Automated README Generation from a Single Script

Cole Lavigne
Department of Computer Science & Engineering
University of Connecticut
Storrs, CT, USA
cole.lavigne@uconn.edu

Hiral Ankur Choksi
Department of Computer Science & Engineering
University of Connecticut
Storrs, CT, USA
hiral_ankur.choksi@uconn.edu

Varun Raghunath Reddy
Department of Computer Science & Engineering
University of Connecticut
Storrs, CT, USA
sur24005@uconn.edu

Abstract

Modern software repositories frequently lack high-quality documentation, especially small single-file utilities where developers provide code but omit usage instructions, dependency information, or CLI examples. This project presents `readmegen`, a modular system that automatically generates fact-grounded README files directly from a Python script. The system combines static analysis (extracting imports, CLI flags, inferred descriptions, and metadata) with Large Language Models (LLMs) to produce structured documentation that is both fluent and empirically verifiable.

We evaluate two variants in this project, which include an AST-guided pipeline and a no-AST LLM-only pipeline using 21 real Python scripts and multiple model combinations (OpenAI GPT-4o-mini, Gemini-2.0-flash). A unified LLM-as-a-Judge framework scores dependencies, CLI completeness, and description quality. Results show that static analysis dramatically improves dependency precision and reduces hallucinations, while the LLM-only pipeline maintains strong natural-language clarity but performs poorly on factual sections. These findings demonstrate that hybrid static-analysis combined with LLM systems can meaningfully narrow the gap between code and the documentation required to support it.

1 Introduction

Software documentation is a persistent but well-known bottleneck in real-world development workflows. Developers consistently write and maintain code but rarely take the time to produce and update comprehensive README files. This problem is even more pronounced in small or single-script repositories, where more than 70% of lightweight projects contain incomplete or entirely missing documentation. The resulting lack of guidance introduces significant obstacles to reproducibility, collaboration, onboarding, and long-term maintainability.

Although this problem is longstanding, recent advances in Large Language Models (LLMs) offer a promising path forward. LLMs excel at natural-language generation and can quickly summarize code. However, one of our central findings during early experimentation was that LLMs prompted to generate README files independently often hallucinate critical details. These hallucinations may include fabricated dependencies, incorrect CLI arguments, or imagined features, undermining the utility of the documentation. This raises the central question of our project:

Can LLMs be guided to generate accurate, complete documentation from a single Python script? If so, what model and architectural choices minimize hallucinations while maintaining high-quality natural language output?

To investigate these questions, we developed `readmegen`, a hybrid README generator that integrates static code analysis with LLM-based natural-language generation. Our system extracts structured metadata directly from the script, which includes imports, CLI flags, docstrings, comments, and inferred descriptions. Our model uses these extracted facts to construct a grounded prompt supplied to LLMs. By constraining generation to verifiable extracted facts, our approach reduces hallucinations while still benefiting from the summarization fluency of LLMs.

Based on our project goals and early prototyping challenges, we investigated the following research questions throughout the semester:

- Can LLMs summarize entire Python scripts into coherent, high-quality README files?
- Does integrating static code analysis reduce hallucinations and improve factual accuracy in generated documentation?
- What architecture best balances LLM fluency with factual grounding from static analysis?
- How consistent are different LLMs when given the same extracted metadata?

To answer these questions, we built an end-to-end system that is capable of automatically generating README files and a fully automated evaluation pipeline employing an LLM judge design. This evaluation framework measures clarity, completeness, helpfulness, dependency precision/recall, and CLI flag extraction accuracy across multiple models (OpenAI and Gemini). The final architecture represents a significant progress push from our midterm presentation, incorporating improvements to prompt engineering, metadata extraction, and evaluation reproducibility.

This paper presents the design, implementation, and empirical study of `readmegen`. We show that hybrid prompting grounded in static analysis significantly reduces hallucinations and improves factual accuracy. Our results also reveal meaningful limitations in current LLM-based documentation systems, identifying opportunities for future work in scaling, evaluation, and applied deployment.

2 Related Work

2.1 Neural Code Summarization

Early approaches to automated code documentation relied heavily on sequence-based neural models. Hu et al. introduced DeepCom, a pioneering framework that leveraged recurrent neural networks and structural information extracted from Abstract Syntax Trees (ASTs) to generate code comments for Java methods [3]. While effective at method-level summarization, such approaches were limited in scope and typically required language-specific training data.

The introduction of Transformer-based pre-trained models significantly advanced the field. CodeBERT [2] demonstrated that pre-training on paired natural language and programming language corpora enables stronger semantic alignment between code and documentation. Building on this idea, CodeT5 [5] proposed an identifier-aware encoder–decoder architecture capable of supporting a wide range of code understanding and generation tasks. These models achieve strong performance on benchmarks such as code summarization and documentation generation, but they typically operate on snippets or functions rather than entire scripts, and they do not explicitly enforce factual grounding during generation.

2.2 Large Language Models in Software Engineering

Recent work has explored the use of general-purpose Large Language Models (LLMs) for software engineering tasks. Sun et al. conduct a systematic study of LLM-based source code summarization, demonstrating that modern LLMs can produce fluent and coherent summaries with minimal prompting, often outperforming task-specific models trained on smaller datasets [4]. However, their study also highlights persistent challenges related to hallucinations, evaluation reliability, and factual correctness.

More broadly, Allamanis et al. survey the application of machine learning techniques to “big code” analysis, emphasizing the importance of probabilistic models for capturing coding conventions and improving developer productivity [1]. This body of work underscores the potential of machine learning to assist developers, while also highlighting the risks of unconstrained generation in safety- or correctness-critical settings.

2.3 Differentiation from Existing Tools

Industry tools such as GitHub Copilot and Codeium leverage similar architectures, but they primarily focus on inline code completion or function-level assistance during development. These systems are designed as interactive coding aids rather than documentation generators.

In contrast, *readmegen* targets project-level documentation generation, with a specific focus on producing complete `README.md` files for standalone Python scripts. Unlike prior work that relies solely on neural generation, our approach explicitly integrates static code analysis to extract verifiable facts, which are used to ground the facts for the model. This design choice directly addresses hallucination and factual inconsistency, positioning *readmegen* as a hybrid system that combines the strengths of static analysis with the robustness of modern LLMs.

3 Methodology

Our goal was to design a system capable of producing high-quality, fact-grounded README files directly from a single Python script. To accomplish this, we developed two parallel pipelines: (1) an *AST-guided pipeline* that incorporates structured static analysis, and (2) a *no-AST LLM-only pipeline* used as a baseline. Both pipelines share the same generation and evaluation infrastructure to ensure comparability.

3.1 System Overview

The final architecture consists of four major components:

1. **Static Analyzer (AST-guided pipeline)**
   Parses Python code via the `ast` module to extract structured metadata, including: imports, CLI flags, docstrings, comments, inferred script purpose, and potential configuration variables. This metadata is the grounding facts for the LLM to be given to be used to reduce hallucinations.
2. **Fact Grounding Module**
   Converts extracted metadata into a normalized `facts.json` representation containing key fields such as:
   - third-party dependencies,
   - command-line flags and options,
   - function/class-level descriptions,
   - inferred script behavior or purpose.

   These facts are injected into model prompts, ensuring the LLM generates documentation consistent with the script’s actual behavior.
3. **LLM Generator**
   An LLM interface supporting multiple model providers (OpenAI, Gemini), enabling controlled comparisons across the same extracted facts. Prompt templates emphasize factual grounding:

   > “You must not invent behavior beyond the provided facts. If information is missing, state assumptions explicitly or omit those sections.”

   The generator outputs a structured README containing sections such as Overview, Features, Installation, Dependencies, Usage, Examples, and Limitations.
4. **Unified Evaluation Framework**
   To measure documentation quality, we developed an automated evaluation pipeline using an *LLM-as-a-Judge* design. Given the reference "ground truth" README and the generated README, the judge model scores:
   - dependency precision and recall,
   - CLI completeness,
   - description clarity,
   - description completeness,
   - helpfulness of the explanation.

   This approach allows repeatable quantitative evaluation across different model configurations.

3.2 No-AST Model - LLM-Only Baseline

To isolate the effect of static analysis, we constructed a pipeline that removes all AST-based metadata. Instead, the model receives only the raw script text and a simplified prompt:

> “Generate a README based solely on the code below.”

This pipeline enables us to measure how strongly static analysis contributes to factual quality, hallucination reduction, and dependency accuracy.

3.3 Evaluation Dataset

We evaluated both pipelines on a curated dataset of 21 real-world Python scripts covering a range of tasks such as image processing, encryption utilities, PDF merging, web scraping, CLI tools, and network scanning. Each script includes a hand-written README used as ground truth during evaluation.

3.4 Experiment Matrix

To fully explore model interactions, we tested both pipelines under four model configurations:

- OpenAI generator → OpenAI judge
- OpenAI generator → Gemini judge
- Gemini generator → OpenAI judge
- Gemini generator → Gemini judge

This matrix allows us to measure:

(1) sensitivity to the generation model,
(2) sensitivity to the judge model,
(3) cross-model stability,
(4) consistency of scoring across backends.

3.5 Summary

Together, these components form a framework for studying documentation generation from single Python scripts. The AST-guided pipeline provides grounded, structured context, while the LLM-only pipeline serves as a baseline to see what LLMs are capable of by themselves in this research space. The evaluation system ensures objective, repeatable scoring across multiple model configurations, enabling a systematic assessment of factors that influence README quality.

4 Evaluation

We evaluate our README generation pipeline under two conditions: (1) **No-AST (LLM-only)** where the generator receives only raw script text and must infer dependencies and CLI usage without static facts, and (2) **AST-grounded** where extracted static signals (imports/requirements, CLI structure, docstrings) are provided to ground dependency and usage sections. Across all runs we report **factual metrics** (dependency precision/recall and CLI completeness) and **subjective ratings for an LLM judge** (clarity, completeness, helpfulness; 1–5 Likert scale). We also evaluate cross-model pairings by varying the *generator* and *judge* models.

4.1 Aggregate Results

Tables 1–4 summarize aggregate performance across generator/judge pairings. We separate **factual correctness** (Dep P/Dep R/CLI) from **narrative quality** (Clarity/Completeness/Helpfulness) to show how static grounding affects factual sections.

Table 1: No-AST (LLM-only) factual metrics.

Generator	Judge	Dep P	Dep R	CLI
OpenAI	OpenAI	0.033	0.012	0.579
OpenAI	Google	0.038	0.014	0.565
Google	OpenAI	0.033	0.012	0.579
Google	Google	0.033	0.012	0.579

Table 2: No-AST (LLM-only) subjective ratings (1–5).

Generator	Judge	Clarity	Completeness	Helpfulness
OpenAI	OpenAI	3.619	2.714	3.143
OpenAI	Google	3.611	2.944	3.500
Google	OpenAI	3.762	2.905	3.333
Google	Google	4.381	2.905	3.238

Table 3: AST-grounded factual metrics.

Generator	Judge	Dep P	Dep R	CLI
OpenAI	OpenAI	1.000	0.875	0.900
OpenAI	Google	1.000	0.875	0.900
Google	OpenAI	1.000	0.875	0.900
Google	Google	1.000	0.875	0.900

Table 4: AST-grounded subjective ratings (1–5).

Generator	Judge	Clarity	Completeness	Helpfulness
OpenAI	OpenAI	3.850	3.200	3.250
OpenAI	Google	3.950	3.300	3.250
Google	OpenAI	3.650	3.050	3.200
Google	Google	2.700	2.250	2.300

4.2 Key Insights

From our project, we can clearly see that **AST grounding is essential for dependency accuracy.** Without AST facts, dependency recovery is effectively unusable (Dep P ≈ 0.033-0.038; Dep R ≈ 0.012-0.014), showing the LLM-only baseline cannot reliably infer install requirements. With AST grounding, dependency precision is 1.000 and recall reaches 0.875 across pairings, indicating static extraction supplies the missing installation facts and prevents dependency hallucination.

CLI completeness improves with AST grounding, but remains imperfect. No-AST CLI completeness is ≈0.565-0.579, while AST-grounded runs reach 0.900. The remaining gap is consistent with scripts whose CLI is partially implicit (dynamic parsing, non-argparse patterns) or where the reference README includes usage details that are not fully recoverable from code structure alone.

Factual scores are stable across judges while narrative scores given from LLM judges are not. Under AST grounding, factual metrics are identical across all generator/judge pairings, suggesting the extracted facts dominate those sections. In contrast, subjective ratings vary sharply by judge (e.g., Gemini-as-judge is substantially harsher in the AST condition), implying rubric interpretation differences even when factual grounding is strong.

More will be discussed in the discussion section of this paper, but our results clearly show the idea that static analysis and using AST parsing along with an LLM generating a valid README is the optimal way to go about automated README generation based off of just one Python script.

5 Discussion

The results presented in Sections 1–4 demonstrate that *fact grounding via static analysis is the dominant factor in producing accurate README files*. While Large Language Models (LLMs) are capable of generating fluent and well-structured documentation, their ability to infer factual details such as dependencies and CLI usage from raw code alone is extremely limited.

5.1 Effectiveness of Static Analysis for Factual Accuracy

Across all No-AST configurations, dependency precision and recall are close to zero, regardless of generator or judge model. This confirms that LLMs, when prompted only with source code, frequently omit required dependencies or hallucinate installation steps that are not present in the script. On the other hand, the AST-grounded pipeline achieves perfect dependency precision and high recall across all model pairings. This gap highlights that dependency information is not reliably inferable from natural language patterns alone and instead requires explicit extraction from code structure.

A similar trend appears in CLI completeness. Without static grounding, CLI coverage averages approximately 0.57, reflecting an inference from usage strings or comments. Once AST-derived CLI metadata is provided, CLI completeness increases to 0.90. The remaining errors are largely attributable to scripts that construct arguments dynamically (e.g., via raw `sys.argv` access) or rely on runtime behavior that cannot be statically resolved. These findings align with examples observed during evaluation, such as Local_File_Organizer.py, where flags were embedded in conditional logic rather than declared through a formal argument parser.

5.2 Narrative Quality and Judge Sensitivity

Unlike our factual metrics, narrative quality scores (clarity, completeness, and helpfulness) show a decent variation across judges. Even under AST grounding, Gemini-judged runs consistently produce lower subjective scores than OpenAI-judged runs. This difference suggests that while factual sections become nearly deterministic when grounded, qualitative assessments remain sensitive to the LLM's perspective and expectations of what constitutes a "good" README.

This judge sensitivity is an important finding in itself. It demonstrates that LLM-as-a-Judge evaluation frameworks are reliable for factual verification but less similar for subjective narrative scoring between various different models, more than likely due to how the model was trained or parameter size differences. As a result, narrative metrics should be interpreted comparatively rather than absolutely, and future work should consider incorporating human evaluation or rubric calibration to improve consistency.

5.3 Script Quality as a Limiting Factor

A recurring theme across both pipelines is that documentation quality is bounded by script quality. Scripts with clear docstrings, structured CLI definitions, and explicit imports (e.g., `seam_carver.py`) consistently yield high scores across all dimensions. On the other hand, scripts that rely on implicit behavior, minimal comments, or dynamic imports limit what any automated system can reliably infer. This has significant meaning in this project because a well constructed and documented code is going to have a better corresponding README than one that doesn't.

Importantly, several low-performing cases were driven not by model failure, but also by incomplete or ambiguous ground-truth READMEs. This introduces unavoidable noise into automated evaluation and highlights the challenge of relying on human-written documentation as an oracle for correctness. Not every human README out there is "ground-truth" worthy, which is also shown in our results as some scripts always did bad compared to others.

5.4 Limitations

This study has several limitations. First, the dataset consists of only 21 single-file Python scripts, which constrains the statistical power and overall meaning of the results. Second, the system currently supports only single-file projects and not yet with multi-file repositories with requirements.txt files, shared state, or external documentation remain out of scope. Third, while static analysis significantly improves factual accuracy, it cannot resolve dynamic behaviors such as runtime imports, reflection, or environment-dependent configuration. Finally, the LLM-as-a-Judge framework introduces its own biases and occasional parsing failures, which required specific handling techniques.

5.5 Future Experiments

If additional time were available, several follow-up experiments would further clarify the strengths and limitations of the *readmegen* framework.

First, we would expand the evaluation dataset to include a larger and more diverse collection of Python scripts. A broader dataset would allow for better analysis of failure modes, particularly for scripts that rely on dynamic imports, unconventional CLI construction, or sparse documentation.

Second, we would evaluate the system on small multi-file repositories rather than strictly single-script inputs. While the current design intentionally focuses on minimal repositories, extending static analysis across multiple files would not only improve our results, but also be more tailored to more real world repositories.

Third, we would experiment with enhanced static signals beyond AST extraction. Potential additions include parsing `requirements.txt` or `pyproject.toml` files when present, identifying environment variables accessed by the script.

Finally, we could also conduct human evaluation to complement the LLM-as-a-Judge framework. Human reviewers could assess whether generated READMEs are practically usable for onboarding and deployment, helping validate whether improvements in factual metrics and narrative scores translate to real developer benefit.

These experiments would deepen our understanding of how static analysis and LLMs can be combined to produce reliable yet usable documentation.

6 Conclusion and Future Work

6.1 Conclusion

In this paper, we presented *readmegen*, a hybrid system for automatically generating README files directly from a single Python script. By combining static code analysis with LLMs, our approach showed a common failure mode of purely LLM-driven documentation systems: factual hallucination.

Through evaluation on 21 real-world Python scripts and multiple generators and judges, we showed that AST-grounded prompting dramatically improves dependency precision and recall, while also increasing CLI completeness. In contrast, LLM-only generation produces fluent and readable documentation but consistently fails to state correct installation requirements. These results demonstrate that static analysis is a critical component for producing good documentation that is meaningful.

At the same time, our findings reveal that narrative quality metrics remains sensitive to the choice of evaluation model. Even when factual grounding is strong, clarity, completeness, and helpfulness scores vary notably across judges, highlighting more challenges.

Overall, this work shows that hybrid static-analysis and LLM systems can meaningfully create documentation for single-script repositories, even if a ground-truth README weren't to exist.

6.2 Future Work

Several directions remain open for future work. First, extending the system to support small multi-file repositories. Also, additional static signals from areas like configuration files, environment variables, and lightweight control-flow analysis could also be interesting to test.

Beyond static analysis, future work could explore human intervention or additional LLM judges into the test, where developers or the LLM can review, correct, or approve extracted facts before README generation. This would help balance trust and oversight for the model while improving accuracy.

Acknowledgments

Our group would like to thank the CSE 5095 class for all of their input on this project we have done this semester. It mean't a lot to us for our production and overall results! Also, we'd like to give a special thanks to Professor Yu for giving us guidance and feedback as we have gone through the semester as well.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. doi:10.1145/3212695

[2] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[3] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210. doi:10.1145/3196321.3196334

[4] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models. arXiv:2407.07959 [cs.SE] https://arxiv.org/abs/2407.07959

[5] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685