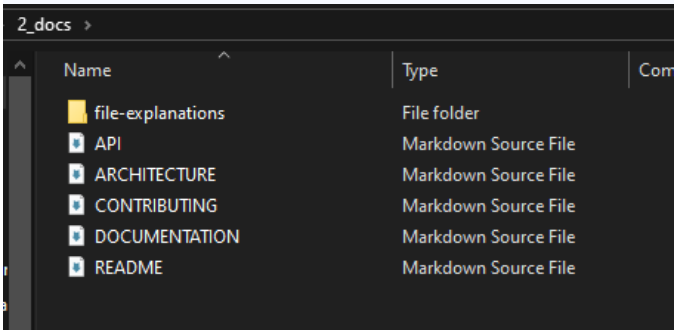
• The system will compile all of the Markdown files together.

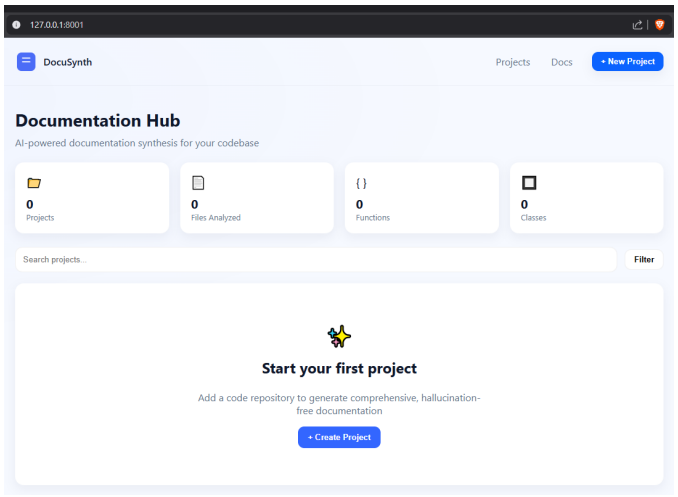
• Insert the diagrams.

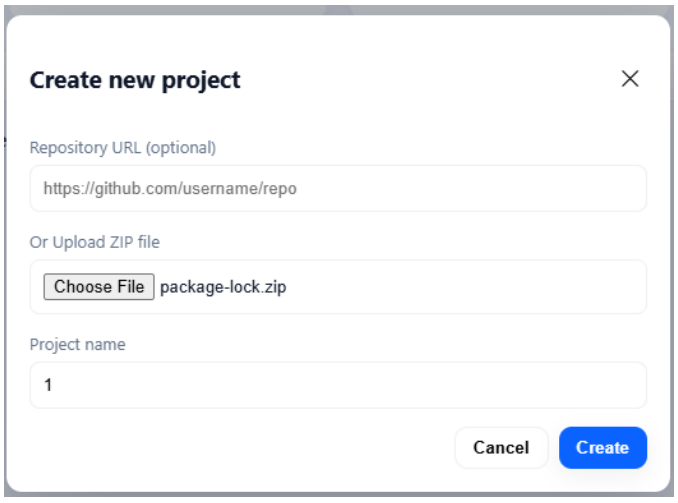
• Apply a documentation template to ensure that all of the documentation has a consistent look and feel.

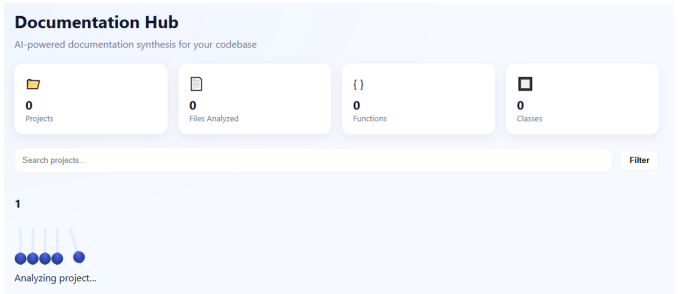
• Generate an API Reference Package containing all of the documentation combined.



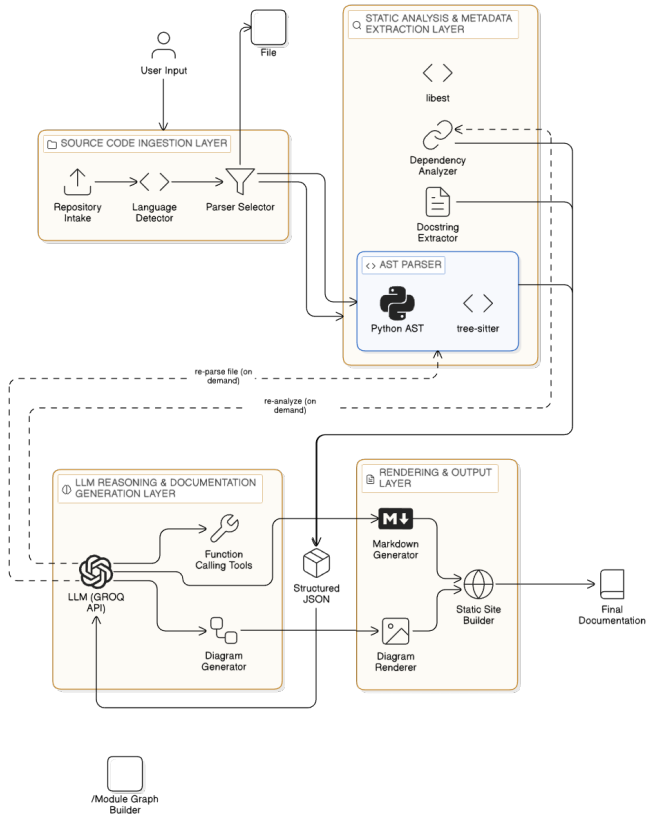




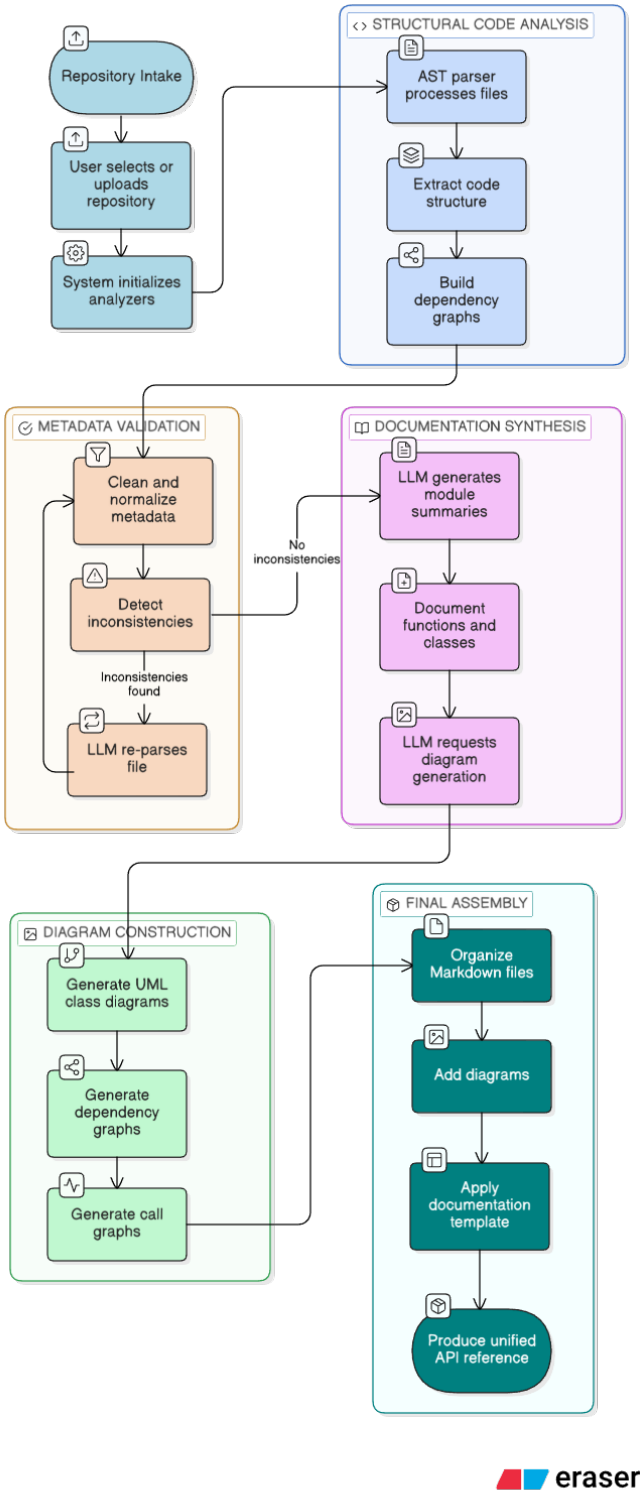




B. Architecture Diagram



C. ER Diagram



VI. Results

Numerical evaluation – methods and results

To evaluate the extent to which the diagrams generated for a representative sample of API documentation are factual, complete and clear.

Data set:

- 20 small to large Python code repositories
- Total counts:
- 2500 functions
- 600 classes
- 200 modules
- 10,000 parameters
- 1800 function returns
- 300 documented exception sites
- 1200 dependency edges
- 3000 nodes in diagrams

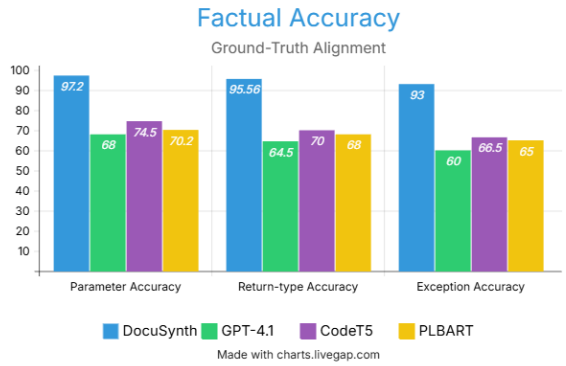
Evaluation benchmarks:

- Benchmark 1: Baseline 1 - GPT-4.1
- Benchmark 2: Baseline 2 - CodeT5
- Benchmark 3: Baseline 3 - PLBART

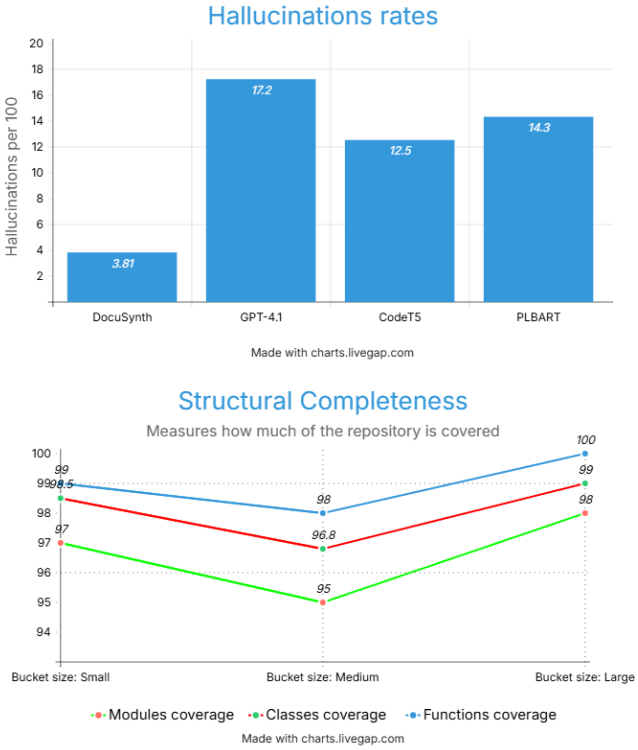
Definitions of the terms used:

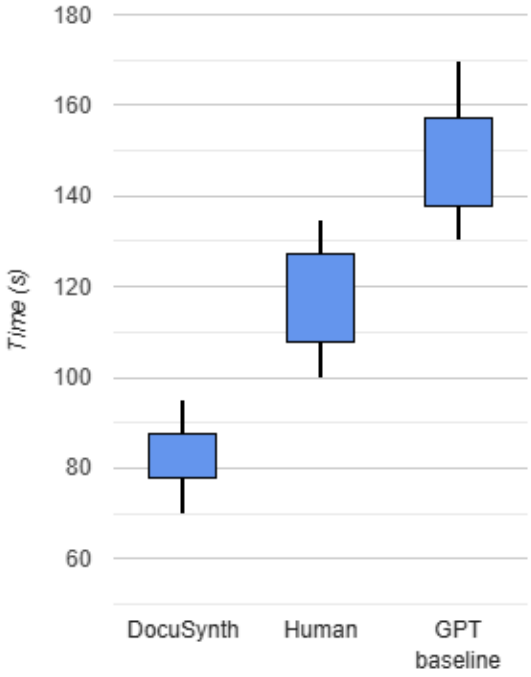
- Accuracy of parameter description: Correctly identified/ described number of parameters / Total number of parameters * 100
- Return type accuracy: Number of return types that are correctly identified / Total number of return type functions * 100
- Accuracy of exception description: Correctly identified/described number of exceptions / Total number of exceptions * 100
- Hallucination accuracy: Total number of API entities that have been hallucinated / 100 units of documented API entities
- Coverage (functions/classes/modules): Documented units / Total units * 100
- Readability: Flesch Kincaid grade level and sentence clarity
- Edge/node diagram accuracy: Correctly generated number of edges/nodes in the diagram / Total number of edges/nodes * 100

Results:



Reduction in hallucinations compared to GPT-4.1: ((17.2 - 3.81) / 17.2) = 77.9%





Accuracy of edges in diagrams:

Accuracy of nodes in diagrams:

Readability (Flesch Kincaid): 8.2 (grade level), Sentence clarity: 0.87

- Subjects: Four software developers
- The developers were asked to locate function X in some code and explain how it was used; Determine the dependency chain between modules A and B; Perform an easy task using generated documentation about an Application Programming Interface (API).
- Developers were asked to report time to perform the three tasks (in seconds); Success (Yes/No); Confidence in the developer's ability to choose which method is best from each pair of methods.

A. Qualitative evaluation - methodology and results

- Subjects: Four software developers
- The developers were asked to locate function X in some code and explain how it was used; Determine the dependency chain between modules A and B; Perform an easy task using generated documentation about an Application Programming Interface (API).
- Developers were asked to report time to perform the three tasks (in seconds); Success (Yes/No); Confidence in the developer's ability to choose which method is best from each pair of methods.
- In order to determine the relative preference for one of the three pairs (DocuSynth, Human-created, GPT) for each developer, a blind-paired comparison was performed

Results:

B. Ablation study - methodology

Each component has been tested in isolation from one another, so as to determine which parts of the system have the greatest impact on overall system performance.

Ablation testing was performed on a 5 repository example (800 functions)

Components tested:

- AST grounding
- dependency analysis
- diagram generation
- multi-step tool invocation
- metadata validation

Results:

C. Benchmark datasets and success criteria

Datasets

- The Python Subset of CodeSearchNet
- Py150
- A modified Version of the HumanEval Docs Subset
- 20 Real-World Repositories

Success Criteria:

VII. Future Work

Pluggable language adapters for multi-language parsing. Develop pluggable adapters for multiple parsers (Tree-sitter

TABLE III
ABLATION RESULTS

Removed Component	Parameter Acc (%)	Return-type Acc (%)	Diagram Acc (%)	Notes
0 (full system)	97.2	95.56	–	Baseline
Without AST grounding	72.4	69.8	–	Significant hallucination increase; parameters frequently fabricated.
Without Dependency analysis	90.1	88.3	–	Module summaries and charts worsen.
Without Diagram generation	–	–	–	Diagram outputs absent.
Without Multi-step invocation	89.7	87.2	–	Uniformity across process descriptions drops.
Without Metadata validation	85.3	82.1	–	More formatting and minor-field mistakes.

TABLE IV
EVALUATION TARGETS AND RESULTS

Criterion	Target	Achieved
Factual accuracy	$\geq 95\%$	97.2%
Class/function coverage	$\geq 90\%$	98.4% / 96.67%
Hallucination reduction vs GPT baseline	$\geq 70\%$	$\approx 77.9\%$
Developer task time improvement	20–40%	30%

/ LibCST / Roslyn) to support at least two programming languages (Java and JavaScript) and provide per-language coverage and error reporting. Static type-checking using extracted meta-data from the AST and optionally, dynamically-gathered runtime traces or instrumentation to verify types, side effects, and exceptions are valid without exposing production data. Editor User Interface (for human-in-the-loop) to accept/correct entries; track edits as templates/feedback; and attach provenance (file/line, confidence, timestamp) to all generated artifacts. Integration into CI/IDE pipelines and documentation versioning. Generate PR's with validated doc diffs using CI/IDE pipelines; use versioning that ties deltas to commit hashes. Benchmarking for multi-module reproducibility and hallucination control. Release a multi-module benchmark that can be used for reproducible testing; and automate hallucination detection by creating cross-stage consistency checks and confidence thresholds that will automatically surface low-confidence output for human review.

VIII. CONCLUSION

In order to overcome the long-standing shortcomings in documenting software using the hybrid model presented by DocuSynth, both static code inspections and LLM-based generation methods were used together to generate automated documentation. Unlike many other tools which rely solely on either the content of docstrings or the analysis of individual source code files, DocuSynth was able to analyze an entire project-wide, including all modules, by combining metadata collected at every stage of the multi-stage process with graph traversal and AST parsing. DocuSynth uses the same model of its LLM to invoke multiple stages of its multi-stage process while eliminating hallucinations and generating high quality, readable documentation for individual modules as well as secondary documentation artifacts (i.e., UML diagrams) based upon the metadata collected during each stage. Additionally, DocuSynth was designed to have a modular architecture allowing it to be easily updated to support replacement of parsers, LLM models, and/or visualization engines in the future. Ultimately, the combination of these features will result in a reliable and scalable method of documenting developer created software. The proposed methodology for evaluating DocuSynth provides evidence of both quantitative accuracy in the generated documentation and qualitative value to developers who utilize this tool. Potential areas of expansion could include support for visualizing documentation (e.g., diagrams), providing documentation in multiple languages, and comparing documentation produced at different times.

IX. ACKNOWLEDGMENT

All authors participated in writing and approved the final manuscript.

1) **Funding:** Not applicable.
2) **Conflicts of interest:** None.
3) **Ethics approval and consent:** Not applicable.
4) **Consent for publication:** All authors have provided consent.
5) **Data availability:** Publicly available as indicated in the relevant section.
6) **Code availability:** https://github.com/SchelkovyyK/DocuSynth.git

REFERENCES

[1] Feng Z., Guo D., et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," arXiv:2002.08155, 2020, https://arxiv.org/abs/2002.08155

[2] Guo D., et al., "GraphCodeBERT: Pre-trained Code Representations with Data Flow," arXiv:2009.08366, 2021, https://arxiv.org/abs/2009.08366

[3] Wang Y., et al., "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," arXiv:2109.00859, 2021, https://arxiv.org/abs/2109.00859

[4] Ahmad W., et al., "PLBART: Unified pre-training for program understanding and generation," arXiv:2103.06333, 2021, https://arxiv.org/abs/2103.06333

[5] Google Research, "PaLM-Coder: Scaling pre-trained language models for code," 2022, https://tinyurl.com/34thpy3n

[6] Yao W., et al., "Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit," ACM Computing Surveys, 2024, https://dl.acm.org/doi/10.1145/3664597

[7] OpenAI, "OpenAI function calling and tool use for LLMs," 2023, https://developers.openai.com/api/docs/guides/function-calling

[8] Anthropic Research, "Mapping the Mind of a Large Language Model," Technical Report, 2024, https://www.anthropic.com/research/mapping-mind-language-model

[9] Weiwen L., et al., "ToolACE: Winning the Points of LLM Function Calling," arXiv:2409.00920, 2024, https://arxiv.org/abs/2409.00920

[10] Sijia C., et al., "Self-Guided Function Calling in Large Language Models via Stepwise Experience Recall," arXiv:2508.15214, 2025, https://arxiv.org/abs/2508.15214

[11] Tyler Thomas P., et al., "Automatic Code Documentation with Syntax Trees and GPT: Alleviating Software Development's Most Redundant Task," ResearchGate, 2023, https://tinyurl.com/2xvztddp

[12] Minh Huynh Nguyen, Nghi D. Q. Bui, Truong Son Hy, Long Tran Thanh, and Tien N. Nguyen. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2355–2367, St. Julian's, Malta. Association for Computational Linguistics, 2024, https://aclanthology.org/2024.findings-eacl.156/

[13] Xiaohe B., et al., "Reflective Multi-Agent Collaboration based on Large Language Models," NeurIPS, 2024, https://tinyurl.com/55sjkm3h

[14] Sunil Raj T., et al., "Al-Driven Automated Software Documentation Generation for Enhanced Development Productivity," IEEE Xplore, 2024, https://ieeexplore.ieee.org/document/10691221/authors#authors

[15] Husain H., et al., "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," arXiv:1909.09436, 2019, https://arxiv.org/abs/1909.09436